Advanced Algorithms

I. Course Intro and Sorting Networks

Thomas Sauerwald
Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
List of Topics

IA Algorithms

IB Complexity Theory

II Advanced Algorithms

I. Course Intro and Sorting Networks
   Outline of this Course
List of Topics

IA Algorithms | IB Complexity Theory | II Advanced Algorithms

- I. Sorting Networks (Sorting, Counting)
- II. Linear Programming
- III. Approximation Algorithms: Covering Problems
- IV. Approximation Algorithms via Exact Algorithms
- V. Approximation Algorithms: Travelling Salesman Problem
- VI. Approximation Algorithms: Randomisation and Rounding
List of Topics

IA Algorithms
IB Complexity Theory
II Advanced Algorithms

I. Sorting Networks (Sorting, Counting)
II. Linear Programming
III. Approximation Algorithms: Covering Problems
IV. Approximation Algorithms via Exact Algorithms
V. Approximation Algorithms: Travelling Salesman Problem
VI. Approximation Algorithms: Randomisation and Rounding

closely follow CLRS3 and use the same numbering
however, slides will be self-contained
List of Topics

IA Algorithms

IB Complexity Theory

II Advanced Algorithms

I. Sorting Networks (Sorting, Counting)

II. Linear Programming

III. Approximation Algorithms: Covering Problems

IV. Approximation Algorithms via Exact Algorithms

V. Approximation Algorithms: Travelling Salesman Problem

VI. Approximation Algorithms: Randomisation and Rounding

closely follow CLRS3 and use the same numbering
however, slides will be self-contained
Outline

Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
maximize \[ 3x_1 + x_2 + 2x_3 \]
subject to
\[ \begin{align*}
x_1 + x_2 + 3x_3 &\leq 30 \\
2x_1 + 2x_2 + 5x_3 &\leq 24 \\
4x_1 + x_2 + 2x_3 &\leq 36 \\
x_1, x_2, x_3 &\geq 0
\end{align*} \]
maximize $3x_1 + x_2 + 2x_3$

subject to

$3x_1 + x_2 + 3x_3 \leq 30$
$2x_1 + 2x_2 + 5x_3 \leq 24$
$4x_1 + x_2 + 2x_3 \leq 36$
$x_1, x_2, x_3 \geq 0$
Linear Programming and Simplex

maximize \( 3x_1 + x_2 + 2x_3 \)
subject to
\[
\begin{align*}
3x_1 &+ 2x_2 + 5x_3 &\leq 30 \\
2x_1 &+ 2x_2 + 3x_3 &\leq 24 \\
4x_1 &+ x_2 + 2x_3 &\leq 36 \\
x_1, x_2, x_3 &\geq 0
\end{align*}
\]
maximize \( 3x_1 + x_2 + 2x_3 \)
subject to
\[
\begin{align*}
3x_1 & + 3x_3 & \leq 30 \\
2x_1 & + 5x_3 & \leq 24 \\
4x_1 & + 2x_3 & \leq 36 \\
x_1, x_2, x_3 & \geq 0
\end{align*}
\]
maximize $3x_1 + x_2 + 2x_3$

subject to

\[
\begin{align*}
    x_1 &+ x_2 &+ 3x_3 &\leq 30 \\
    2x_1 &+ 2x_2 &+ 5x_3 &\leq 24 \\
    4x_1 &+ x_2 &+ 2x_3 &\leq 36 \\
    x_1, x_2, x_3 &\geq 0
\end{align*}
\]
SOLUTION OF A LARGE-SCALE TRAVELING-SALESMAN PROBLEM

G. DANTZIG, R. FULKERSON, AND S. JOHNSON
The Rand Corporation, Santa Monica, California
(Received August 9, 1954)

It is shown that a certain tour of 49 cities, one in each of the 48 states and Washington, D. C., has the shortest road distance.

THE TRAVELING-SALESMAN PROBLEM might be described as follows: Find the shortest route (tour) for a salesman starting from a given city, visiting each of a specified group of cities, and then returning to the original point of departure. More generally, given an \( n \) by \( n \) symmetric matrix \( D = (d_{ij}) \), where \( d_{ij} \) represents the ‘distance’ from \( I \) to \( J \), arrange the points in a cyclic order in such a way that the sum of the \( d_{ij} \) between consecutive points is minimal. Since there are only a finite number of possibilities (at most \( \frac{1}{2} (n-1)! \)) to consider, the problem is to devise a method of picking out the optimal arrangement which is reasonably efficient for fairly large values of \( n \). Although algorithms have been devised for problems of similar nature, e.g., the optimal assignment problem,\(^3\,\,7\,\,8\) little is known about the traveling-salesman problem. We do not claim that this note alters the situation very much; what we shall do is outline a way of approaching the problem that sometimes, at least, enables one to find an optimal path and prove it so. In particular, it will be shown that a certain arrangement of 49 cities, one in each of the 48 states and Washington, D. C., is best, the \( d_{ij} \) used representing road distances as taken from an atlas.
1. Manchester, N. H. 
4. Cleveland, Ohio 
7. Indianapolis, Ind. 
8. Chicago, Ill. 
9. Milwaukee, Wis. 
10. Minneapolis, Minn. 
11. Pierre, S. D. 
12. Bismarck, N. D. 
13. Helena, Mont. 
15. Portland, Ore. 
16. Boise, Idaho 
17. Salt Lake City, Utah 
18. Carson City, Nev. 
19. Los Angeles, Calif. 
21. Santa Fe, N. M. 
22. Denver, Colo. 
23. Cheyenne, Wyo. 
24. Omaha, Neb. 
25. Des Moines, Iowa 
26. Kansas City, Mo. 
27. Topeka, Kans. 
28. Oklahoma City, Okla. 
29. Dallas, Tex. 
30. Little Rock, Ark. 
31. Memphis, Tenn. 
32. Jackson, Miss. 
33. New Orleans, La. 
34. Birmingham, Ala. 
35. Atlanta, Ga. 
36. Jacksonville, Fla. 
37. Columbia, S. C. 
38. Raleigh, N. C. 
40. Washington, D. C. 
42. Portland, Me. 
A. Baltimore, Md. 
B. Wilmington, Del. 
C. Philadelphia, Penn. 
D. Newark, N. J. 
E. New York, N. Y. 
F. Hartford, Conn. 
G. Providence, R. I.
Computing the Optimal Tour

maximize \( x_1 + x_2 \)
subject to
\[ 4x_1 - x_2 \leq 8 \]
\[ 2x_1 + x_2 \leq 10 \]
\[ 5x_1 - 2x_2 \leq 2 \]
\[ x_1, x_2 \geq 0 \]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

We are going to use our own implementation of the Simplex-Algorithm along with a visualization to solve a series of linear programs in order to solve the TSP instance optimally!
There are a couple of exercises spread across the recordings to test your understanding!
Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
### Overview: Sorting Networks

#### (Serial) Sorting Algorithms

- we already know several (comparison-based) sorting algorithms: Insertion sort, Bubble sort, Merge sort, Quick sort, Heap sort
- execute one operation at a time
- can handle arbitrarily large inputs
- sequence of comparisons is not set in advance
Overview: Sorting Networks

(Serial) Sorting Algorithms

- we already know several (comparison-based) sorting algorithms: Insertion sort, Bubble sort, Merge sort, Quick sort, Heap sort
- execute one operation at a time
- can handle arbitrarily large inputs
- sequence of comparisons is not set in advance

Sorting Networks

- only perform comparisons
- can only handle inputs of a fixed size
- sequence of comparisons is set in advance
Overview: Sorting Networks

(Serial) Sorting Algorithms

- we already know several (comparison-based) sorting algorithms: Insertion sort, Bubble sort, Merge sort, Quick sort, Heap sort
- execute one operation at a time
- can handle arbitrarily large inputs
- sequence of comparisons is not set in advance

Sorting Networks

- only perform comparisons
- can only handle inputs of a fixed size
- sequence of comparisons is set in advance
- Comparisons can be performed in parallel

Allows to sort \( n \) numbers in sublinear time!
Overview: Sorting Networks

(Serial) Sorting Algorithms

- we already know several (comparison-based) sorting algorithms: Insertion sort, Bubble sort, Merge sort, Quick sort, Heap sort
- execute one operation at a time
- can handle arbitrarily large inputs
- sequence of comparisons is not set in advance

Sorting Networks

- only perform comparisons
- can only handle inputs of a fixed size
- sequence of comparisons is set in advance
- Comparisons can be performed in parallel

Simple concept, but surprisingly deep and complex theory!

Allows to sort $n$ numbers in sublinear time!
Comparison Networks

- A comparison network consists solely of wires and comparators:
Comparison Networks

A comparison network consists solely of wires and comparators:
- comparator is a device with, on given two inputs, \( x \) and \( y \), returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)

\[\begin{align*}
\text{comparator} & \quad x' = \min(x, y) \\
\text{comparator} & \quad y' = \max(x, y)
\end{align*}\]

**Figure 27.1** (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7, y = 3 \) and outputs \( x' = 3, y' = 7 \) are shown.
Comparison Networks

- A comparison network consists solely of wires and comparators:
  - comparator is a device with, on given two inputs, \( x \) and \( y \), returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)

**Figure 27.1** (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7 \), \( y = 3 \) and outputs \( x' = 3 \), \( y' = 7 \) are shown.
Comparison Networks

- A **comparison network** consists solely of **wires** and **comparators**:
  - **comparator** is a device with, on given two inputs, \( x \) and \( y \), returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)
  - **wire** connect output of one comparator to the input of another

---

\[ x \rightarrow \text{comparator} \rightarrow x' = \min(x, y) \]
\[ y \rightarrow \text{comparator} \rightarrow y' = \max(x, y) \]

**Figure 27.1**  (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7 \), \( y = 3 \) and outputs \( x' = 3 \), \( y' = 7 \) are shown.
Comparison Networks

- A comparison network consists solely of wires and comparators:
  - Comparator is a device with, on given two inputs, \( x \) and \( y \), returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)
  - Wire connect output of one comparator to the input of another
  - Special wires: \( n \) input wires \( a_1, a_2, \ldots, a_n \) and \( n \) output wires \( b_1, b_2, \ldots, b_n \)

Figure 27.1  (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7 \), \( y = 3 \) and outputs \( x' = 3 \), \( y' = 7 \) are shown.
Comparison Networks

Comparison Network

- A comparison network consists solely of wires and comparators:
  - comparator is a device with, on given two inputs, x and y, returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)
  - wire connect output of one comparator to the input of another
  - special wires: \( n \) input wires \( a_1, a_2, \ldots, a_n \) and \( n \) output wires \( b_1, b_2, \ldots, b_n \)

Convention: use the same name for both a wire and its value.

\[
\begin{align*}
\text{comparator} & \quad x' = \min(x, y) \\
\text{comparator} & \quad y' = \max(x, y)
\end{align*}
\]

Figure 27.1  (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7, y = 3 \) and outputs \( x' = 3, y' = 7 \) are shown.
Comparison Networks

A comparison network consists solely of wires and comparators:
- comparator is a device with, on given two inputs, \( x \) and \( y \), returns two outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \)
- wire connects output of one comparator to the input of another
- special wires: \( n \) input wires \( a_1, a_2, \ldots, a_n \) and \( n \) output wires \( b_1, b_2, \ldots, b_n \)

A sorting network is a comparison network which works correctly (that is, it sorts every input).

\[
\begin{align*}
\text{Input} & \quad \rightarrow \quad \text{Comparator} \quad \rightarrow \quad \text{Output} \\
\quad x & \quad \rightarrow \quad x' = \min(x, y) \\
\quad y & \quad \rightarrow \quad y' = \max(x, y)
\end{align*}
\]

\[
\begin{align*}
\quad x & \quad \rightarrow \quad 7 \quad \rightarrow \quad 3 \quad \rightarrow \quad x' = \min(x, y) \\
\quad y & \quad \rightarrow \quad 3 \quad \rightarrow \quad 7 \quad \rightarrow \quad y' = \max(x, y)
\end{align*}
\]

Figure 27.1  (a) A comparator with inputs \( x \) and \( y \) and outputs \( x' \) and \( y' \). (b) The same comparator, drawn as a single vertical line. Inputs \( x = 7, y = 3 \) and outputs \( x' = 3, y' = 7 \) are shown.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1)
This network would not be a sorting network (Exercise 2)

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1)

This network would not be a sorting network (Exercise 2)

Depth of a wire:
Input wire has depth 0
If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$

Maximum depth of an output wire equals total running time

Interconnections between comparators must be acyclic

✓

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.
A horizontal line represents a sequence of distinct wires.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:

Input wire has depth 0.

If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be acyclic

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1)
This network would not be a sorting network (Exercise 2)

Depth of a wire:
Input wire has depth 0
If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Interconnections between comparators must be acyclic.

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.
A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).
This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths \(d_x\) and \(d_y\), then outputs have depth \(\max\{d_x, d_y\} + 1\).

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.

I. Course Intro and Sorting Networks

Introduction to Sorting Networks
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be acyclic

- Interconnections between comparators must be acyclic.
- A horizontal line represents a sequence of distinct wires.
- This network is in fact a sorting network (Exercise 1).
- This network would not be a sorting network (Exercise 2).
- Depth of a wire: Input wire has depth 0. If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.
- Maximum depth of an output wire equals total running time.
- Interconnections between comparators must be acyclic. Tracing back a path must never cycle back on itself and go through the same comparator twice.
A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Interconnections between comparators must be acyclic.

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths \( d_x \) and \( d_y \), then outputs have depth \( \max\{d_x, d_y\} + 1 \).

Maximum depth of an output wire equals total running time.

I. Course Intro and Sorting Networks

Introduction to Sorting Networks 12
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be acyclic

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1)

This network would not be a sorting network (Exercise 2)

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be **acyclic ✓**

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic ✓

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be acyclic.

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Input wire has depth 0.

If a comparator has two inputs of depths \( d_x \) and \( d_y \), then outputs have depth \( \max\{d_x, d_y\} + 1 \).

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.
Interconnections between comparators must be acyclic.

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).
This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be **acyclic**

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).
This network would not be a sorting network (Exercise 2).

**Depth of a wire:**
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be **acyclic**.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).
This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Interconnections between comparators must be **acyclic**

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

**Maximum depth of an output wire equals total running time**

Interconnections between comparators must be **acyclic**

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be acyclic

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:

Input wire has depth 0 if a comparator has two inputs of depths \( d_x \) and \( d_y \), then outputs have depth \( \max\{d_x, d_y\} + 1 \).

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Interconnections between comparators must be acyclic.

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

Interconnections between comparators must be **acyclic**

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).
This network would not be a sorting network (Exercise 2).

Depth of a wire:
Input wire has depth 0.
If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.
Interconnections between comparators must be acyclic.
Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
Input wire has depth 0.
If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

- Tracing back a path must never cycle back on itself and go through the same comparator twice.

F

I. Course Intro and Sorting Networks

Introduction to Sorting Networks
Example of a Comparison Network (Figure 27.2, CLRS2)

This network is in fact a sorting network (Exercise 1)
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:

- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.

This network would not be a sorting network (Exercise 2).
Example of a Comparison Network (Figure 27.2, CLRS2)

This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

Depth of a wire:

- Input wire has depth 0.
- If a comparator has two inputs of depths \(d_x\) and \(d_y\), then outputs have depth \(\max\{d_x, d_y\} + 1\).
- Maximum depth of an output wire equals total running time.
- Interconnections between comparators must be acyclic.
- Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic. Tracing back a path must never cycle back on itself and go through the same comparator twice.

**Depth** of a wire:
- Input wire has depth 0
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

Depth of a wire:

- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$
A horizontal line represents a sequence of distinct wires. This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

**Depth of a wire:**
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

\[ \text{Depth of a wire:} \]
- Input wire has depth 0
- If a comparator has two inputs of depths \( d_x \) and \( d_y \), then outputs have depth \( \max\{d_x, d_y\} + 1 \)
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths \(d_x\) and \(d_y\), then outputs have depth \(\max\{d_x, d_y\} + 1\)

Interconnections between comparators must be acyclic.
A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

**Depth** of a wire:

- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1). This network would not be a sorting network (Exercise 2).

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.
Example of a Comparison Network (Figure 27.2, CLRS2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network (Exercise 1).

This network would not be a sorting network (Exercise 2).

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.
Tracing back a path must never cycle back on itself and go through the same comparator twice.
Zero-One Principle

**Zero-One Principle**: A sorting network works correctly on arbitrary inputs if it works correctly on binary inputs.
Zero-One Principle: A sorting networks works correctly on arbitrary inputs if it works correctly on binary inputs.

Lemma 27.1

If a comparison network transforms the input \( a = \langle a_1, a_2, \ldots, a_n \rangle \) into the output \( b = \langle b_1, b_2, \ldots, b_n \rangle \), then for any monotonically increasing function \( f \), the network transforms \( f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \) into \( f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle \).
Zero-One Principle

Zero-One Principle: A sorting networks works correctly on arbitrary inputs if it works correctly on binary inputs.

Lemma 27.1

If a comparison network transforms the input \( a = \langle a_1, a_2, \ldots, a_n \rangle \) into the output \( b = \langle b_1, b_2, \ldots, b_n \rangle \), then for any monotonically increasing function \( f \), the network transforms \( f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \) into \( f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle \).

\[
\begin{align*}
\text{Figure 27.4} & \quad \text{The operation of the comparator in the proof of Lemma 27.1. The function } f \text{ is monotonically increasing.}
\end{align*}
\]
Zero-One Principle

Zero-One Principle: A sorting networks works correctly on arbitrary inputs if it works correctly on binary inputs.

Lemma 27.1

If a comparison network transforms the input $a = \langle a_1, a_2, \ldots, a_n \rangle$ into the output $b = \langle b_1, b_2, \ldots, b_n \rangle$, then for any monotonically increasing function $f$, the network transforms $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ into $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Theorem 27.2 (Zero-One Principle)

If a comparison network with $n$ inputs sorts all $2^n$ possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.
Proof of the Zero-One Principle

**Theorem 27.2 (Zero-One Principle)**

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.
Proof of the Zero-One Principle

**Theorem 27.2 (Zero-One Principle)**

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

**Proof:**

For the sake of contradiction, suppose the network does not correctly sort. Let \( a = \langle a_1, a_2, ..., a_n \rangle \) be the input with \( a_i < a_j \), but the network places \( a_j \) before \( a_i \) in the output.

Define a monotonically increasing function \( f \) as:

\[
\begin{align*}
f(x) &= 0 \text{ if } x \leq a_i, \\
f(x) &= 1 \text{ if } x > a_i.
\end{align*}
\]

Since the network places \( a_j \) before \( a_i \), by the previous lemma, \( f(a_j) \) is placed before \( f(a_i) \).

But \( f(a_j) = 1 \) and \( f(a_i) = 0 \), which contradicts the assumption that the network sorts all sequences of 0’s and 1’s correctly.
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:
- For the sake of contradiction, suppose the network does not correctly sort.
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let \( a = \langle a_1, a_2, \ldots, a_n \rangle \) be the input with \( a_i < a_j \), but the network places \( a_j \) before \( a_i \) in the output.
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let \( a = \langle a_1, a_2, \ldots, a_n \rangle \) be the input with \( a_i < a_j \), but the network places \( a_j \) before \( a_i \) in the output.
- Define a monotonically increasing function \( f \) as:
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let \( a = \langle a_1, a_2, \ldots, a_n \rangle \) be the input with \( a_i < a_j \), but the network places \( a_j \) before \( a_i \) in the output.
- Define a monotonically increasing function \( f \) as:

\[
f(x) = \begin{cases} 
0 & \text{if } x \leq a_i, \\
1 & \text{if } x > a_i.
\end{cases}
\]
Proof of the Zero-One Principle

**Theorem 27.2 (Zero-One Principle)**

If a comparison network with $n$ inputs sorts all $2^n$ possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

**Proof:**

- For the sake of contradiction, suppose the network does not correctly sort.
- Let $a = \langle a_1, a_2, \ldots, a_n \rangle$ be the input with $a_i < a_j$, but the network places $a_j$ before $a_i$ in the output.
- Define a monotonically increasing function $f$ as:
  
  $$f(x) = \begin{cases} 
  0 & \text{if } x \leq a_i, \\
  1 & \text{if } x > a_i. 
  \end{cases}$$

- Since the network places $a_j$ before $a_i$, by the previous lemma.
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let \( a = \langle a_1, a_2, \ldots, a_n \rangle \) be the input with \( a_i < a_j \), but the network places \( a_j \) before \( a_i \) in the output
- Define a monotonically increasing function \( f \) as:

\[
f(x) = \begin{cases} 
0 & \text{if } x \leq a_i, \\
1 & \text{if } x > a_i.
\end{cases}
\]

- Since the network places \( a_j \) before \( a_i \), by the previous lemma
  \( \Rightarrow f(a_j) \) is placed before \( f(a_i) \)
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with $n$ inputs sorts all $2^n$ possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let $a = \langle a_1, a_2, \ldots, a_n \rangle$ be the input with $a_i < a_j$, but the network places $a_j$ before $a_i$ in the output.
- Define a monotonically increasing function $f$ as:

  $$f(x) = \begin{cases} 
  0 & \text{if } x \leq a_i, \\
  1 & \text{if } x > a_i.
  \end{cases}$$

- Since the network places $a_j$ before $a_i$, by the previous lemma $\Rightarrow f(a_j)$ is placed before $f(a_i)$.
- But $f(a_j) = 1$ and $f(a_i) = 0$, which contradicts the assumption that the network sorts all sequences of 0’s and 1’s correctly. 

\[ \square \]
Some Basic (Recursive) Sorting Networks

These are Sorting Networks, but with depth $\Theta(n)$. 

$n$-wire Sorting Network
Some Basic (Recursive) Sorting Networks

These are Sorting Networks, but with depth $\Theta(n)$.
Some Basic (Recursive) Sorting Networks

These are Sorting Networks, but with depth $\Theta(n)$. 

$n$-wire Sorting Network
Some Basic (Recursive) Sorting Networks

These are Sorting Networks, but with depth $\Theta(n)$.
Some Basic (Recursive) Sorting Networks

These are Sorting Networks, but with depth $\Theta(n)$.
Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
Bitonic Sequences

A sequence is bitonic if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Sequences of one or two numbers are defined to be bitonic.
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples:
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**
- \( \langle 1, 4, 6, 8, 3, 2 \rangle \) ?

---

---
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**
- \(\langle 1, 4, 6, 8, 3, 2 \rangle\) ✓
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**
- \(\langle 1, 4, 6, 8, 3, 2 \rangle\) ✓
- \(\langle 6, 9, 4, 2, 3, 5 \rangle\) ?
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**
- \( \langle 1, 4, 6, 8, 3, 2 \rangle \ ✓ \\
- \( \langle 6, 9, 4, 2, 3, 5 \rangle \ ✓ \\

Bitonic Sequences
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples:

- $\langle 1, 4, 6, 8, 3, 2 \rangle$ ✓
- $\langle 6, 9, 4, 2, 3, 5 \rangle$ ✓
- $\langle 9, 8, 3, 2, 4, 6 \rangle$ ?
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples:

- \(1, 4, 6, 8, 3, 2\) ✓
- \(6, 9, 4, 2, 3, 5\) ✓
- \(9, 8, 3, 2, 4, 6\) ✓
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples:

- \(\langle 1, 4, 6, 8, 3, 2 \rangle\) ✓
- \(\langle 6, 9, 4, 2, 3, 5 \rangle\) ✓
- \(\langle 9, 8, 3, 2, 4, 6 \rangle\) ✓
- \(\langle 4, 5, 7, 1, 2, 6 \rangle\) ?
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples:

- \( \langle 1, 4, 6, 8, 3, 2 \rangle \) ✓
- \( \langle 6, 9, 4, 2, 3, 5 \rangle \) ✓
- \( \langle 9, 8, 3, 2, 4, 6 \rangle \) ✓
- \( \langle 4, 5, 7, 1, 2, 6 \rangle \)

---

Bitonic Sequence

- \( \langle 4, 5, 7, 1, 2, 6 \rangle \)
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**

- \( \langle 1, 4, 6, 8, 3, 2 \rangle \) ✓
- \( \langle 6, 9, 4, 2, 3, 5 \rangle \) ✓
- \( \langle 9, 8, 3, 2, 4, 6 \rangle \) ✓
- \( \langle 4, 5, 7, 1, 2, 6 \rangle \) **✗**
- binary sequences: ?
A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

**Examples:**

- \(\langle 1, 4, 6, 8, 3, 2 \rangle\) ✓
- \(\langle 6, 9, 4, 2, 3, 5 \rangle\) ✓
- \(\langle 9, 8, 3, 2, 4, 6 \rangle\) ✓
- \(\langle 4, 5, 7, 1, 2, 6 \rangle\) ✓
- Binary sequences: \(0^i1^j0^k\), or, \(1^i0^j1^k\), for \(i, j, k \geq 0\).
Towards Bitonic Sorting Networks

Half-Cleaner

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$. 
Towards Bitonic Sorting Networks

**Half-Cleaner**

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$.

We always assume that $n$ is even.
Towards Bitonic Sorting Networks

Half-Cleaner

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$. 
A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$. 

![Diagram of a half-cleaner comparison network]
Towards Bitonic Sorting Networks

Half-Cleaner

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$. 

I. Course Intro and Sorting Networks

Batcher’s Sorting Network
A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$. 

Half-Cleaner

If the input to a half-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the following properties:

- Both the top half and the bottom half are bitonic.
- Every element in the top is not larger than any element in the bottom.
- At least one half is clean.

**Lemma 27.3**

If the input to a half-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the following properties:

- Both the top half and the bottom half are bitonic.
- Every element in the top half is at least as small as every element of the bottom half.
- At least one half is clean.

**Proof**

The comparison network $H_{ALF-CLEANER}[n]$ consists of $n$ inputs and $n$ outputs. Without loss of generality, suppose that the input is of the form $00\ldots011\ldots100\ldots0$. (The situation in which the input is of the form $11\ldots100\ldots011\ldots1$ is symmetric.) There are possible cases depending upon the block of consecutive 0's or 1's in which the midpoint $n/2$ falls, and one of these cases (the one in which the midpoint occurs in the block of 1's) is further split into two cases. The four cases are shown in Figure 27.8. In each case shown, the lemma holds.
Towards Bitonic Sorting Networks

**Half-Cleaner**

A half-cleaner is a comparison network of depth 1 in which input wire \( i \) is compared with wire \( i + n/2 \) for \( i = 1, 2, \ldots, n/2 \).

**Lemma 27.3**

If the input to a half-cleaner is a bitonic sequence of 0’s and 1’s, then the output satisfies the following properties:

- both the top half and the bottom half are bitonic,
- every element in the top is not larger than any element in the bottom,
- at least one half is clean.

Proof: The comparison network HALFD-CLEANER \[ n \] compares inputs \( i \) and \( i + n/2 \) for \( i = 1, 2, \ldots, n/2 \). Without loss of generality, suppose that the input is of the form 00\ldots011\ldots100\ldots0. (The situation in which the input is of the form 11\ldots100\ldots011\ldots1 is symmetric.) There are possible cases depending upon the block of consecutive 0’s or 1’s in which the midpoint falls, and one of these cases (the one in which the midpoint occurs in the block of 1’s) is further split into two cases. The four cases are shown in Figure 27.8. In each case shown, the lemma holds.
Towards Bitonic Sorting Networks

Half-Cleaner

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$.

Lemma 27.3

If the input to a half-cleaner is a bitonic sequence of 0’s and 1’s, then the output satisfies the following properties:

- both the top half and the bottom half are bitonic,
- every element in the top is not larger than any element in the bottom,
- at least one half is clean.

Figure 27.7 shows $H_{ALF-CLEANER}$, the half-cleaner with 8 inputs and 8 outputs. When a bitonic sequence of 0’s and 1’s is applied as input to a half-cleaner, the half-cleaner produces an output sequence in which smaller values are in the top half, larger values are in the bottom half, and both halves are bitonic. In fact, at least one of the halves is clean — consisting of either all 0’s or all 1’s — and it is from this property that we derive the name “half-cleaner.” (Note that all clean sequences are bitonic.) The next lemma proves these properties of half-cleaners.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$. 

This suggests a recursive approach, since it now suffices to sort the top and bottom half separately.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$. 

![Diagram](image-url)

(a)
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$. 

- **(a)** Cases in which the division occurs in the middle subsequence of 1's.
- **(b)** Cases in which the division occurs in a subsequence of 0's. For all cases, every element in the top half of the output is at least as small as every element in the bottom half, both halves are bitonic, and at least one half is clean.

This suggests a recursive approach, since it now suffices to sort the top and bottom half separately.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$. 

The possible comparisons in $H_{ALF-CLEANER}$ (Figure 27.8). The input sequence is assumed to be a bitonic sequence of 0's and 1's, and without loss of generality, we assume that it is of the form $00\ldots011\ldots100\ldots0$. Subsequences of 0's are white, and subsequences of 1's are gray. We can think of the $n$ inputs as being divided into two halves such that for $i = 1, 2, \ldots, n/2$, inputs $i$ and $i+n/2$ are compared.

(a)–(b) Cases in which the division occurs in the middle subsequence of 1's.
(c)–(d) Cases in which the division occurs in a subsequence of 0's. For all cases, every element in the top half of the output is at least as small as every element in the bottom half, both halves are bitonic, and at least one half is clean.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$.

This suggests a recursive approach, since it now suffices to sort the top and bottom half separately.
The Bitonic Sorter

The comparison network \textsc{Bitonic-Sorter}[n], shown here for \( n = 8 \). (a) The recursive construction: \textsc{Half-Cleaner}[n] followed by two copies of \textsc{Bitonic-Sorter}[n/2] that operate in parallel. (b) The network after unrolling the recursion. Each half-cleaner is shaded. Sample zero-one values are shown on the wires.

\textbf{Figure 27.9}
The Bitonic Sorter

Figure 27.9 The comparison network BITONIC-SORTER[n], shown here for n = 8. (a) The recursive construction: HALF-CLEANER[n] followed by two copies of BITONIC-SORTER[n/2] that operate in parallel. (b) The network after unrolling the recursion. Each half-cleaner is shaded. Sample zero-one values are shown on the wires.

Recursive Formula for depth \( D(n) \):

\[
D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + 1 & \text{if } n = 2^k. 
\end{cases}
\]
The Bitonic Sorter

Recursive Formula for depth $D(n)$:

$$D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + 1 & \text{if } n = 2^k. 
\end{cases}$$

Henceforth we will always assume that $n$ is a power of 2.

Figure 27.9 The comparison network BITONIC-SORTER[$n$], shown here for $n = 8$. (a) The recursive construction: HALF-CLEANER[$n$] followed by two copies of BITONIC-SORTER[$n/2$] that operate in parallel. (b) The network after unrolling the recursion. Each half-cleaner is shaded. Sample zero-one values are shown on the wires.
The Bitonic Sorter

Figure 27.9 The comparison network BITONIC-SORTER[n], shown here for \( n = 8 \). (a) The recursive construction: HALF-CLEANER[n] followed by two copies of BITONIC-SORTER[n/2] that operate in parallel. (b) The network after unrolling the recursion. Each half-cleaner is shaded. Sample zero-one values are shown on the wires.

Recursive Formula for depth \( D(n) \):

\[
D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + 1 & \text{if } n = 2^k. 
\end{cases}
\]

Henceforth we will always assume that \( n \) is a power of 2.

BITONIC-SORTER[n] has depth \( \log n \) and sorts any zero-one bitonic sequence.
Merging Networks

- can merge two sorted input sequences into one sorted output sequence
- will be based on a modification of BITONIC-SORTER[n]
Merging Networks

- can merge two sorted input sequences into one sorted output sequence
- will be based on a modification of BITONIC-SORTER[n]

**Basic Idea:**

Consider two given sequences $X = 00000111$, $Y = 00001111$

Concatenating $X$ with $Y_R$ (the reversal of $Y$) will produce a bitonic sequence:

$$0000011111110000$$

Hence, in order to merge the sequences $X$ and $Y$, it suffices to perform a bitonic sort on $X$ concatenated with $Y_R$. 
Merging Networks

- can merge two sorted input sequences into one sorted output sequence
- will be based on a modification of BITONIC-SORTER[$n$]

**Basic Idea:**
- consider two given sequences $X = 00000111$, $Y = 0001111$
Merging Networks can merge two sorted input sequences into one sorted output sequence. It will be based on a modification of BITONIC-SORTER[$n$].

**Basic Idea:**
- Consider two given sequences $X = 00000111$, $Y = 00001111$
- Concatenating $X$ with $Y^R$ (the reversal of $Y$) $\Rightarrow$ 0000011111110000
Merging Networks

- can merge two sorted input sequences into one sorted output sequence
- will be based on a modification of BITONIC-SORTER[n]

Basic Idea:
- consider two given sequences $X = 00000111$, $Y = 00001111$
- concatenating $X$ with $Y^R$ (the reversal of $Y$) $\Rightarrow 0000111111110000$

This sequence is bitonic!
Merging Networks

- can merge **two sorted** input sequences into **one sorted** output sequence
- will be based on a modification of BITONIC-SORTER\[n\]

**Basic Idea:**
- consider two given sequences \(X = 00000111, Y = 00001111\)
- concatenating \(X\) with \(Y^R\) (the reversal of \(Y\)) \(\Rightarrow\) 0000011111110000

This sequence is bitonic!

Hence in order to merge the sequences \(X\) and \(Y\), it suffices to perform a **bitonic sort** on \(X\) concatenated with \(Y^R\).
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of \( \text{BITONIC-SORTER}[n] \) compares \( i \) and \( n/2 + i \)
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of BITONIC-SORTER\([n]\) compares \( i \) and \( n/2 + i \)

\[ \Rightarrow \text{First part of MERGER}\([n]\) compares inputs} \ i \ \text{and} \ n - i + 1 \ \text{for} \ i = 1, 2, \ldots, n/2 \]
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of BITONIC-SORTER[\( n \)] compares \( i \) and \( n/2 + i \)

\[ \Rightarrow \text{First part of MERGER}[n] \text{ compares inputs } i \text{ and } n - i + 1 \text{ for } i = 1, 2, \ldots, n/2 \]

**Figure 27.10** Comparing the first stage of MERGER[\( n \)] with HALF-CLEANER[\( n \)], for \( n = 8 \).
(a) The first stage of MERGER[\( n \)] transforms the two monotonic input sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \) into two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n \rangle \).
(b) The equivalent operation for HALF-CLEANER[\( n \)]. The bitonic input sequence \( \langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1} \rangle \) is transformed into the two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_n, b_{n-1}, \ldots, b_{n/2+1} \rangle \).
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of BITONIC-SORTER[\( n \)] compares \( i \) and \( n/2 + i \)

\[ \Rightarrow \text{First part of MERGER}[n] \text{ compares inputs } i \text{ and } n - i + 1 \text{ for } i = 1, 2, \ldots, n/2 \]

---

**Figure 27.10** Comparing the first stage of MERGER[\( n \)] with HALF-CLEANER[\( n \)], for \( n = 8 \).

(a) The first stage of MERGER[\( n \)] transforms the two monotonic input sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \) into two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n \rangle \).

(b) The equivalent operation for HALF-CLEANER[\( n \)]. The bitonic input sequence \( \langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1} \rangle \) is transformed into the two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_n, b_{n-1}, \ldots, b_{n/2+1} \rangle \).
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of BITONIC-SORTER\([n]\) compares \( i \) and \( n/2 + i \)

\[ \Rightarrow \text{First part of MERGER}[n] \text{ compares inputs } i \text{ and } n - i + 1 \text{ for } i = 1, 2, \ldots, n/2 \]
- Remaining part is identical to BITONIC-SORTER\([n]\)

**Figure 27.10** Comparing the first stage of MERGER\([n]\) with HALF-CLEANER\([n]\), for \( n = 8 \).

(a) The first stage of MERGER\([n]\) transforms the two monotonic input sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \) into two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n \rangle \).

(b) The equivalent operation for HALF-CLEANER\([n]\). The bitonic input sequence \( \langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1} \rangle \) is transformed into the two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_n, b_{n-1}, \ldots, b_{n/2+1} \rangle \).
Figure 27.11  A network that merges two sorted input sequences into one sorted output sequence. The network MERGER$[n]$ can be viewed as BITONIC-SORTER$[n]$ with the first half-cleaner altered to compare inputs $i$ and $n - i + 1$ for $i = 1, 2, \ldots, n/2$. Here, $n = 8$. (a) The network decomposed into the first stage followed by two parallel copies of BITONIC-SORTER$[n/2]$. (b) The same network with the recursion unrolled. Sample zero-one values are shown on the wires, and the stages are shaded.
Construction of a Sorting Network

Main Components

1. **Bitonic-Sorter** \([n]\)
   - sorts any bitonic sequence
   - depth \(\log n\)
Construction of a Sorting Network

Main Components

1. **BITONIC-SORTER \([n]\)**
   - sorts any bitonic sequence
   - depth \(\log n\)

2. **MERGER\([n]\)**
   - merges two sorted input sequences
   - depth \(\log n\)

---

I. Course Intro and Sorting Networks

Batcher’s Sorting Network
Construction of a Sorting Network

Main Components

1. **Bitonic-Sorter** [$n$]
   - sorts any bitonic sequence
   - depth $\log n$

2. **Merger** [$n$]
   - merges two sorted input sequences
   - depth $\log n$

Batcher’s Sorting Network

- **Sorter** [$n$] is defined recursively:
  - If $n = 2^k$, use two copies of **Sorter** [$n/2$] to sort two subsequences of length $n/2$ each. Then merge them using **Merger** [$n$].
  - If $n = 1$, network consists of a single wire.
Construction of a Sorting Network

Main Components

1. **Bitonic-Sorter**\([n]\)
   - sorts any bitonic sequence
   - depth \(\log n\)

2. **Merger**\([n]\)
   - merges two sorted input sequences
   - depth \(\log n\)

Batcher’s Sorting Network

- **Sorter**\([n]\) is defined recursively:
  - If \(n = 2^k\), use two copies of **Sorter**\([n/2]\) to sort two subsequences of length \(n/2\) each. Then merge them using **Merger**\([n]\).
  - If \(n = 1\), network consists of a single wire.

Can be seen as a parallel version of merge sort
Unrolling the Recursion (Figure 27.12)

Let $S_{k}$ be the depth of a sorting network with $2^k$ inputs. Suppose that we have $2^n$ numbers to be sorted and we know that every number is within $\Theta(\log n)$ of its correct position in the sorted order. Show that we can sort the $n$ numbers in depth $S_{k} + 2M_{k}$. 

**Figure 27.12**
Unrolling the Recursion (Figure 27.12)

I. Course Intro and Sorting Networks

1. Batcher’s Sorting Network

2. Depth of a Merging Network

Let $S_k$ be the depth of a sorting network with $2^k$ elements. Suppose that we have a sequence of $a_1, a_2, \ldots, a_n$ and wish to partition them into the smallest and the largest. Prove that we can do this in constant additional elements.

Show that the depth of $S_\log n$ is $2 \log n - \log \log n$.

Show that we can sort the $n$ numbers in depth $S(k) + 2M(k)$.

Let $S_k$ be the depth of a sorting network with $2^k$ numbers to be sorted and we know that every number is within $\log n + 1$ positions of its correct position in the sorted order. Show that we can sort the $n$ numbers in depth $S(k) + 2M(k)$.

The recursive construction.

Unrolling the recursion.

The sorting network $S_k$ has depth $\log n - \log \log n$.

The depth of a merging network with $2^k$ elements is $k$.

Figure 27.12
Unrolling the Recursion (Figure 27.12)

Let $S(k)$ be the depth of a sorting network with $2^k$ inputs, and let $M(k)$ be the depth of each comparator. The recursive construction.

The sorting network $S(n)$ has depth $\Theta(\log_2 n)$ and sorts any input.

<table>
<thead>
<tr>
<th>depth</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

I. Course Intro and Sorting Networks
Batcher's Sorting Network
Unrolling the Recursion (Figure 27.12)

Recursion for $D(n)$:

$$D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + \log n & \text{if } n = 2^k.
\end{cases}$$
Unrolling the Recursion (Figure 27.12)

Recursion for $D(n)$:

$$D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + \log n & \text{if } n = 2^k. 
\end{cases}$$

Solution: $D(n) = \Theta(\log^2 n)$. 

Let $27.5-4$ numbers to be sorted and we know that every number is within the smallest and the largest. Prove that we can do this in constant additional elements.

The recursive construction.

The sorting network $S_k$ is constructed recursively by merging two subsequences. Suppose that we have a sequence of $n$ inputs, and let $n$ exactly $a_1, a_2, \ldots, a_n$ be the inputs. Suppose that we have a sequence of $n$ inputs. Suppose that we have a sequence of inputs, and let $n_1 \leq n_2 \leq \ldots \leq n_k = n$. Then we can sort the $n$ numbers in depth $S(k) + 2M(k)$.
Recursion for $D(n)$:

$$D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + \log n & \text{if } n = 2^k.
\end{cases}$$

Solution: $D(n) = \Theta(\log^2 n)$.

**SORTER**[$n$] has depth $\Theta(\log^2 n)$ and sorts any input.
Outline

Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
A Glimpse at the AKS Network

There exists a sorting network with depth $O(\log n)$. 

Ajtai, Komlós, Szemerédi (1983)
A Glimpse at the AKS Network

Ajtai, Komlós, Szemerédi (1983)

There exists a sorting network with depth $O(\log n)$.

Quite elaborate construction, and involves huge constants.
There exists a sorting network with depth $O(\log n)$. 

Ajtai, Komlós, Szemerédi (1983)

A perfect halver is a comparison network that, given any input, places the $n/2$ smaller keys in $b_1, \ldots, b_{n/2}$ and the $n/2$ larger keys in $b_{n/2+1}, \ldots, b_n$. 

Perfect Halver

We will prove that such networks can be constructed in constant depth!
A Glimpse at the AKS Network

There exists a sorting network with depth $O(\log n)$.

Ajtai, Komlós, Szemerédi (1983)

A perfect halver is a comparison network that, given any input, places the $n/2$ smaller keys in $b_1, \ldots, b_{n/2}$ and the $n/2$ larger keys in $b_{n/2+1}, \ldots, b_n$.

Perfect halver of depth $\log n$ exist $\implies$ yields sorting networks of depth $\Theta((\log n)^2)$. 
A Glimpse at the AKS Network

Ajtai, Komlós, Szemerédi (1983)

There exists a sorting network with depth $O(\log n)$.

Perfect Halver

A perfect halver is a comparison network that, given any input, places the $n/2$ smaller keys in $b_1, \ldots, b_{n/2}$ and the $n/2$ larger keys in $b_{n/2+1}, \ldots, b_n$.

Approximate Halver

An $(n, \epsilon)$-approximate halver, $\epsilon < 1$, is a comparison network that for every $k = 1, 2, \ldots, n/2$ places at most $\epsilon k$ of its $k$ smallest keys in $b_{n/2+1}, \ldots, b_n$ and at most $\epsilon k$ of its $k$ largest keys in $b_1, \ldots, b_{n/2}$.
A Glimpse at the AKS Network

There exists a sorting network with depth $O(\log n)$.  

Ajtai, Komlós, Szemerédi (1983)

A perfect halver is a comparison network that, given any input, places the $n/2$ smaller keys in $b_1, \ldots, b_{n/2}$ and the $n/2$ larger keys in $b_{n/2+1}, \ldots, b_n$.

Perfect Halver

An $(n, \epsilon)$-approximate halver, $\epsilon < 1$, is a comparison network that for every $k = 1, 2, \ldots, n/2$ places at most $\epsilon k$ of its $k$ smallest keys in $b_{n/2+1}, \ldots, b_n$ and at most $\epsilon k$ of its $k$ largest keys in $b_1, \ldots, b_{n/2}$.

Approximate Halver

We will prove that such networks can be constructed in constant depth!

I. Course Intro and Sorting Networks Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
Expander Graphs

A bipartite \((n, d, \mu)\)-expander is a graph with:
- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is union of \(d\) perfect matchings
- For every subset \(S \subseteq V\) being in one part,

\[
|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}
\]
A bipartite \((n, d, \mu)\)-expander is a graph with:

- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is union of \(d\) perfect matchings
- For every subset \(S \subseteq V\) being in one part,

\[
|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}
\]
A bipartite \((n, d, \mu)\)-expander is a graph with:

- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is union of \(d\) perfect matchings
- For every subset \(S \subseteq V\) being in one part,

\[|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}\]
Expander Graphs

A bipartite $(n, d, \mu)$-expander is a graph with:
- $G$ has $n$ vertices ($n/2$ on each side)
- the edge-set is union of $d$ perfect matchings
- For every subset $S \subseteq V$ being in one part,

$$|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}$$
**Expander Graphs**

A bipartite \((n, d, \mu)\)-expander is a graph with:
- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is union of \(d\) perfect matchings
- For every subset \(S \subseteq V\) being in one part,

\[
|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}
\]

Specific definition tailored for sorting network - many other variants exist!
Expander Graphs

A bipartite \((n, d, \mu)\)-expander is a graph with:
- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is union of \(d\) perfect matchings
- For every subset \(S \subseteq V\) being in one part,
  \[|N(S)| > \min\{\mu \cdot |S|, n/2 - |S|\}\]

Expander Graphs:
- **probabilistic construction** “easy”: take \(d\) (disjoint) random matchings
- **explicit construction** is a deep mathematical problem with ties to number theory, group theory, combinatorics etc.
- **many applications** in networking, complexity theory and coding theory
From Expanders to Approximate Halvers

I. Course Intro and Sorting Networks

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
From Expanders to Approximate Halvers

\[ L \quad R \]

I. Course Intro and Sorting Networks
Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
From Expanders to Approximate Halvers

$L$ $R$

I. Course Intro and Sorting Networks
Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
From Expanders to Approximate Halvers

\[ L \quad R \]
From Expanders to Approximate Halvers
From Expanders to Approximate Halvers

I. Course Intro and Sorting Networks
Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
From Expanders to Approximate Halvers
From Expanders to Approximate Halvers

I. Course Intro and Sorting Networks

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)
Existence of Approximate Halvers (non-examinable)

Proof:

\( X := \text{keys with the } k \text{ smallest inputs} \)

\( Y := \text{wires in lower half with } k \text{ smallest outputs} \)

For every \( u \in N(Y) \):

\[ \exists \text{comparat.} (u, v), v \in Y \]

Let \( u_t, v_t \) be their keys after the comparator

Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))

Further:

\[ u_d \leq u_t \leq v_t \leq v_d \]

\[ \Rightarrow u_d \in X \]

Since \( u \) was arbitrary:

\[ |Y| + |N(Y)| \leq k. \]

Since \( G \) is a bipartite \((n, d, \mu)\)-expander:

\[ |Y| + |N(Y)| > |Y| + \min\{\mu |Y|, n/2 - |Y|\} = \min\{\mu |Y|, n/2\}. \]

Combining the two bounds above yields:

\[ (1 + \mu) |Y| \leq k. \]

Same argument \( \Rightarrow \) at most \( \epsilon \cdot k \), \( \epsilon := 1 / (\mu + 1) \), of the \( k \) largest input keys are placed in \( b_1, ..., b_{n/2} \).

Here we used that \( k \leq n/2 \).
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs

\( Y := \) wires in lower half with \( k \) smallest outputs

For every \( u \in N(Y) \):

\[ \exists \text{ comparat. } (u, v), v \in Y \]

Let \( u_t, v_t \) be their keys after the comparator.

Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \)).

Further:

\[ u_d \leq u_t \leq v_t \leq v_d \implies u_d \in X \]

Since \( u \) was arbitrary:

\[ |Y| + |N(Y)| \leq k. \]

Since \( G \) is a bipartite \((n, d, \mu)\)-expander:

\[ |Y| + |N(Y)| > |Y| + \min\{\mu |Y|, n/2 - |Y|\} = \min\{1 + \mu |Y|, n/2\}. \]

Combining the two bounds above yields:

\[ (1 + \mu) |Y| \leq k. \]

Same argument \( \Rightarrow \) at most \( \epsilon \cdot k \), \( \epsilon := 1/(\mu + 1) \), of the \( k \) largest input keys are placed in \( b_1, \ldots, b_{n/2} \).

Here we used that \( k \leq n/2 \).
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs

For every \( u \in N(Y) \):

\[ \exists \text{comparat.} (u, v), v \in Y \]

Let \( u_t, v_t \) be their keys after the comparator

Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))

Further:

\[ u_d \leq u_t \leq v_t \leq v_d \]

Since \( u \) was arbitrary:

\[ |Y| + |N(Y)| \leq k. \]

Since \( G \) is a bipartite \((n, d, \mu)\)-expander:

\[ |Y| + |N(Y)| > |Y| + \min\{\mu, n/2 - |Y|\} = \min\{(1 + \mu)|Y|, n/2\}. \]

Combining the two bounds above yields:

\[ (1 + \mu)|Y| \leq k. \]

Same argument \(\Rightarrow\) at most \(\epsilon \cdot k\), \(\epsilon = 1/\left(\mu + 1\right)\), of the \(k\) largest input keys are placed in \(b_1, \ldots, b_n/2\). Here we used that \(k \leq n/2\).
Existence of Approximate Halvers (non-examinable)

Proof:

- $X := \text{keys with the } k \text{ smallest inputs}$
- $Y := \text{wires in lower half with } k \text{ smallest outputs}$
- For every $u \in N(Y)$: $\exists$ comparat. $(u, v)$, $v \in Y$
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X \) := keys with the \( k \) smallest inputs
- \( Y \) := wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \((u, v), v \in Y\)
- Let \( u_t, v_t \) be their keys after the comparator
- Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \( (u, v), v \in Y \)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))

The diagram illustrates the sorting process with wires and keys, showing the flow of data through the comparator and the eventual separation into two halves based on the comparison.
Existence of Approximate Halvers (non-examinable)

Proof:

- $X :=$ keys with the $k$ smallest inputs
- $Y :=$ wires in lower half with $k$ smallest outputs
- For every $u \in N(Y)$: $\exists$ comparat. $(u, v), v \in Y$
- Let $u_t, v_t$ be their keys after the comparator
  Let $u_d, v_d$ be their keys at the output (note $v_d \in X$)

Further:

Further:

$u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X$

Since $u$ was arbitrary:

$|Y| + |N(Y)| \leq k$.

Since $G$ is a bipartite $(n, d, \mu)$-expander:

$|Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\} = \min\{(1+\mu)|Y|, n/2\}$.

Combining the two bounds above yields:

$(1+\mu)|Y| \leq k$.

Same argument $\Rightarrow$ at most $\epsilon \cdot k$, $\epsilon := 1/(\mu + 1)$, of the $k$ largest input keys are placed in $b_1, \ldots, b_{n/2}$.

Here we used that $k \leq n/2$. 

Typical application of expander graphs in parallel algorithms.

Much more work needed to construct the AKS sorting network.
Existence of Approximate Halvers (non-examinable)

Proof:

- $X := \text{keys with the } k \text{ smallest inputs}$
- $Y := \text{wires in lower half with } k \text{ smallest outputs}$
- For every $u \in N(Y): \exists \text{ comparat. } (u, v), v \in Y$
- Let $u_t, v_t$ be their keys after the comparator
  Let $u_d, v_d$ be their keys at the output (note $v_d \in X$)
- Further: $u_d \leq u_t \leq v_t \leq v_d$
Existence of Approximate Halvers (non-examinable)

Proof:

- $X :=$ keys with the $k$ smallest inputs
- $Y :=$ wires in lower half with $k$ smallest outputs
- For every $u \in N(Y)$: $\exists$ comparat. $(u, v), v \in Y$
- Let $u_t, v_t$ be their keys after the comparator
  - Let $u_d, v_d$ be their keys at the output (note $v_d \in X$)
- Further: $u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X$
- Since $u$ was arbitrary:
  \[ |Y| + |N(Y)| \leq k. \]
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \((u, v), v \in Y\)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:

\[
|Y| + |N(Y)| \leq k.
\]

- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \text{keys with the } k \text{ smallest inputs} \)
- \( Y := \text{wires in lower half with } k \text{ smallest outputs} \)
- For every \( u \in N(Y) \): \( \exists \text{ comparat. } (u, v), v \in Y \)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[ |Y| + |N(Y)| \leq k. \]
- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
  \[ |Y| + |N(Y)| \leq k. \]
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \( (u, v), v \in Y \)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[
  |Y| + |N(Y)| \leq k.
  \]
- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
  \[
  |Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\}
  \]
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \( (u, v) \), \( v \in Y \)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[ |Y| + |N(Y)| \leq k. \]
- Since \( G \) is a bipartite \( (n, d, \mu) \)-expander:
  \[ |Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\} \]
  \[ = \min\{(1 + \mu)|Y|, n/2\}. \]
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y) \): \( \exists \) comparat. \((u,v), v \in Y\)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[ |Y| + |N(Y)| \leq k. \]
- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
  \[ |Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\} \]
  \[ = \min\{(1 + \mu)|Y|, n/2\}. \]
- Combining the two bounds above yields:
  \[ (1 + \mu)|Y| \leq k. \]
Existence of Approximate Halvers (non-examinable)

Proof:

- \( X := \) keys with the \( k \) smallest inputs
- \( Y := \) wires in lower half with \( k \) smallest outputs
- For every \( u \in N(Y): \exists \) comparat. \((u, v), v \in Y\)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output (note \( v_d \in X \))
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[
  |Y| + |N(Y)| \leq k.
  \]
- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
  \[
  |Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\}
  = \min\{(1 + \mu)|Y|, n/2\}.
  \]
- Combining the two bounds above yields:
  \[
  (1 + \mu)|Y| \leq k.
  \]

Here we used that \( k \leq n/2 \)
Existence of Approximate Halvers (non-examinable)

Proof:

- $X := \text{keys with the } k \text{ smallest inputs}$
- $Y := \text{wires in lower half with } k \text{ smallest outputs}$
- For every $u \in N(Y)$: $\exists$ comparat. $(u, v), v \in Y$
- Let $u_t, v_t$ be their keys after the comparator
  Let $u_d, v_d$ be their keys at the output (note $v_d \in X$)
- Further: $u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X$
- Since $u$ was arbitrary:
  \[ |Y| + |N(Y)| \leq k.\]
- Since $G$ is a bipartite $(n, d, \mu)$-expander:
  \[
  |Y| + |N(Y)| > |Y| + \min\{\mu|Y|, n/2 - |Y|\}
  = \min\{(1 + \mu)|Y|, n/2\}.
  \]
- Combining the two bounds above yields:
  \[ (1 + \mu)|Y| \leq k. \]
- Same argument $\Rightarrow$ at most $\epsilon \cdot k$,
  $\epsilon := 1/(\mu + 1)$, of the $k$ largest input keys are placed in $b_1, \ldots, b_{n/2}$.

- typical application of expander graphs in parallel algorithms
- Much more work needed to construct the AKS sorting network
Donald E. Knuth (Stanford)

“Batcher’s method is much better, unless \( n \) exceeds the total memory capacity of all computers on earth!”

Richard J. Lipton (Georgia Tech)

“The AKS sorting network is galactic: it needs that \( n \) be larger than \( 2^{78} \) or so to finally be smaller than Batcher’s network for \( n \) items.”
Outline of this Course

Some Highlights

Introduction to Sorting Networks

Batcher’s Sorting Network

Bonus Material: Construction of an Optimal Sorting Network (non-examinable)

Counting Networks
Siblings of Sorting Network

- Sorting Networks
  - sorts any input of size $n$
  - special case of Comparison Networks

Diagram of a comparator: 7 < 2

Switching (Shuffling) Networks balance any stream of tokens over $n$ wires

Special case of Balancing Networks

Counting Networks
Siblings of Sorting Network

**Sorting Networks**
- sorts any input of size $n$
- special case of Comparison Networks

**Switching (Shuffling) Networks**
- creates a random permutation of $n$ items
- special case of Permutation Networks
Siblings of Sorting Network

Sorting Networks
- sorts any input of size \( n \)
- special case of Comparison Networks

Switching (Shuffling) Networks
- creates a random permutation of \( n \) items
- special case of Permutation Networks

Counting Networks
- balances any stream of tokens over \( n \) wires
- special case of Balancing Networks
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Values could represent addresses in memories or destinations on an interconnection network.
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)
### Counting Network

#### Distributed Counting

Processors collectively assign successive values from a given range.

#### Balancing Networks

- constructed in a similar manner like *sorting networks*
- instead of comparators, consists of *balancers*
- *balancers* are *asynchronous* flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

![Diagram of a balancer network](image-url)
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top,...)
**Counting Network**

---

**Distributed Counting**

Processors collectively assign successive values from a given range.

---

**Balancing Networks**

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

---

I. Course Intro and Sorting Networks Counting Networks
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, . . .)
Counting Network

Distributed Counting
Processors collectively assign successive values from a given range.

Balancing Networks
- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, . . .)
**Counting Network**

---

**Distributed Counting**

Processors collectively assign successive values from a given range.

---

**Balancing Networks**

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- **balancers** are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

---

**Diagram**

A diagram illustrating the flow of data through a balancing network, showing the flow from inputs to one of two outputs, alternating between the top and bottom outputs.
Counting Network

- Distributed Counting
  Processors collectively assign successive values from a given range.

- Balancing Networks
  - constructed in a similar manner like sorting networks
  - instead of comparators, consists of balancers
  - balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top,...)
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

![Diagram of Balancing Networks]
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

![Diagram of a balancing network with balancers]
Counting Network

Distributed Counting
Processors collectively assign successive values from a given range.

Balancing Networks
- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, . . .)
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, . . .)

Number of tokens differs by at most one
1. Let $x_1, x_2, \ldots, x_n$ be the number of tokens (ever received) on the designated input wires

2. Let $y_1, y_2, \ldots, y_n$ be the number of tokens (ever received) on the designated output wires
Counting Network (Formal Definition)

1. Let $x_1, x_2, \ldots, x_n$ be the number of tokens (ever received) on the designated input wires.
2. Let $y_1, y_2, \ldots, y_n$ be the number of tokens (ever received) on the designated output wires.
3. In a quiescent state: $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$.
4. A counting network is a balancing network with the step-property:

$$0 \leq y_i - y_j \leq 1 \text{ for any } i < j.$$
Bitonic Counting Network

Counting Network (Formal Definition)

1. Let \( x_1, x_2, \ldots, x_n \) be the number of tokens (ever received) on the designated input wires
2. Let \( y_1, y_2, \ldots, y_n \) be the number of tokens (ever received) on the designated output wires
3. In a quiescent state: \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \)
4. A counting network is a balancing network with the step-property:
   \[ 0 \leq y_i - y_j \leq 1 \text{ for any } i < j. \]

Bitonic Counting Network: Take Batcher’s Sorting Network and replace each comparator by a balancer.
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$

2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.

3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$. 

Facts

Proof (by induction on $n$ being a power of 2)

Case $n = 2$ is clear, since $\text{MERGER}[2]$ is a single balancer.

Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the $\text{MERGER}[n/2]$ subnetworks.

IH $\Rightarrow z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ have the step property.

Let $Z := \sum_{i=1}^{n/2} z_i$ and $Z' := \sum_{i=1}^{n/2} z'_i$.

Claim: $|Z - Z'| \leq 1$ (since $Z' = \lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$).

Case 1: If $Z = Z'$, then F2 implies the output of $\text{MERGER}[n]$ is $y_i = z_1 + \lfloor \frac{(i-1)}{2} \rfloor \checkmark$

Case 2: If $|Z - Z'| = 1$, F3 implies $z_i = z'_i$ for $i = 1, \ldots, n/2$ except a unique $j$ with $z_j \neq z'_j$.

Balancer between $z_j$ and $z'_j$ will ensure that the step property holds.
Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$.
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists i = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Consider a MERGER[$n$]. Then if the inputs $x_1, \ldots, x_{n/2}$ and $x_{n/2+1}, \ldots, x_n$ have the step property, then so does the output $y_1, \ldots, y_n$.

Proof (by induction on $n$ being a power of 2)
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor$, and $\sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil$.
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)
- Case $n = 2$ is clear, since MERGER[2] is a single balancer.
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$

2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.

3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer
- $n > 2$:
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$ and $\sum_{i=1}^{n/2} x_{2i} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists ! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[$n/2$] subnetworks
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$.
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer.
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[n/2] subnetworks.
Facts

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have \( \sum_{i=1}^{n/2} x_{2i-1} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil \) and \( \sum_{i=1}^{n/2} x_{2i} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor \).

2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).

3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

Proof (by induction on \( n \) being a power of 2)

- Case \( n = 2 \) is clear, since \textsc{Merger}[2] is a single balancer.
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the \textsc{Merger}[\( n/2 \)] subnetworks.
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\frac{1}{2} \sum_{i=1}^{n/2} x_{2i-1} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$, and $\frac{1}{2} \sum_{i=1}^{n/2} x_{2i} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$.

2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.

3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer.
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[$n/2$] subnetworks.
Correctness of the Bitonic Counting Network (non-examinable)

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have 
   \[
   \sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor, \quad \text{and} \quad \sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil
   \]

2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).

3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( i \neq j \).

**Proof (by induction on \( n \) being a power of 2)**

- **Case** \( n = 2 \) is clear, since \( \text{MERGER}[2] \) is a single balancer
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the \( \text{MERGER}[n/2] \) subnetworks
Correctness of the Bitonic Counting Network (non-examinable)

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have \( \sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil \), and \( \sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor \).
2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).
3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

Proof (by induction on \( n \) being a power of 2)

- Case \( n = 2 \) is clear, since \textsc{MERGER}[2] is a single balancer
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the \textsc{MERGER}[n/2] subnetworks
- IH \( \Rightarrow z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) have the step property
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor$, and $\sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil$
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[n/2] subnetworks
- IH $\Rightarrow z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ have the step property
- Let $Z := \sum_{i=1}^{n/2} z_i$ and $Z' := \sum_{i=1}^{n/2} z'_i$
Correctness of the Bitonic Counting Network (non-examinable)

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have 
   \[
   \sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor, \quad \text{and} \quad \sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil
   \]

2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).

3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

Proof (by induction on \( n \) being a power of 2)

- Case \( n = 2 \) is clear, since MERGER[2] is a single balancer.
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the MERGER\([n/2]\) subnetworks.
  - IH \( \Rightarrow \) \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) have the step property.
  - Let \( Z := \sum_{i=1}^{n/2} z_i \) and \( Z' := \sum_{i=1}^{n/2} z'_i \).
  - Claim: \( |Z - Z'| \leq 1 \) (since \( Z' = \left\lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \right\rfloor + \left\lceil \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \right\rceil \))
Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have \( \sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor \), and \( \sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil \)
2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).
3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

Proof (by induction on \( n \) being a power of 2)

- Case \( n = 2 \) is clear, since MERGER[2] is a single balancer
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the MERGER[n/2] subnetworks
- IH \( \Rightarrow z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) have the step property
- Let \( Z := \sum_{i=1}^{n/2} z_i \) and \( Z' := \sum_{i=1}^{n/2} z'_i \)
- Claim: \( |Z - Z'| \leq 1 \) (since \( Z' = \left\lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \right\rfloor + \left\lceil \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \right\rceil \))
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$.
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists ! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer.
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[$n/2$] subnetworks.
- IH $\Rightarrow z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ have the step property.
- Let $Z := \sum_{i=1}^{n/2} z_i$ and $Z' := \sum_{i=1}^{n/2} z'_i$.
- Claim: $|Z - Z'| \leq 1$ (since $Z' = \lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \rceil$).
- Case 1: If $Z = Z'$, then F2 implies the output of MERGER[$n$] is $y_i = z_{1+\lfloor (i-1)/2 \rfloor}$. \(\square\)
Correctness of the Bitonic Counting Network (non-examinable)

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor$, and $\sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil$
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$ being a power of 2)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[$n/2$] subnetworks
- IH $\Rightarrow$ $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ have the step property
- Let $Z := \sum_{i=1}^{n/2} z_i$ and $Z' := \sum_{i=1}^{n/2} z'_i$
- Claim: $|Z - Z'| \leq 1$ (since $Z' = \left\lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \right\rfloor + \left\lceil \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \right\rceil$)
- Case 1: If $Z = Z'$, then F2 implies the output of MERGER[$n$] is $y_i = z_{1+\lfloor(i-1)/2\rfloor}$ ✓
- Case 2: If $|Z - Z'| = 1$, F3 implies $z_i = z'_i$ for $i = 1, \ldots, n/2$ except a unique $j$ with $z_j \neq z'_j$.
  Balancer between $z_j$ and $z'_j$ will ensure that the step property holds.
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$.
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n,...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i$, $i + n$, $i + 2 \cdot n$, ...

Bitonic Counting Network in Action (Asynchronous Execution)
Counting can be done as follows:

Add local counter to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$
Counting can be done as follows: add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \times n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2 \cdot n,...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n,...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i$, $i + n$, $i + 2 \cdot n$, ...
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: Add local counter to each output wire \(i\), to assign consecutive numbers \(i, i + n, i + 2 \cdot n, \ldots\).
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
- Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows:

Add local counter to each output wire, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: Add local counter to each output wire, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$.
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n,...$. 

I. Course Intro and Sorting Networks Counting Networks
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n,...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$.

Bitonic Counting Network in Action (Asynchronous Execution)
Counting can be done as follows: Add local counter to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$. 

I. Course Intro and Sorting Networks

Counting Networks
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2n, ...$. 
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2n, ...$.
Counting can be done as follows:

Add local counter to each output wire $i$ to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: add local counter to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n, ...$.
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$.

I. Course Intro and Sorting Networks

Counting Networks
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n, ...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2n, \ldots$.
Counting can be done as follows: Add local counter to each output wire, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, \ldots$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$.
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:

1. Add a local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire, to assign consecutive numbers $i$, $i + n$, $i + 2 \cdot n$, ...
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire \(i\), to assign consecutive numbers \(i, i + n, i + 2 \cdot n, \ldots\).
Counting can be done as follows: Add local counter to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
Counting can be done as follows: Add local counter to each output wire \(i\), to assign consecutive numbers \(i, i+n, i+2 \cdot n, \ldots\).
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows: Add local counter to each output wire \(i\), to assign consecutive numbers \(i, i+n, i+2n, \ldots\).
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n, ...$.
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$. 

I. Course Intro and Sorting Networks Counting Networks
Bitonic Counting Network in Action (Asynchronous Execution)

Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
Counting can be done as follows:
Add local counter to each output wire \(i\), to assign consecutive numbers \(i, i+n, i+2\cdot n, \ldots\)
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$.
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:

Add local counter to each output wire $i$ to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2n, ...$
Counting can be done as follows: Add local counter to each output wire $i$ to assign consecutive numbers $i, i + n, i + 2 \cdot n, ...$
Counting can be done as follows:
Add **local counter** to each output wire \( i \), to assign consecutive numbers \( i, i + n, i + 2 \cdot n, \ldots \).
A Periodic Counting Network [Aspnes, Herlihy, Shavit, JACM 1994]

Consists of $\log n$ BLOCK $[n]$ networks each of which has depth $\log n$
A Periodic Counting Network [Aspnes, Herlihy, Shavit, JACM 1994]

Consists of $\log n$ BLOCK[$n$] networks each of which has depth $\log n$
If a network is a counting network, then it is also a sorting network.

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

The converse is not true!
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.
If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
From Counting to Sorting

If a network is a counting network, then it is also a sorting network.

Proof.
- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.
- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$. 

![Diagram of C and S networks]
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.

\[ C \]

\[ S \]
Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires

![Diagram of counting and sorting networks](image-url)
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\implies$ all ones will be routed to the lower wires.

![Diagram of C and S networks]

- $S$ corresponds to $C$ $\implies$ all zeros will be routed to the lower wires.
- By the Zero-One Principle, $S$ is a sorting network.
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.

Proof Diagram:

\[ C \]

\begin{align*}
0 & \quad 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{align*}

\[ S \]

\begin{align*}
1 & \\
0 & \\
0 & \\
1 & \\
\end{align*}
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let \( C \) be a counting network, and \( S \) be the corresponding sorting network.
- Consider an input sequence \( a_1, a_2, \ldots, a_n \in \{0, 1\}^n \) to \( S \).
- Define an input \( x_1, x_2, \ldots, x_n \in \{0, 1\}^n \) to \( C \) by \( x_i = 1 \) iff \( a_i = 0 \).
- \( C \) is a counting network \( \Rightarrow \) all ones will be routed to the lower wires.
- \( S \) corresponds to \( C \) \( \Rightarrow \) all zeros will be routed to the lower wires.

![Diagram showing the process of routing ones and zeros in \( C \) and \( S \).]
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.
- $S$ corresponds to $C$ $\Rightarrow$ all zeros will be routed to the lower wires.
From Counting to Sorting

Counting vs. Sorting

If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.
- $S$ corresponds to $C$ $\Rightarrow$ all zeros will be routed to the lower wires.
- By the Zero-One Principle, $S$ is a sorting network.

\[ C \quad \begin{array}{cccccc}
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{array} \quad S \quad \begin{array}{ccc}
1 & 0 \\
0 & 0 \\
0 & 1 \\
1 & 1
\end{array} \]
Exercise: Consider a network which is a sorting network, but not a counting network.
**Hint:** Try to find a simple network with 4 wires that corresponds to a basic sequential sorting algorithm.
II. Linear Programming

Thomas Sauerwald

Easter 2021
Introduction

Formulating Problems as Linear Programs

Standard and Slack Forms

Simplex Algorithm

Finding an Initial Solution
- linear programming is a powerful tool in optimisation
- inspired more sophisticated techniques such as quadratic optimisation, convex optimisation, integer programming and semi-definite programming
- we will later use the connection between linear and integer programming to tackle several problems (Vertex-Cover, Set-Cover, TSP, satisfiability)
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equality
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equalities

Example: Political Advertising (from CLRS3)
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equalities

Example: Political Advertising (from CLRS3)

- Imagine you are a politician trying to win an election
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equalities

Example: Political Advertising (from CLRS3)

- Imagine you are a politician trying to win an election
- Your district has three different types of areas: Urban, suburban and rural, each with, respectively, 100,000, 200,000 and 50,000 registered voters
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equality

Example: Political Advertising (from CLRS3)

- Imagine you are a politician trying to win an election
- Your district has three different types of areas: Urban, suburban and rural, each with, respectively, 100,000, 200,000 and 50,000 registered voters
- **Aim:** at least half of the registered voters in each of the three regions should vote for you
What are Linear Programs?

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equalities

Example: Political Advertising (from CLRS3)

- Imagine you are a politician trying to win an election
- Your district has three different types of areas: Urban, suburban and rural, each with, respectively, 100,000, 200,000 and 50,000 registered voters
- Aim: at least half of the registered voters in each of the three regions should vote for you
- Possible Actions: Advertise on one of the primary issues which are (i) building more roads, (ii) gun control, (iii) farm subsidies and (iv) a gasoline tax dedicated to improve public transit.
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

### Possible Solution:
- $20,000 on advertising to building roads
- $0 on advertising to gun control
- $4,000 on advertising to farm subsidies
- $9,000 on advertising to a gasoline tax

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>
### Political Advertising Continued

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

**Possible Solution:**
- $20,000 on advertising to building roads
- $0 on advertising to gun control
- $4,000 on advertising to farm subsidies
- $9,000 on advertising to a gasoline tax

**Total cost: $33,000**
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

**Possible Solution:**
- $20,000 on advertising to building roads
- $0 on advertising to gun control
- $4,000 on advertising to farm subsidies
- $9,000 on advertising to a gasoline tax

**Total cost: $33,000**

What is the best possible strategy?
Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.
Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 =$ number of thousands of dollars spent on advertising on building roads
- $x_2 =$ number of thousands of dollars spent on advertising on gun control
- $x_3 =$ number of thousands of dollars spent on advertising on farm subsidies
- $x_4 =$ number of thousands of dollars spent on advertising on gasoline tax
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1$ = number of thousands of dollars spent on advertising on building roads
- $x_2$ = number of thousands of dollars spent on advertising on gun control
- $x_3$ = number of thousands of dollars spent on advertising on farm subsidies
- $x_4$ = number of thousands of dollars spent on advertising on gasoline tax

Constraints:
Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 =$ number of thousands of dollars spent on advertising on building roads
- $x_2 =$ number of thousands of dollars spent on advertising on gun control
- $x_3 =$ number of thousands of dollars spent on advertising on farm subsidies
- $x_4 =$ number of thousands of dollars spent on advertising on gasoline tax

Constraints:
- $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$
Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 = \text{number of thousands of dollars spent on advertising on building roads}$
- $x_2 = \text{number of thousands of dollars spent on advertising on gun control}$
- $x_3 = \text{number of thousands of dollars spent on advertising on farm subsidies}$
- $x_4 = \text{number of thousands of dollars spent on advertising on gasoline tax}$

Constraints:

- $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$
- $5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$
### Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>-2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>-5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 =$ number of thousands of dollars spent on advertising on building roads
- $x_2 =$ number of thousands of dollars spent on advertising on gun control
- $x_3 =$ number of thousands of dollars spent on advertising on farm subsidies
- $x_4 =$ number of thousands of dollars spent on advertising on gasoline tax

**Constraints:**
- $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$
- $5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$
- $3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 = \text{number of thousands of dollars spent on advertising on building roads}$
- $x_2 = \text{number of thousands of dollars spent on advertising on gun control}$
- $x_3 = \text{number of thousands of dollars spent on advertising on farm subsidies}$
- $x_4 = \text{number of thousands of dollars spent on advertising on gasoline tax}$

**Constraints:**
- $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$
- $5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$
- $3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$

**Objective:** Minimize $x_1 + x_2 + x_3 + x_4$
The Linear Program

---

**Linear Program for the Advertising Problem**

<table>
<thead>
<tr>
<th>Objective</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{minimize} ) ( x_1 + x_2 + x_3 + x_4 )</td>
<td>(-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50)</td>
</tr>
<tr>
<td></td>
<td>(5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100)</td>
</tr>
<tr>
<td></td>
<td>(3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25)</td>
</tr>
<tr>
<td></td>
<td>(x_1, x_2, x_3, x_4 \geq 0)</td>
</tr>
</tbody>
</table>
The Linear Program

Linear Program for the Advertising Problem

minimize \( x_1 + x_2 + x_3 + x_4 \)
subject to

\[
\begin{align*}
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25 \\
x_1, x_2, x_3, x_4 & \geq 0
\end{align*}
\]

The solution of this linear program yields the optimal advertising strategy.
The Linear Program

**Linear Program for the Advertising Problem**

Minimize

\[ x_1 + x_2 + x_3 + x_4 \]

Subject to

\[ -2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \]
\[ 5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \]
\[ 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \]

\[ x_1, x_2, x_3, x_4 \geq 0 \]

The solution of this linear program yields the optimal advertising strategy.

**Formal Definition of Linear Program**
The Linear Program

**Linear Program for the Advertising Problem**

minimize \( x_1 + x_2 + x_3 + x_4 \)

subject to

\[
\begin{align*}
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25 \\
x_1, x_2, x_3, x_4 & \geq 0
\end{align*}
\]

The solution of this linear program yields the optimal advertising strategy.

**Formal Definition of Linear Program**

- Given \( a_1, a_2, \ldots, a_n \) and a set of variables \( x_1, x_2, \ldots, x_n \), a linear function \( f \) is defined by

\[
f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.
\]
The Linear Program

Linear Program for the Advertising Problem

\[
\begin{align*}
\text{minimize} & \quad x_1 + x_2 + x_3 + x_4 \\
\text{subject to} & \\
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25 \\
x_1, x_2, x_3, x_4 & \geq 0
\end{align*}
\]

The solution of this linear program yields the optimal advertising strategy.

Formal Definition of Linear Program

- Given \(a_1, a_2, \ldots, a_n\) and a set of variables \(x_1, x_2, \ldots, x_n\), a linear function \(f\) is defined by
  \[
  f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.
  \]

- **Linear Equality:** \(f(x_1, x_2, \ldots, x_n) = b\)
- **Linear Inequality:** \(f(x_1, x_2, \ldots, x_n) \leq b\)
The Linear Program

Linear Program for the Advertising Problem

Minimize

\[ x_1 + x_2 + x_3 + x_4 \]

Subject to

\[
\begin{align*}
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25 \\
\end{align*}
\]

\[ x_1, x_2, x_3, x_4 \geq 0 \]

The solution of this linear program yields the optimal advertising strategy.

Formal Definition of Linear Program

- Given \( a_1, a_2, \ldots, a_n \) and a set of variables \( x_1, x_2, \ldots, x_n \), a linear function \( f \) is defined by

\[
 f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.
\]

- **Linear Equality:** \( f(x_1, x_2, \ldots, x_n) = b \)
- **Linear Inequality:** \( f(x_1, x_2, \ldots, x_n) \leq b \)
The Linear Program

Linear Program for the Advertising Problem

minimize \( x_1 + x_2 + x_3 + x_4 \)
subject to

\[
\begin{align*}
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25
\end{align*}
\]

\( x_1, x_2, x_3, x_4 \geq 0 \)

The solution of this linear program yields the optimal advertising strategy.

Formal Definition of Linear Program

- Given \( a_1, a_2, \ldots, a_n \) and a set of variables \( x_1, x_2, \ldots, x_n \), a linear function \( f \) is defined by

\[
f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n.
\]

- Linear Equality: \( f(x_1, x_2, \ldots, x_n) = b \)
- Linear Inequality: \( f(x_1, x_2, \ldots, x_n) \geq b \)
- Linear-Programming Problem: either minimize or maximize a linear function subject to a set of linear constraints
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
A Small(er) Example

maximize \( x_1 + x_2 \)

subject to

\[
\begin{align*}
4x_1 - x_2 &\leq 8 \\
2x_1 + x_2 &\leq 10 \\
5x_1 - 2x_2 &\geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.

II. Linear Programming Introduction
A Small(er) Example

maximize $x_1 + x_2$
subject to
$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$

Any setting of $x_1$ and $x_2$ satisfying all constraints is a feasible solution.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.

II. Linear Programming Introduction
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.
A Small(er) Example

maximize $x_1 + x_2$

subject to

$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$

Any setting of $x_1$ and $x_2$ satisfying all constraints is a feasible solution.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.

II. Linear Programming Introduction
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 &\leq 8 \\
2x_1 + x_2 &\leq 10 \\
5x_1 - 2x_2 &\geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]
Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
egin{align*}
4x_1 - x_2 &\leq 8 \\
2x_1 + x_2 &\leq 10 \\
5x_1 - 2x_2 &\geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.
A Small(er) Example

maximize $x_1 + x_2$
subject to
$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$

Any setting of $x_1$ and $x_2$ satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line $x_1 + x_2 = z$ as far up as possible.

II. Linear Programming Introduction
A Small(er) Example

maximize $x_1 + x_2$

subject to

$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$

Any setting of $x_1$ and $x_2$ satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line $x_1 + x_2 = z$ as far up as possible.
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 &\leq 8 \\
2x_1 + x_2 &\leq 10 \\
5x_1 - 2x_2 &\geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.

II. Linear Programming Introduction
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.
maximize $x_1 + x_2$
subject to
$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$

Any setting of $x_1$ and $x_2$ satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line $x_1 + x_2 = z$ as far up as possible.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
A Small(er) Example

maximize \[ x_1 + x_2 \]
subject to

\[
\begin{align*}
4x_1 &- x_2 \leq 8 \\
2x_1 &+ x_2 \leq 10 \\
5x_1 &- 2x_2 \geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.

II. Linear Programming Introduction 8
**A Small(er) Example**

maximize \( x_1 + x_2 \)
subject to

\[
4x_1 - x_2 \leq 8 \\
2x_1 + x_2 \leq 10 \\
5x_1 - 2x_2 \geq -2 \\
x_1, x_2 \geq 0
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

**Graphical Procedure:** Move the line \( x_1 + x_2 = z \) as far up as possible.
A Small(er) Example

maximize \[ x_1 + x_2 \]
subject to
\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
A Small(er) Example

maximize \( x_1 + x_2 \)

subject to

\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.
A Small(er) Example

maximize \( x_1 + x_2 \)

subject to

\[
4x_1 - x_2 \leq 8 \\
2x_1 + x_2 \leq 10 \\
5x_1 - 2x_2 \geq -2 \\
x_1, x_2 \geq 0
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
4x_1 - x_2 \leq 8 \\
2x_1 + x_2 \leq 10 \\
5x_1 - 2x_2 \geq -2 \\
x_1, x_2 \geq 0
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to

\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0 \\
\end{align*}
\]

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.
**A Small(er) Example**

maximize \( x_1 + x_2 \)
subject to

\[
\begin{align*}
4x_1 - x_2 & \leq 8 \\
2x_1 + x_2 & \leq 10 \\
5x_1 - 2x_2 & \geq -2 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.

**Graphical Procedure:** Move the line \( x_1 + x_2 = z \) as far up as possible.
A Small(er) Example

Maximize
\[ x_1 + x_2 \]
Subject to
\[ 4x_1 - x_2 \leq 8 \]
\[ 2x_1 + x_2 \leq 10 \]
\[ 5x_1 - 2x_2 \geq -2 \]
\[ x_1, x_2 \geq 0 \]

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
Introduction

Formulating Problems as Linear Programs

Standard and Slack Forms

Simplex Algorithm

Finding an Initial Solution
Shortest Paths

Single-Pair Shortest Path Problem

- **Given:** directed graph \( G = (V, E) \) with edge weights \( w : E \to \mathbb{R} \), pair of vertices \( s, t \in V \)

II. Linear Programming

Formulating Problems as Linear Programs

10
Shortest Paths

Single-Pair Shortest Path Problem

- **Given**: directed graph \( G = (V, E) \) with edge weights \( w : E \to \mathbb{R} \), pair of vertices \( s, t \in V \)
- **Goal**: Find a path of minimum weight from \( s \) to \( t \) in \( G \)

![Graph Image](image-url)
Shortest Paths

Single-Pair Shortest Path Problem

- **Given**: directed graph \( G = (V, E) \) with edge weights \( w : E \rightarrow \mathbb{R} \), pair of vertices \( s, t \in V \)
- **Goal**: Find a path of minimum weight from \( s \) to \( t \) in \( G \)

\[ p = (v_0 = s, v_1, \ldots, v_k = t) \text{ such that } w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \text{ is minimized.} \]
Shortest Paths

Single-Pair Shortest Path Problem

- **Given**: directed graph \( G = (V, E) \) with edge weights \( w : E \rightarrow \mathbb{R} \), pair of vertices \( s, t \in V \)
- **Goal**: Find a path of minimum weight from \( s \) to \( t \) in \( G \)

\[ p = (v_0 = s, v_1, \ldots, v_k = t) \] such that \[ w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k) \] is minimized.
Shortest Paths

Single-Pair Shortest Path Problem

- **Given**: directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, pair of vertices $s, t \in V$
- **Goal**: Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

Shortest Paths as LP

subject to
Shortest Paths

**Single-Pair Shortest Path Problem**

- **Given**: directed graph \( G = (V, E) \) with edge weights \( w : E \to \mathbb{R} \), pair of vertices \( s, t \in V \)
- **Goal**: Find a path of minimum weight from \( s \) to \( t \) in \( G \)

\[ p = (v_0 = s, v_1, \ldots, v_k = t) \]
\[ w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k) \text{ is minimized.} \]

**Shortest Paths as LP**

\[
\begin{align*}
    d_v & \leq d_u + w(u, v) & \text{for each edge } (u, v) \in E, \\
    d_s & = 0.
\end{align*}
\]
Shortest Paths

Given: directed graph $G = (V, E)$ with edge weights $w : E \rightarrow \mathbb{R}$, pair of vertices $s, t \in V$

Goal: Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

Shortest Paths as LP

Maximize $d_t$

Subject to $d_v \leq d_u + w(u, v)$ for each edge $(u, v) \in E$,

$d_s = 0.$
Shortest Paths

Given: directed graph $G = (V, E)$ with edge weights $w : E \rightarrow \mathbb{R}$, pair of vertices $s, t \in V$

Goal: Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

Shortest Paths as LP

$$\begin{align*}
\text{maximize} & \quad dt \\
\text{subject to} & \quad dv \leq du + w(u, v) \quad \text{for each edge } (u, v) \in E, \\
& \quad ds = 0.
\end{align*}$$

Recall: When Bellman-Ford terminates, all these inequalities are satisfied. Solution $d$ satisfies $dv = \min u : (u, v) \in E \{ du + w(u, v) \}$

This is a maximization problem!
Shortest Paths

Single-Pair Shortest Path Problem

- **Given**: directed graph $G = (V, E)$ with edge weights $w : E \rightarrow \mathbb{R}$, pair of vertices $s, t \in V$
- **Goal**: Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

Shortest Paths as LP

```
maximize
subject to
$\quad d_t$
$\quad d_v \leq d_u + w(u, v)$ for each edge $(u, v) \in E,$
$\quad d_s = 0.$
```

Recall: When BELLMAN-FORD terminates, all these inequalities are satisfied.

This is a maximization problem!
Shortest Paths

Single-Pair Shortest Path Problem

- **Given:** directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, pair of vertices $s, t \in V$
- **Goal:** Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

Recall: When **Bellman-Ford** terminates, all these inequalities are satisfied.

Shortest Paths as LP

maximize $d_t$

subject to

$d_v \leq d_u + w(u, v)$ for each edge $(u, v) \in E,$

$d_s = 0.$

Solution $\bar{d}$ satisfies $\bar{d}_v = \min_{u : (u, v) \in E} \{\bar{d}_u + w(u, v)\}$
Maximum Flow

Given: directed graph \( G = (V, E) \) with edge capacities \( c : E \rightarrow \mathbb{R}^+ \) (recall \( c(u, v) = 0 \) if \((u, v) \not\in E\)), pair of vertices \( s, t \in V \)

Maximum Flow as LP

II. Linear Programming Formulating Problems as Linear Programs 11
Maximum Flow

Given: directed graph \( G = (V, E) \) with edge capacities \( c : E \rightarrow \mathbb{R}^+ \) (recall \( c(u, v) = 0 \) if \( (u, v) \notin E \)), pair of vertices \( s, t \in V \).

Maximum Flow Problem

- Given: directed graph \( G = (V, E) \) with edge capacities \( c : E \rightarrow \mathbb{R}^+ \) (recall \( c(u, v) = 0 \) if \( (u, v) \notin E \)), pair of vertices \( s, t \in V \).

Maximum Flow as LP

II. Linear Programming Formulating Problems as Linear Programs 11
Maximum Flow

Maximum Flow Problem

- **Given:** directed graph $G = (V, E)$ with edge capacities $c : E \rightarrow \mathbb{R}^+$ (recall $c(u, v) = 0$ if $(u, v) \notin E$), pair of vertices $s, t \in V$
- **Goal:** Find a maximum flow $f : V \times V \rightarrow \mathbb{R}$ from $s$ to $t$ which satisfies the capacity constraints and flow conservation

![Graph Diagram]

- The graph shows directed edges with capacities indicated.
- The network represents the flow problem, with nodes $s, t, 2, 3, 4, 5$.
- The flow conservation and capacity constraints are visualized in the diagram.
Maximum Flow

**Maximum Flow Problem**

- **Given:** directed graph $G = (V, E)$ with edge capacities $c : E \rightarrow \mathbb{R}^+$ (recall $c(u, v) = 0$ if $(u, v) \notin E$), pair of vertices $s, t \in V$
- **Goal:** Find a maximum flow $f : V \times V \rightarrow \mathbb{R}$ from $s$ to $t$ which satisfies the capacity constraints and flow conservation

![Diagram of a directed graph with edge capacities and flow values]

$|f| = 19$
Maximum Flow

- **Maximum Flow Problem**
  - **Given:** directed graph $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}^+$ (recall $c(u, v) = 0$ if $(u, v) \notin E$), pair of vertices $s, t \in V$
  - **Goal:** Find a maximum flow $f : V \times V \to \mathbb{R}$ from $s$ to $t$ which satisfies the capacity constraints and flow conservation

```
maximize \[ \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \]
subject to
\[ f_{uv} \leq c(u, v) \] for each $u, v \in V,$
\[ \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \] for each $u \in V \setminus \{s, t\},$
\[ f_{uv} \geq 0 \] for each $u, v \in V.$
```

Maximum Flow as LP
Minimum-Cost Flow

Extension of the Maximum Flow Problem
Minimum-Cost Flow

Extension of the Maximum Flow Problem

Minimum-Cost-Flow Problem

- **Given**: directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{R}^+$, pair of vertices $s, t \in V$, cost function $a : E \rightarrow \mathbb{R}^+$, flow demand of $d$ units.
Minimum-Cost Flow

- **Given:** directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{R}^+$, pair of vertices $s, t \in V$, cost function $a : E \rightarrow \mathbb{R}^+$, flow demand of $d$ units
- **Goal:** Find a flow $f : V \times V \rightarrow \mathbb{R}$ from $s$ to $t$ with $|f| = d$ while minimising the total cost $\sum_{(u, v) \in E} a(u, v)f_{uv}$ incurred by the flow.
Minimum-Cost Flow

**Minimum-Cost-Flow Problem**

- **Given:** directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{R}^+$, pair of vertices $s, t \in V$, cost function $a : E \rightarrow \mathbb{R}^+$, flow demand of $d$ units
- **Goal:** Find a flow $f : V \times V \rightarrow \mathbb{R}$ from $s$ to $t$ with $|f| = d$ while minimising the total cost $\sum_{(u,v) \in E} a(u, v)f_{uv}$ incurred by the flow.

**Figure 29.3** (a) An example of a minimum-cost-flow problem. We denote the capacities by $c$ and the costs by $a$. Vertex $s$ is the source and vertex $t$ is the sink, and we wish to send 4 units of flow from $s$ to $t$. (b) A solution to the minimum-cost flow problem in which 4 units of flow are sent from $s$ to $t$. For each edge, the flow and capacity are written as flow/capacity.
Minimum-Cost Flow

Given: directed graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}^+$, pair of vertices $s, t \in V$, cost function $a : E \to \mathbb{R}^+$, flow demand of $d$ units

Goal: Find a flow $f : V \times V \to \mathbb{R}$ from $s$ to $t$ with $|f| = d$ while minimising the total cost $\sum_{(u, v) \in E} a(u, v)f_{uv}$ incurred by the flow.

Optimal Solution with total cost:
$\sum_{(u, v) \in E} a(u, v)f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$

Figure 29.3 (a) An example of a minimum-cost-flow problem. We denote the capacities by $c$ and the costs by $a$. Vertex $s$ is the source and vertex $t$ is the sink, and we wish to send 4 units of flow from $s$ to $t$. (b) A solution to the minimum-cost flow problem in which 4 units of flow are sent from $s$ to $t$. For each edge, the flow and capacity are written as flow/capacity.
Minimum-Cost Flow as a LP

\[ \text{minimize } \sum_{(u,v) \in E} a(u,v) f_{uv} \]
\[ \text{subject to } \]
\[ f_{uv} \leq c(u,v) \quad \text{for each } u, v \in V, \]
\[ \sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} = 0 \quad \text{for each } u \in V \setminus \{s, t\}, \]
\[ \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} = d, \]
\[ f_{uv} \geq 0 \quad \text{for each } u, v \in V. \]
Minimum-Cost Flow as a LP

minimize \( \sum_{(u,v) \in E} a(u, v) f_{uv} \)
subject to
\[ f_{uv} \leq c(u, v) \text{ for each } u, v \in V, \]
\[ \sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} = 0 \text{ for each } u \in V \setminus \{s, t\}, \]
\[ \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} = d, \]
\[ f_{uv} \geq 0 \text{ for each } u, v \in V. \]

Real power of Linear Programming comes from the ability to solve new problems!
Outline

Introduction

Formulating Problems as Linear Programs

Standard and Slack Forms

Simplex Algorithm

Finding an Initial Solution
Standard and Slack Forms

Standard Form

maximize \[ \sum_{j=1}^{n} c_j x_j \]
subject to \[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for} \; i = 1, 2, \ldots, m \]
\[ x_j \geq 0 \quad \text{for} \; j = 1, 2, \ldots, n \]
Standard and Slack Forms

Standard Form

maximize \[ \sum_{j=1}^{n} c_j x_j \]
subject to \[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \]
\[ x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n \]
Standard and Slack Forms

**Standard Form**

\[
\text{maximize } \sum_{j=1}^{n} c_j x_j \\
\text{subject to } \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m \\
x_j \geq 0 \text{ for } j = 1, 2, \ldots, n
\]

- Objective Function
- \( n + m \) Constraints
Standard and Slack Forms

\[ \text{Standard Form} \]

\[
\begin{align*}
\text{maximize} & \quad \sum_{j=1}^{n} c_j x_j & \text{Objective Function} \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i & \text{for } i = 1, 2, \ldots, m \\
& \quad x_j \geq 0 & \text{for } j = 1, 2, \ldots, n
\end{align*}
\]

\[ n + m \text{ Constraints} \]
Standard and Slack Forms

Standard Form

\[
\begin{align*}
\text{maximize} & \quad \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \\
& \quad x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n
\end{align*}
\]

\(n + m\) Constraints

Standard Form (Matrix-Vector-Notation)

\[
\begin{align*}
\text{maximize} & \quad c^T x \\
\text{subject to} & \quad Ax \leq b \\
& \quad x \geq 0
\end{align*}
\]

Inner product of two vectors

Matrix-vector product

II. Linear Programming
Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.
2. There might be variables without nonnegativity constraints.
3. There might be equality constraints.
4. There might be inequality constraints (with $\geq$ instead of $\leq$).
Converting Linear Programs into Standard Form

Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.
2. There might be variables without nonnegativity constraints.
3. There might be equality constraints.
4. There might be inequality constraints (with $\geq$ instead of $\leq$).

**Goal:** Convert linear program into an equivalent program which is in standard form
Converting Linear Programs into Standard Form

Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.
2. There might be variables without nonnegativity constraints.
3. There might be equality constraints.
4. There might be inequality constraints (with $\geq$ instead of $\leq$).

Goal: Convert linear program into an equivalent program which is in standard form

Equivalence: a correspondence (not necessarily a bijection) between solutions.
Converting into Standard Form (1/5)

Reasons for a LP not being in standard form:

1. The objective might be a minimization rather than maximization.
Converting into Standard Form (1/5)

Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.

\[
\begin{align*}
\text{minimize} & \quad -2x_1 + 3x_2 \\
\text{subject to} & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \leq 4 \\
& x_1 \geq 0
\end{align*}
\]
Reasons for a LP not being in standard form:
1. The objective might be a **minimization** rather than **maximization**.

Minimize: \(-2x_1 + 3x_2\)

Subject to:

\[
\begin{align*}
x_1 + x_2 &= 7 \\
x_1 - 2x_2 &
\leq 4 \\
x_1 &\geq 0
\end{align*}
\]

Negate objective function
Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.

\[
\begin{align*}
\text{minimize} & \quad -2x_1 + 3x_2 \\
\text{subject to} & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \leq 4 \\
& x_1 \geq 0
\end{align*}
\]

Negate objective function

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x_2 \\
\text{subject to} & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \leq 4 \\
& x_1 \geq 0
\end{align*}
\]
Converting into Standard Form (2/5)

Reasons for a LP not being in standard form:
2. There might be variables without nonnegativity constraints.

Replace $x_2$ by two non-negative variables $x'_2$ and $x''_2$. 
Reasons for a LP not being in standard form:

2. There might be variables without nonnegativity constraints.

maximize \[ 2x_1 - 3x_2 \]
subject to
\[ x_1 + x_2 = 7 \]
\[ x_1 - 2x_2 < 4 \]
\[ x_1 \geq 0 \]
Reasons for a LP not being in standard form:

2. There might be variables without nonnegativity constraints.

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x_2 \\
\text{subject to} & \quad x_1 + x_2 = 7 \\
& \quad x_1 - 2x_2 \leq 4 \\
& \quad x_1 \geq 0
\end{align*}
\]

Replace \( x_2 \) by two non-negative variables \( x'_2 \) and \( x''_2 \).
Converting into Standard Form (2/5)

Reasons for a LP not being in standard form:
2. There might be variables without nonnegativity constraints.

\[
\text{maximize } 2x_1 - 3x_2 \\
\text{subject to } \\
x_1 + x_2 = 7 \\
x_1 - 2x_2 < 4 \\
x_1 \geq 0
\]

Replace \(x_2\) by two non-negative variables \(x_2'\) and \(x_2''\)

\[
\text{maximize } 2x_1 - 3x_2' + 3x_2'' \\
\text{subject to } \\
x_1 + x_2' - x_2'' = 7 \\
x_1 - 2x_2' + 2x_2'' < 4 \\
x_1, x_2', x_2'' \geq 0
\]
Reasons for a LP not being in standard form:

3. There might be equality constraints.
Reasons for a LP not being in standard form:
3. There might be equality constraints.

maximize \( 2x_1 - 3x_2' + 3x_2'' \)
subject to
\[
\begin{align*}
  x_1 + x_2' - x_2'' &= 7 \\
  x_1 - 2x_2' + 2x_2'' &\leq 4 \\
  x_1, x_2', x_2'' &\geq 0
\end{align*}
\]
Reasons for a LP not being in standard form:

3. There might be equality constraints.

maximize $2x_1 - 3x_2' + 3x_2''$

subject to

$x_1 + x_2' - x_2'' = 7$

$x_1 - 2x_2' + 2x_2'' \leq 4$

$x_1, x_2', x_2'' \geq 0$

Replace each equality by two inequalities.
Reasons for a LP not being in standard form:

3. There might be equality constraints.

maximize \[ 2x_1 - 3x_2' + 3x_2'' \]
subject to
\[ x_1 + x_2' - x_2'' = 7 \]
\[ x_1 - 2x_2' + 2x_2'' \leq 4 \]
\[ x_1, x_2', x_2'' \geq 0 \]
Replace each equality by two inequalities.

maximize \[ 2x_1 - 3x_2' + 3x_2'' \]
subject to
\[ x_1 + x_2' - x_2'' \leq 7 \]
\[ x_1 + x_2' - x_2'' \geq 7 \]
\[ x_1 - 2x_2' + 2x_2'' \leq 4 \]
\[ x_1, x_2', x_2'' \geq 0 \]
Reasons for a LP not being in standard form:

4. There might be inequality constraints (with $\geq$ instead of $\leq$).
Converting into Standard Form (4/5)

Reasons for a LP not being in standard form:
4. There might be inequality constraints (with $\geq$ instead of $\leq$).

maximize $2x_1 - 3x'_2 + 3x''_2$
subject to
$x_1 + x'_2 - x''_2 \leq 7$
$x_1 + x'_2 - x''_2 \geq 7$
$x_1 - 2x'_2 + 2x''_2 \leq 4$
$x_1, x'_2, x''_2 \geq 0$

Negate respective inequalities.
Reasons for a LP not being in standard form:

4. There might be inequality constraints (with $\geq$ instead of $\leq$).

Maximize

$$2x_1 - 3x'_2 + 3x''_2$$

Subject to

\[
\begin{align*}
x_1 + x'_2 - x''_2 & \leq 7 \\
x_1 + x'_2 - x''_2 & \geq 7 \\
x_1 - 2x'_2 + 2x''_2 & \leq 4 \\
x_1, x'_2, x''_2 & \geq 0
\end{align*}
\]

Negate respective inequalities.
Reasons for a LP not being in standard form:

4. There might be inequality constraints (with ≥ instead of ≤).

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x_2' + 3x_2'' \\
\text{subject to} & \quad x_1 + x_2' - x_2'' \leq 7 \\
& \quad x_1 + x_2' - x_2'' \geq 7 \\
& \quad x_1 - 2x_2' + 2x_2'' \leq 4 \\
& \quad x_1, x_2', x_2'' \geq 0
\end{align*}
\]

Negate respective inequalities.

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x_2' + 3x_2'' \\
\text{subject to} & \quad -x_1 + x_2' - x_2'' \leq 7 \\
& \quad -x_1 - x_2' + x_2'' \leq -7 \\
& \quad x_1 - 2x_2' + 2x_2'' \leq 4 \\
& \quad x_1, x_2', x_2'' \geq 0
\end{align*}
\]
maximize $2x_1 - 3x_2 + 3x_3$

subject to

$x_1 + x_2 - x_3 \leq 7$
$-x_1 - x_2 + x_3 \leq -7$
$x_1 - 2x_2 + 2x_3 \leq 4$

$x_1, x_2, x_3 \geq 0$
Converting into Standard Form (5/5)

maximize $2x_1 - 3x_2 + 3x_3$

subject to

$$x_1 + x_2 - x_3 \leq 7$$

$$-x_1 - x_2 + x_3 \leq -7$$

$$x_1 - 2x_2 + 2x_3 \leq 4$$

$x_1, x_2, x_3 \geq 0$

Rename variable names (for consistency).
Converting into Standard Form (5/5)

It is always possible to convert a linear program into standard form.
Goal: Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.
Goal: Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.
Goal: Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

Introducing Slack Variables
**Goal:** Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

---

Introducing Slack Variables

- Let $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ be an inequality constraint
**Goal:** Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

---

Introducing Slack Variables

- Let \( \sum_{j=1}^{n} a_{ij}x_j \leq b_i \) be an inequality constraint
- Introduce a slack variable \( s \) by
**Goal:** Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

Introducing Slack Variables

- Let $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ be an inequality constraint
- Introduce a slack variable $s$ by

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j$$
**Goal:** Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

---

Introducing Slack Variables

- Let $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ be an inequality constraint
- Introduce a slack variable $s$ by

\[
s = b_i - \sum_{j=1}^{n} a_{ij}x_j
\]

$s \geq 0$. 

---

II. Linear Programming Standard and Slack Forms 22
Goal: Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

Introducing Slack Variables

- Let \( \sum_{j=1}^{n} a_{ij}x_j \leq b_i \) be an inequality constraint
- Introduce a slack variable \( s \) by

\[
s = b_i - \sum_{j=1}^{n} a_{ij}x_j
\]

\( s \geq 0. \)

\( s \) measures the slack between the two sides of the inequality.
Goal: Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

Introducing Slack Variables

- Let \( \sum_{j=1}^{n} a_{ij}x_j \leq b_i \) be an inequality constraint
- Introduce a slack variable \( s \) by

\[
s = b_i - \sum_{j=1}^{n} a_{ij}x_j
\]

\( s \geq 0 \).

- Denote slack variable of the \( i \)th inequality by \( x_{n+i} \)
maximize $2x_1 - 3x_2 + 3x_3$
subject to

\[
\begin{align*}
    x_1 + x_2 - x_3 &\leq 7 \\
    -x_1 - x_2 + x_3 &\leq -7 \\
    x_1 - 2x_2 + 2x_3 &\leq 4 \\
    x_1, x_2, x_3 &\geq 0
\end{align*}
\]
maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[ x_1 + x_2 - x_3 \leq 7 \]
\[ -x_1 - x_2 + x_3 \leq -7 \]
\[ x_1 - 2x_2 + 2x_3 \leq 4 \]
\[ x_1, x_2, x_3 \geq 0 \]

Introduce slack variables
maximize \quad 2x_1 - 3x_2 + 3x_3

subject to
\begin{align*}
x_1 + x_2 - x_3 & \leq 7 \\
-x_1 - x_2 + x_3 & \leq -7 \\
x_1 - 2x_2 + 2x_3 & \leq 4 \\
x_1, x_2, x_3 & \geq 0
\end{align*}

Introduce slack variables

subject to
\begin{align*}
x_4 & = 7 - x_1 - x_2 + x_3
\end{align*}
Converting Standard Form into Slack Form (2/3)

maximize  
subject to

\[ 2x_1 - 3x_2 + 3x_3 \]
\[ x_1 + x_2 - x_3 \leq 7 \]
\[ -x_1 - x_2 + x_3 \leq -7 \]
\[ x_1 - 2x_2 + 2x_3 \leq 4 \]
\[ x_1, x_2, x_3 \geq 0 \]

Introduce slack variables

subject to

\[ x_4 = 7 - x_1 - x_2 + x_3 \]
\[ x_5 = -7 + x_1 + x_2 - x_3 \]
Converting Standard Form into Slack Form (2/3)

maximize \( 2x_1 - 3x_2 + 3x_3 \)
subject to
\[
\begin{align*}
  x_1 + x_2 - x_3 & \leq 7 \\
  -x_1 - x_2 + x_3 & \leq -7 \\
  x_1 - 2x_2 + 2x_3 & \leq 4 \\
  x_1, x_2, x_3 & \geq 0
\end{align*}
\]

Introduce slack variables

subject to
\[
\begin{align*}
  x_4 & = 7 - x_1 - x_2 + x_3 \\
  x_5 & = -7 + x_1 + x_2 - x_3 \\
  x_6 & = 4 - x_1 + 2x_2 - 2x_3
\end{align*}
\]
Converting Standard Form into Slack Form (2/3)

maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[
\begin{align*}
    x_1 &+ x_2 - x_3 &\leq& 7 \\
    -x_1 &- x_2 &+ x_3 &\leq -7 \\
    x_1 &- 2x_2 &+ 2x_3 &\leq 4 \\
\end{align*}
\]
\[ x_1, x_2, x_3 \geq 0 \]

Introduce slack variables

subject to
\[
\begin{align*}
    x_4 &= 7 - x_1 - x_2 + x_3 \\
    x_5 &= -7 + x_1 + x_2 - x_3 \\
    x_6 &= 4 - x_1 + 2x_2 - 2x_3 \\
\end{align*}
\]
\[ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \]
Converting Standard Form into Slack Form (2/3)

maximize 2\(x_1\) − 3\(x_2\) + 3\(x_3\)
subject to
\[
\begin{align*}
-x_1 & + x_2 & - x_3 & \leq 7 \\
-x_1 & - x_2 & + x_3 & \leq -7 \\
2x_2 & - 2x_3 & \leq 4 \\
\end{align*}
\]
\(x_1, x_2, x_3\) \(\geq 0\)

Introduce slack variables

maximize 2\(x_1\) − 3\(x_2\) + 3\(x_3\)
subject to
\[
\begin{align*}
x_4 & = 7 - x_1 - x_2 + x_3 \\
x_5 & = -7 + x_1 + x_2 - x_3 \\
x_6 & = 4 - x_1 + 2x_2 - 2x_3 \\
\end{align*}
\]
\(x_1, x_2, x_3, x_4, x_5, x_6\) \(\geq 0\)
maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[ x_4 = 7 - x_1 - x_2 + x_3 \]
\[ x_5 = -7 + x_1 + x_2 - x_3 \]
\[ x_6 = 4 - x_1 + 2x_2 - 2x_3 \]
\[ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \]

This is called slack form.
maximize 2 \cdot x_1 - 3 \cdot x_2 + 3 \cdot x_3 \\
subject to \\
x_4 = 7 - x_1 - x_2 + x_3 \\
x_5 = -7 + x_1 + x_2 - x_3 \\
x_6 = 4 - x_1 + 2 \cdot x_2 - 2 \cdot x_3 \\
\quad x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \\
Use variable z to denote objective function \\
and omit the nonnegativity constraints.

This is called slack form.
Converting Standard Form into Slack Form (3/3)

maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[
\begin{align*}
  x_4 & = 7 - x_1 - x_2 + x_3 \\
  x_5 & = -7 + x_1 + x_2 - x_3 \\
  x_6 & = 4 - x_1 + 2x_2 - 2x_3
\end{align*}
\]
\[ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \]

Use variable \( z \) to denote objective function and omit the nonnegativity constraints.

<table>
<thead>
<tr>
<th>( z )</th>
<th>( 2x_1 )</th>
<th>( -3x_2 )</th>
<th>( +3x_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_4 )</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( x_5 )</td>
<td>-7</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>( x_6 )</td>
<td>4</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>
Converting Standard Form into Slack Form (3/3)

maximize \(2x_1 - 3x_2 + 3x_3\)

subject to

\[
egin{align*}
x_4 &= 7 - x_1 - x_2 + x_3 \\
x_5 &= -7 + x_1 + x_2 - x_3 \\
x_6 &= 4 - x_1 + 2x_2 - 2x_3
\end{align*}
\]

\(x_1, x_2, x_3, x_4, x_5, x_6 \geq 0\)

Use variable \(z\) to denote objective function and omit the nonnegativity constraints.

This is called slack form.
Basic and Non-Basic Variables

\[ z = 2x_1 - 3x_2 + 3x_3 \]

\[ x_4 = 7 - x_1 - x_2 + x_3 \]

\[ x_5 = -7 + x_1 + x_2 - x_3 \]

\[ x_6 = 4 - x_1 + 2x_2 - 2x_3 \]
Basic and Non-Basic Variables

\[
\begin{align*}
  z & = & 2x_1 & - & 3x_2 & + & 3x_3 \\
  x_4 & = & 7 & - & x_1 & - & x_2 & + & x_3 \\
  x_5 & = & -7 & + & x_1 & + & x_2 & - & x_3 \\
  x_6 & = & 4 & - & x_1 & + & 2x_2 & - & 2x_3
\end{align*}
\]

**Basic Variables:** \( B = \{4, 5, 6\} \)
Basic and Non-Basic Variables

\[ z = 2x_1 - 3x_2 + 3x_3 \]
\[ x_4 = 7 - x_1 - x_2 + x_3 \]
\[ x_5 = -7 + x_1 + x_2 - x_3 \]
\[ x_6 = 4 - x_1 + 2x_2 - 2x_3 \]

Basic Variables: \( B = \{4, 5, 6\} \)

Non-Basic Variables: \( N = \{1, 2, 3\} \)
Basic and Non-Basic Variables

\[ z = 2x_1 - 3x_2 + 3x_3 \]
\[ x_4 = \begin{array}{c} 7 \\ -7 \\ 4 \end{array} - \begin{array}{c} x_1 \\ x_1 \\ x_1 \end{array} - \begin{array}{c} x_2 \\ x_2 \\ 2x_2 \end{array} + \begin{array}{c} x_3 \\ x_3 \end{array} - \begin{array}{c} 2x_3 \end{array} \]

Basic Variables: \( B = \{4, 5, 6\} \)
Non-Basic Variables: \( N = \{1, 2, 3\} \)

Slack Form (Formal Definition)

Slack form is given by a tuple \((N, B, A, b, c, v)\) so that

\[ z = v + \sum_{j \in N} c_j x_j \]
\[ x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B, \]

and all variables are non-negative.
Basic and Non-Basic Variables

\[
\begin{align*}
  z &= 2x_1 - 3x_2 + 3x_3 \\
  x_4 &= 7 - x_1 - x_2 + x_3 \\
  x_5 &= -7 + x_1 + x_2 - x_3 \\
  x_6 &= 4 - x_1 + 2x_2 - 2x_3
\end{align*}
\]

Basic Variables: \(B = \{4, 5, 6\}\)

Non-Basic Variables: \(N = \{1, 2, 3\}\)

Slack Form (Formal Definition)

Slack form is given by a tuple \((N, B, A, b, c, v)\) so that

\[
\begin{align*}
  z &= v + \sum_{j \in N} c_j x_j \\
  x_i &= b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B
\end{align*}
\]

and all variables are non-negative.

Variables/Coefficients on the right hand side are indexed by \(B\) and \(N\).
Slack Form (Example)

\[
\begin{align*}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{align*}
\]
Slack Form (Example)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]
Slack Form (Example)

\[
\begin{align*}
    z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
    x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
    x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
    x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{align*}
\]

Slack Form Notation

- \[B = \{1, 2, 4\}, \ N = \{3, 5, 6\}\]
Slack Form (Example)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Slack Form Notation

- \( B = \{1, 2, 4\}, \ N = \{3, 5, 6\} \)
- 
  \[
  A = \begin{pmatrix}
    a_{13} & a_{15} & a_{16} \\
    a_{23} & a_{25} & a_{26} \\
    a_{43} & a_{45} & a_{46}
  \end{pmatrix}
  = \begin{pmatrix}
    -1/6 & -1/6 & 1/3 \\
    8/3 & 2/3 & -1/3 \\
    1/2 & -1/2 & 0
  \end{pmatrix}
\]
Slack Form (Example)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Slack Form Notation

- \( B = \{1, 2, 4\}, \; N = \{3, 5, 6\} \)
- \( A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix} \)
- \( b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} \)
Slack Form (Example)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]

\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]

\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]

\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Slack Form Notation

- \( B = \{1, 2, 4\}, \ N = \{3, 5, 6\} \)
- \[
A = \begin{pmatrix}
a_{13} & a_{15} & a_{16} \\
a_{23} & a_{25} & a_{26} \\
a_{43} & a_{45} & a_{46}
\end{pmatrix} = \begin{pmatrix}
-1/6 & -1/6 & 1/3 \\
8/3 & 2/3 & -1/3 \\
1/2 & -1/2 & 0
\end{pmatrix}
\]
- \[
b = \begin{pmatrix}
b_1 \\
b_2 \\
b_4
\end{pmatrix} = \begin{pmatrix}
8 \\
4 \\
18
\end{pmatrix}, \quad c = \begin{pmatrix}
c_3 \\
c_5 \\
c_6
\end{pmatrix} = \begin{pmatrix}
-1/6 \\
-1/6 \\
-2/3
\end{pmatrix}
\]
Slack Form (Example)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Slack Form Notation

- \( B = \{1, 2, 4\}, \ N = \{3, 5, 6\} \)
- \[ A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -\frac{1}{6} & -\frac{1}{6} & \frac{1}{3} \\ \frac{8}{3} & \frac{2}{3} & -\frac{1}{3} \\ \frac{1}{2} & -\frac{1}{2} & 0 \end{pmatrix} \]
- \[ b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix}, \ c = \begin{pmatrix} c_3 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} -\frac{1}{6} \\ -\frac{1}{6} \\ -\frac{2}{3} \end{pmatrix} \]
- \( v = 28 \)
A point \( x \) is a \textbf{vertex} if it cannot be represented as a strict convex combination of two other points in the feasible set.
The Structure of Optimal Solutions

Definition

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.
The Structure of Optimal Solutions

**Definition**
A point $x$ is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

**Theorem**
If the slack form has an optimal solution, one of them occurs at a vertex.
The Structure of Optimal Solutions

Definition
A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):
- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex

\[ x - d \text{ and } x + d \text{ are feasible} \]

Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$

W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)

Consider $x + \lambda d$ as a function of $\lambda \geq 0$
The Structure of Optimal Solutions

**Definition**

A point \( x \) is a *vertex* if it cannot be represented as a strict convex combination of two other points in the feasible set.

**Theorem**

If the slack form has an optimal solution, one of them occurs at a vertex.

**Proof Sketch (informal and non-examinable):**

- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex

\[ \begin{align*}
\text{Consider } x + \lambda d \text{ as a function of } \lambda \geq 0
\end{align*} \]
The Structure of Optimal Solutions

**Definition**
A point $x$ is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

**Theorem**
If the slack form has an optimal solution, **one of them** occurs at a vertex.

**Proof Sketch** (informal and non-examinable):
- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex
  $\implies \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible

![Graph showing the structure of optimal solutions with a shaded region and a vector $d$ forming a line segment from $x$](image)
The Structure of Optimal Solutions

Definition

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem

If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex
  $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
The Structure of Optimal Solutions

Definition
A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):
- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
The Structure of Optimal Solutions

**Definition**

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

**Theorem**

If the slack form has an optimal solution, one of them occurs at a vertex.

**Proof Sketch (informal and non-examinable):**

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
The Structure of Optimal Solutions

Definition
A point \( x \) is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):
- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex \( \Rightarrow \exists \) vector \( d \) s.t. \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \) \( \Rightarrow \) \( Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \( -d \))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)
The Structure of Optimal Solutions

Definition
A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):
- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$
- Case 1: There exists $j$ with $d_j < 0$
The Structure of Optimal Solutions

**Definition**

A point \( x \) is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

**Theorem**

If the slack form has an optimal solution, **one of them** occurs at a vertex.

**Proof Sketch (informal and non-examinable):**

- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex \( \Rightarrow \exists \) vector \( d \) s.t. \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \Rightarrow Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \(-d\))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)
- **Case 1:** There exists \( j \) with \( d_j < 0 \)
  - Increase \( \lambda \) from 0 to \( \lambda' \) until a new entry of \( x + \lambda d \) becomes zero
The Structure of Optimal Solutions

**Definition**

A point \( x \) is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

**Theorem**

If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex
  \( \Rightarrow \exists \) vector \( d \) s.t. \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \) \( \Rightarrow Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \( -d \))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)

- **Case 1:** There exists \( j \) with \( d_j < 0 \)
  - Increase \( \lambda \) from 0 to \( \lambda' \) until a new entry of \( x + \lambda d \) becomes zero
  - \( x + \lambda' d \) feasible, since \( A(x + \lambda' d) = Ax = b \) and \( x + \lambda' d \geq 0 \)
The Structure of Optimal Solutions

Definition
A point \( x \) is a \textbf{vertex} if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, \textbf{one of them} occurs at a vertex.

Proof Sketch (informal and non-examinable):
- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex ⇒ \( \exists \) vector \( d \) s.t. \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \) ⇒ \( Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \( -d \))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)

Case 1: There exists \( j \) with \( d_j < 0 \)
- Increase \( \lambda \) from 0 to \( \lambda' \) until a new entry of \( x + \lambda d \) becomes zero
- \( x + \lambda' d \) feasible, since \( A(x + \lambda' d) = Ax = b \) and \( x + \lambda' d \geq 0 \)
- \( c^T (x + \lambda' d) = c^T x + c^T \lambda' d \geq c^T x \)
The Structure of Optimal Solutions

**Definition**

A point $x$ is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

**Theorem**

If the slack form has an optimal solution, **one of them** occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex
  $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$
- **Case 2**: For all $j$, $d_j \geq 0$
The Structure of Optimal Solutions

Definition
A point \( x \) is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem
If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. \( Ax = b \). Let \( x \) be optimal but not a vertex
  \( \Rightarrow \exists \) vector \( d \) s.t. \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \) \( \Rightarrow \) \( Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \( -d \))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)

- Case 2: For all \( j \), \( d_j \geq 0 \)
  - \( x + \lambda d \) is feasible for all \( \lambda \geq 0 \): \( A(x + \lambda d) = b \) and \( x + \lambda d \geq x \geq 0 \)
The Structure of Optimal Solutions

Definition

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem

If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex
  $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$

  **Case 2:** For all $j$, $d_j \geq 0$
  - $x + \lambda d$ is feasible for all $\lambda \geq 0$: $A(x + \lambda d) = b$ and $x + \lambda d \geq x \geq 0$
  - If $\lambda \to \infty$, then $c^T (x + \lambda d) \to \infty$
The Structure of Optimal Solutions

Definition

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem

If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex $\implies \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \implies Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$

Case 2: For all $j$, $d_j \geq 0$

- $x + \lambda d$ is feasible for all $\lambda \geq 0$: $A(x + \lambda d) = b$ and $x + \lambda d \geq x \geq 0$
- If $\lambda \to \infty$, then $c^T (x + \lambda d) \to \infty$
- $\implies$ This contradicts the assumption that there exists an optimal solution.
The Structure of Optimal Solutions

Definition

A point $x$ is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem

If the slack form has an optimal solution, one of them occurs at a vertex.

Proof Sketch (informal and non-examinable):

- Rewrite LP s.t. $Ax = b$. Let $x$ be optimal but not a vertex
  $\Rightarrow \exists$ vector $d$ s.t. $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$

  - Case 2: For all $j$, $d_j \geq 0$
    - $x + \lambda d$ is feasible for all $\lambda \geq 0$: $A(x + \lambda d) = b$ and $x + \lambda d \geq x \geq 0$
    - If $\lambda \to \infty$, then $c^T (x + \lambda d) \to \infty$
  $\Rightarrow$ This contradicts the assumption that there exists an optimal solution.

II. Linear Programming

Standard and Slack Forms
Outline

Introduction

Formulating Problems as Linear Programs

Standard and Slack Forms

Simplex Algorithm

Finding an Initial Solution
Simplex Algorithm: Introduction

- classical method for solving linear programs (Dantzig, 1947)
- usually fast in practice although worst-case runtime not polynomial
- iterative procedure somewhat similar to Gaussian elimination
Simplex Algorithm: Introduction

- classical method for solving linear programs (Dantzig, 1947)
- usually fast in practice although worst-case runtime not polynomial
- iterative procedure somewhat similar to Gaussian elimination

### Basic Idea:

- Each iteration corresponds to a “basic solution” of the slack form
- All non-basic variables are 0, and the basic variables are determined from the equality constraints
- Each iteration converts one slack form into an equivalent one while the objective value will not decrease
- Conversion (“pivoting”) is achieved by switching the roles of one basic and one non-basic variable
Simplex Algorithm: Introduction

- classical method for solving linear programs (Dantzig, 1947)
- usually fast in practice although worst-case runtime not polynomial
- iterative procedure somewhat similar to Gaussian elimination

**Basic Idea:**
- Each iteration corresponds to a “basic solution” of the slack form
- All non-basic variables are 0, and the basic variables are determined from the equality constraints
- Each iteration converts one slack form into an equivalent one while the objective value will not decrease
- Conversion (“pivoting”) is achieved by switching the roles of one basic and one non-basic variable

In that sense, it is a **greedy algorithm**.
Extended Example: Conversion into Slack Form

maximize \[ 3x_1 + x_2 + 2x_3 \]
subject to
\[ x_1 + x_2 + 3x_3 \leq 30 \]
\[ 2x_1 + 2x_2 + 5x_3 \leq 24 \]
\[ 4x_1 + x_2 + 2x_3 \leq 36 \]
\[ x_1, x_2, x_3 \geq 0 \]
Extended Example: Conversion into Slack Form

maximize \( 3x_1 + x_2 + 2x_3 \)

subject to

\[
\begin{align*}
x_1 + x_2 + 3x_3 & \leq 30 \\
2x_1 + 2x_2 + 5x_3 & \leq 24 \\
4x_1 + x_2 + 2x_3 & \leq 36 \\
x_1, x_2, x_3 & \geq 0
\end{align*}
\]

Conversion into slack form
Extended Example: Conversion into Slack Form

maximize \[ 3x_1 + x_2 + 2x_3 \]
subject to
\[ x_1 + x_2 + 3x_3 \leq 30 \]
\[ 2x_1 + 2x_2 + 5x_3 \leq 24 \]
\[ 4x_1 + x_2 + 2x_3 \leq 36 \]
\[ x_1, x_2, x_3 \geq 0 \]

Conversion into slack form

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]
Extended Example: Iteration 1

\[ z = 3x_1 + x_2 + 2x_3 \]

\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]

\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]

\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]
Extended Example: Iteration 1

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Basic solution: \((x_1, x_2, \ldots, x_6) = (0, 0, 0, 30, 24, 36)\)
Extended Example: Iteration 1

\[
\begin{align*}
    z &= 3x_1 + x_2 + 2x_3 \\
    x_4 &= 30 - x_1 - x_2 - 3x_3 \\
    x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
    x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]

Basic solution: \((x_1, x_2, \ldots, x_6) = (0, 0, 0, 30, 24, 36)\)

This basic solution is \textbf{feasible}
Extended Example: Iteration 1

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Basic solution: \((x_1, x_2, \ldots, x_6) = (0, 0, 0, 30, 24, 36)\)

This basic solution is **feasible**

Objective value is 0.
Extended Example: Iteration 1

Increasing the value of $x_1$ would increase the objective value.

\[
\begin{align*}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]

Basic solution: \((x_1, x_2, \ldots, x_6) = (0, 0, 0, 30, 24, 36)\)

This basic solution is **feasible**

Objective value is 0.
Increasing the value of $x_1$ would increase the objective value.

\[
\begin{align*}
    z &= 3x_1 + x_2 + 2x_3 \\
    x_4 &= 30 - x_1 - x_2 - 3x_3 \\
    x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
    x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]

The third constraint is the tightest and limits how much we can increase $x_1$. 
Extended Example: Iteration 1

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Increasing the value of \( x_1 \) would increase the objective value.

The third constraint is the tightest and limits how much we can increase \( x_1 \).

Switch roles of \( x_1 \) and \( x_6 \):
Extended Example: Iteration 1

Increasing the value of $x_1$ would increase the objective value.

\[ z = 3x_1 + x_2 + 2x_3 \]

\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]

\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]

\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

The third constraint is the tightest and limits how much we can increase $x_1$.

Switch roles of $x_1$ and $x_6$:
- Solving for $x_1$ yields:

\[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \]
Extended Example: Iteration 1

Increasing the value of $x_1$ would increase the objective value.

\[ z = 3x_1 + x_2 + 2x_3 \]

\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]

\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]

\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

The third constraint is the tightest and limits how much we can increase $x_1$.

Switch roles of $x_1$ and $x_6$:
- Solving for $x_1$ yields:
  \[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \]
- Substitute this into $x_1$ in the other three equations
Extended Example: Iteration 2

\[ z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \]

\[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \]

\[ x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \]

\[ x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \]
Extended Example: Iteration 2

\[ z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \]
\[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \]
\[ x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \]
\[ x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \]

Basic solution: \((x_1, x_2, \ldots, x_6) = (9, 0, 0, 21, 6, 0)\) with objective value 27
Extended Example: Iteration 2

Increasing the value of \( x_3 \) would increase the objective value.

\[
\begin{align*}
    z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
    x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
    x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
    x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{align*}
\]

Basic solution: \((x_1, x_2, \ldots, x_6) = (9, 0, 0, 21, 6, 0)\) with objective value 27
Extended Example: Iteration 2

Increasing the value of $x_3$ would increase the objective value.

\[ z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \]

\[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \]

\[ x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \]

\[ x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \]

The third constraint is the tightest and limits how much we can increase $x_3$. 

II. Linear Programming Simplex Algorithm
Extended Example: Iteration 2

Increasing the value of \( x_3 \) would increase the objective value.

\[
\begin{align*}
    z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
    x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
    x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
    x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{align*}
\]

The third constraint is the tightest and limits how much we can increase \( x_3 \).

Switch roles of \( x_3 \) and \( x_5 \):
Extended Example: Iteration 2

Increasing the value of $x_3$ would increase the objective value.

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}$$

The third constraint is the tightest and limits how much we can increase $x_3$.

Switch roles of $x_3$ and $x_5$:

- Solving for $x_3$ yields:
  $$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} - \frac{x_6}{8}.$$
Extended Example: Iteration 2

Increasing the value of $x_3$ would increase the objective value.

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}$$

The third constraint is the tightest and limits how much we can increase $x_3$.

Switch roles of $x_3$ and $x_5$:

- Solving for $x_3$ yields:
  $$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} - \frac{x_6}{8}.$$  

- Substitute this into $x_3$ in the other three equations
Extended Example: Iteration 3

\[ z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \]

\[ x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \]

\[ x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \]

\[ x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \]
Extended Example: Iteration 3

\[ z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \]

\[ x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \]

\[ x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \]

\[ x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \]

Basic solution: \((x_1, x_2, \ldots, x_6) = (\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0)\) with objective value \(\frac{111}{4} = 27.75\)
Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

\[
\begin{align*}
    z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
    x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
    x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
    x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

Basic solution: $(x_1, x_2, \ldots, x_6) = (\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0)$ with objective value $\frac{111}{4} = 27.75$
Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}$$

The second constraint is the tightest and limits how much we can increase $x_2$. 
Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

\[
\begin{align*}
  z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
  x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
  x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
  x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

The second constraint is the tightest and limits how much we can increase $x_2$.

Switch roles of $x_2$ and $x_3$: 

Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}$$

The second constraint is the tightest and limits how much we can increase $x_2$.

Switch roles of $x_2$ and $x_3$:

- Solving for $x_2$ yields:
  $$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}.$$
Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

\[
\begin{align*}
  z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
  x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
  x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
  x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

The second constraint is the tightest and limits how much we can increase $x_2$.

Switch roles of $x_2$ and $x_3$:

- Solving for $x_2$ yields:
  \[
  x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}.
  \]
- Substitute this into $x_2$ in the other three equations
Extended Example: Iteration 4

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]

\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]

\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]

\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]
Extended Example: Iteration 4

Extended Example: Iteration 4

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]

\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]

\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]

\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Basic solution: \((\overline{x_1}, \overline{x_2}, \ldots, \overline{x_6}) = (8, 4, 0, 18, 0, 0)\) with objective value 28
Extended Example: Iteration 4

All coefficients are negative, and hence this basic solution is **optimal**!

\[
\begin{align*}
  z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
  x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
  x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
  x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{align*}
\]

Basic solution: \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (8, 4, 0, 18, 0, 0)\) with objective value 28
Extended Example: Visualization of SIMPLEX

Exercise: How many basic solutions (including non-feasible ones) are there?
Extended Example: Visualization of SIMPLEX

Exercise: How many basic solutions (including non-feasible ones) are there?
Extended Example: Visualization of SIMPLEX

Exercise: How many basic solutions (including non-feasible ones) are there?
Extended Example: Visualization of SIMPLEX

Exercise: How many basic solutions (including non-feasible ones) are there?
Extended Example: Alternative Runs (1/2)

\[
\begin{align*}
    z &= 3x_1 + x_2 + 2x_3 \\
    x_4 &= 30 - x_1 - x_2 - 3x_3 \\
    x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
    x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]
Extended Example: Alternative Runs (1/2)

\[ z = 3x_1 + x_2 + 2x_3 \]

\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]

\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]

\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_2 \) and \( x_5 \)

Switch roles of \( x_1 \) and \( x_6 \)

II. Linear Programming Simplex Algorithm
Extended Example: Alternative Runs (1/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_2 \) and \( x_5 \)

\[ z = 12 + 2x_1 - \frac{x_3}{2} - \frac{x_5}{2} \]
\[ x_2 = 12 - x_1 - \frac{5x_3}{2} - \frac{x_5}{2} \]
\[ x_4 = 18 - x_2 - \frac{x_3}{2} + \frac{x_5}{2} \]
\[ x_6 = 24 - 3x_1 + \frac{x_3}{2} + \frac{x_5}{2} \]
Extended Example: Alternative Runs (1/2)

\[
\begin{align*}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3 \\
\end{align*}
\]

Switch roles of \(x_2\) and \(x_5\)

\[
\begin{align*}
z &= 12 + 2x_1 - \frac{x_3}{2} - \frac{x_5}{2} \\
x_2 &= 12 - x_1 - \frac{5x_3}{2} - \frac{x_5}{2} \\
x_4 &= 18 - x_2 - \frac{x_3}{2} + \frac{x_5}{2} \\
x_6 &= 24 - 3x_1 + \frac{x_3}{2} + \frac{x_5}{2} \\
\end{align*}
\]

Switch roles of \(x_1\) and \(x_6\)
Extended Example: Alternative Runs (1/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_2 \) and \( x_5 \)

\[ z = 12 + 2x_1 - \frac{x_3}{2} - \frac{x_5}{2} \]
\[ x_2 = 12 - x_1 - \frac{5x_3}{2} - \frac{x_5}{2} \]
\[ x_4 = 18 - x_2 - \frac{x_3}{2} + \frac{x_5}{2} \]
\[ x_6 = 24 - 3x_1 + \frac{x_3}{2} + \frac{x_5}{2} \]

Switch roles of \( x_1 \) and \( x_6 \)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]
Extended Example: Alternative Runs (2/2)

\[
\begin{align*}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3 \\
\end{align*}
\]
Extended Example: Alternative Runs (2/2)

\[
\begin{align*}
    z &= 3x_1 + x_2 + 2x_3 \\
    x_4 &= 30 - x_1 - x_2 - 3x_3 \\
    x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
    x_6 &= 36 - 4x_1 - x_2 - 2x_3 \\
\end{align*}
\]

Switch roles of \(x_3\) and \(x_5\)

Switch roles of \(x_1\) and \(x_6\)

Switch roles of \(x_2\) and \(x_3\)
Extended Example: Alternative Runs (2/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_3 \) and \( x_5 \)

\[ z = \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \]
\[ x_4 = \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \]
\[ x_3 = \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \]
\[ x_6 = \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5} \]
Extended Example: Alternative Runs (2/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_3 \) and \( x_5 \)

\[ z = \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \]
\[ x_4 = \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \]
\[ x_3 = \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \]
\[ x_6 = \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5} \]

Switch roles of \( x_1 \) and \( x_6 \)
Extended Example: Alternative Runs (2/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \( x_3 \) and \( x_5 \)

\[ z = \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \]
\[ x_4 = \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \]
\[ x_3 = \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \]
\[ x_6 = \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5} \]

Switch roles of \( x_1 \) and \( x_6 \)

\[ z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \]
\[ x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \]
\[ x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \]
\[ x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \]
Extended Example: Alternative Runs (2/2)

\[
\begin{align*}
  z & = 3x_1 + x_2 + 2x_3 \\
  x_4 & = 30 - x_1 - x_2 - 3x_3 \\
  x_5 & = 24 - 2x_1 - 2x_2 - 5x_3 \\
  x_6 & = 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]

Switch roles of \(x_3\) and \(x_5\)

\[
\begin{align*}
  z & = \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \\
  x_4 & = \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \\
  x_3 & = \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \\
  x_6 & = \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5}
\end{align*}
\]

Switch roles of \(x_1\) and \(x_6\), Switch roles of \(x_2\) and \(x_3\)

\[
\begin{align*}
  z & = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
  x_1 & = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
  x_3 & = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
  x_4 & = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]
Extended Example: Alternative Runs (2/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \(x_3\) and \(x_5\)

\[ z = \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \]
\[ x_4 = \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \]
\[ x_3 = \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \]
\[ x_6 = \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5} \]

Switch roles of \(x_1\) and \(x_6\)

\[ z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \quad z = \frac{28}{3} - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \quad x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \quad x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \quad x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]
The Pivot Step Formally

\[ \text{PIVOT}(N, B, A, b, c, v, l, e) \]

1. // Compute the coefficients of the equation for new basic variable \( x_e \).
2. let \( \hat{A} \) be a new \( m \times n \) matrix
3. \( \hat{b}_e = b_l/a_{le} \)
4. for each \( j \in N - \{e\} \)
5. \( \hat{a}_{ej} = a_{lj}/a_{le} \)
6. \( \hat{a}_{el} = 1/a_{le} \)
7. // Compute the coefficients of the remaining constraints.
8. for each \( i \in B - \{l\} \)
9. \( \hat{b}_i = b_i - a_{ie}\hat{b}_e \)
10. for each \( j \in N - \{e\} \)
11. \( \hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej} \)
12. \( \hat{a}_{il} = -a_{ie}\hat{a}_{el} \)
13. // Compute the objective function.
14. \( \hat{v} = v + c_e\hat{b}_e \)
15. for each \( j \in N - \{e\} \)
16. \( \hat{c}_j = c_j - c_e\hat{a}_{ej} \)
17. \( \hat{c}_l = -c_e\hat{a}_{el} \)
18. // Compute new sets of basic and nonbasic variables.
19. \( \hat{N} = N - \{e\} \cup \{l\} \)
20. \( \hat{B} = B - \{l\} \cup \{e\} \)
21. return \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})\)
The Pivot Step Formally

\[ \text{PIVOT}(N, B, A, b, c, v, l, e) \]

1. // Compute the coefficients of the equation for new basic variable \( x_e \).
2. let \( \hat{A} \) be a new \( m \times n \) matrix
3. \( \hat{b}_e = b_l/a_{le} \)
4. for each \( j \in N - \{e\} \)
5. \( \hat{a}_{ej} = a_{lj}/a_{le} \)
6. \( \hat{a}_{el} = 1/a_{le} \)
7. // Compute the coefficients of the remaining constraints.
8. for each \( i \in B - \{l\} \)
9. \( \hat{b}_i = b_i - a_{ie}\hat{b}_e \)
10. for each \( j \in N - \{e\} \)
11. \( \hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej} \)
12. \( \hat{a}_{il} = -a_{ie}\hat{a}_{el} \)
13. // Compute the objective function.
14. \( \hat{v} = v + c_e\hat{b}_e \)
15. for each \( j \in N - \{e\} \)
16. \( \hat{c}_j = c_j - c_e\hat{a}_{ej} \)
17. \( \hat{c}_l = -c_e\hat{a}_{el} \)
18. // Compute new sets of basic and nonbasic variables.
19. \( \hat{N} = N - \{e\} \cup \{l\} \)
20. \( \hat{B} = B - \{l\} \cup \{e\} \)
21. return \( (\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v}) \)
The Pivot Step Formally

\[ \text{PIVOT}(N, B, A, b, c, v, l, e) \]

1. // Compute the coefficients of the equation for new basic variable \( x_e \).
2. let \( \hat{A} \) be a new \( m \times n \) matrix
3. \( \hat{b}_e = b_l/a_{le} \)
4. for each \( j \in N - \{e\} \)
5. \( \hat{a}_{ej} = a_{lj}/a_{le} \)
6. \( \hat{a}_{el} = 1/a_{le} \)
7. // Compute the coefficients of the remaining constraints.
8. for each \( i \in B - \{l\} \)
9. \( \hat{b}_i = b_i - a_{ie} \hat{b}_e \)
10. for each \( j \in N - \{e\} \)
11. \( \hat{a}_{ij} = a_{ij} - a_{ie} \hat{a}_{ej} \)
12. \( \hat{a}_{il} = -a_{ie} \hat{a}_{el} \)
13. // Compute the objective function.
14. \( \hat{v} = v + c_e \hat{b}_e \)
15. for each \( j \in N - \{e\} \)
16. \( \hat{c}_j = c_j - c_e \hat{a}_{ej} \)
17. \( \hat{c}_l = -c_e \hat{a}_{el} \)
18. // Compute new sets of basic and nonbasic variables.
19. \( \hat{N} = N - \{e\} \cup \{l\} \)
20. \( \hat{B} = B - \{l\} \cup \{e\} \)
21. return \( (\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v}) \)

Rewrite “tight” equation for entering variable \( x_e \).

Substituting \( x_e \) into other equations.
The Pivot Step Formally

\[
\text{PIVOT}(N, B, A, b, c, v, l, e)
\]

1. // Compute the coefficients of the equation for new basic variable \(x_e\).
2. let \(\hat{A}\) be a new \(m \times n\) matrix
3. \(\hat{b}_e = b_l/a_{le}\)
4. for each \(j \in N \setminus \{e\}\)
   5. \(\hat{a}_{ej} = a_{lj}/a_{le}\)
   6. \(\hat{a}_{el} = 1/a_{le}\)
7. // Compute the coefficients of the remaining constraints.
8. for each \(i \in B \setminus \{l\}\)
   9. \(\hat{b}_i = b_i - a_{ie}\hat{b}_e\)
   10. for each \(j \in N \setminus \{e\}\)
      11. \(\hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej}\)
      12. \(\hat{a}_{il} = -a_{ie}\hat{a}_{el}\)
13. // Compute the objective function.
14. \(\hat{v} = v + c_e\hat{b}_e\)
15. for each \(j \in N \setminus \{e\}\)
   16. \(\hat{c}_j = c_j - c_e\hat{a}_{ej}\)
   17. \(\hat{c}_l = -c_e\hat{a}_{el}\)
18. // Compute new sets of basic and nonbasic variables.
19. \(\hat{N} = N \setminus \{e\} \cup \{l\}\)
20. \(\hat{B} = B \setminus \{l\} \cup \{e\}\)
21. return \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})\)
The Pivot Step Formally

**Algorithm PIVOT** \((N, B, A, b, c, \nu, l, e)\)

1. // Compute the coefficients of the equation for new basic variable \(x_e\).
2. let \(A\) be a new \(m \times n\) matrix
3. \(\hat{b}_e = b_l/a_{le}\)
4. for each \(j \in N - \{e\}\)
5. \(\hat{a}_{ej} = a_{lj}/a_{le}\)
6. \(\hat{a}_{el} = 1/a_{le}\)
7. // Compute the coefficients of the remaining constraints.
8. for each \(i \in B - \{l\}\)
9. \(\hat{b}_i = b_i - a_{ie}\hat{b}_e\)
10. for each \(j \in N - \{e\}\)
11. \(\hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej}\)
12. \(\hat{a}_{il} = -a_{ie}\hat{a}_{el}\)
13. // Compute the objective function.
14. \(\hat{\nu} = \nu + c_{e}\hat{b}_e\)
15. for each \(j \in N - \{e\}\)
16. \(\hat{c}_j = c_j - c_{e}\hat{a}_{ej}\)
17. \(\hat{c}_l = -c_{e}\hat{a}_{el}\)
18. // Compute new sets of basic and nonbasic variables.
19. \(\hat{N} = N - \{e\} \cup \{l\}\)
20. \(\hat{B} = B - \{l\} \cup \{e\}\)
21. return \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{\nu})\)
The Pivot Step Formally

\[ \text{PIVOT}(N, B, A, b, c, v, l, e) \]

1. // Compute the coefficients of the equation for new basic variable \( x_e \).
2. let \( \hat{A} \) be a new \( m \times n \) matrix
3. \( \hat{b}_e = b_l / a_{le} \)
4. for each \( j \in N - \{e\} \)
5. \( \hat{a}_{ej} = a_{lj} / a_{le} \)
6. \( \hat{a}_{el} = 1 / a_{le} \)
7. // Compute the coefficients of the remaining constraints.
8. for each \( i \in B - \{l\} \)
9. \( \hat{b}_i = b_i - a_{ie} \hat{b}_e \)
10. for each \( j \in N - \{e\} \)
11. \( \hat{a}_{ij} = a_{ij} - a_{ie} \hat{a}_{ej} \)
12. \( \hat{a}_{il} = -a_{ie} \hat{a}_{el} \)
13. // Compute the objective function.
14. \( \hat{v} = v + c_e \hat{b}_e \)
15. for each \( j \in N - \{e\} \)
16. \( \hat{c}_j = c_j - c_e \hat{a}_{ej} \)
17. \( \hat{c}_l = -c_e \hat{a}_{el} \)
18. // Compute new sets of basic and nonbasic variables.
19. \( \hat{N} = N - \{e\} \cup \{l\} \)
20. \( \hat{B} = B - \{l\} \cup \{e\} \)
21. return \( (\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v}) \)

Rewrite “tight” equation for entering variable \( x_e \).
Substituting \( x_e \) into other equations.
Substituting \( x_e \) into objective function.
Update non-basic and basic variables.
Consider a call to \texttt{PIVOT}(N, B, A, b, c, \nu, l, e) in which \( a_{le} \neq 0 \). Let the values returned from the call be \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{\nu})\), and let \( \bar{x} \) denote the basic solution after the call. Then

1. \( x_j = 0 \) for each \( j \in \hat{N} \).
2. \( x_e = b_l / a_{le} \).
3. \( x_i = \hat{b}_i - a_{ie} \hat{b}_e \) for each \( i \in \hat{B} \setminus \{e\} \).
Effect of the Pivot Step (extra material, non-examinable)

Consider a call to \texttt{PIVOT}(N, B, A, b, c, \nu, l, e) in which \(a_{le} \neq 0\). Let the values returned from the call be \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{\nu})\), and let \(\overline{x}\) denote the basic solution after the call. Then

1. \(\overline{x}_j = 0\) for each \(j \in \hat{N}\).
2. \(\overline{x}_e = b_l/a_{le}\).
3. \(\overline{x}_i = b_i - a_{ie}\hat{b}_e\) for each \(i \in \hat{B} \setminus \{e\}\).
Lemma 29.1

Consider a call to \texttt{P}IVOT\((N, B, A, b, c, \nu, l, e)\) in which \(a_{le} \neq 0\). Let the values returned from the call be \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{\nu})\), and let \(\bar{x}\) denote the basic solution after the call. Then

1. \(\bar{x}_j = 0\) for each \(j \in \hat{N}\).
2. \(\bar{x}_e = b_l / a_{le}\).
3. \(\bar{x}_i = b_i - a_{ie} \hat{b}_e\) for each \(i \in \hat{B} \setminus \{e\}\).

Proof:
Consider a call to \( \text{PIVOT}(N, B, A, b, c, v, l, e) \) in which \( a_{le} \neq 0 \). Let the values returned from the call be \( (\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v}) \), and let \( \bar{x} \) denote the basic solution after the call. Then

1. \( x_j = 0 \) for each \( j \in \hat{N} \).
2. \( x_e = b_l / a_{le} \).
3. \( x_i = b_i - a_{ie} \hat{b}_e \) for each \( i \in \hat{B} \setminus \{e\} \).

Proof:

1. holds since the basic solution always sets all non-basic variables to zero.
2. When we set each non-basic variable to 0 in a constraint

\[
x_i = \hat{b}_i - \sum_{j \in \hat{N}} \hat{a}_{ij} x_j,
\]

we have \( x_i = \hat{b}_i \) for each \( i \in \hat{B} \). Hence \( x_e = \hat{b}_e = b_l / a_{le} \).
3. After substituting into the other constraints, we have

\[
\bar{x}_i = \hat{b}_i = b_i - a_{ie} \hat{b}_e.
\]
Effect of the Pivot Step (extra material, non-examinable)

Lemma 29.1

Consider a call to \texttt{PIVOT}(N, B, A, b, c, v, l, e) in which \(a_{le} \neq 0\). Let the values returned from the call be \((\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})\), and let \(\bar{x}\) denote the basic solution after the call. Then

1. \(\bar{x}_j = 0\) for each \(j \in \hat{N}\).
2. \(\bar{x}_e = b_l / a_{le}\).
3. \(\bar{x}_i = b_i - a_{ie}\hat{b}_e\) for each \(i \in \hat{B} \setminus \{e\}\).

Proof:

1. holds since the basic solution always sets all non-basic variables to zero.
2. When we set each non-basic variable to 0 in a constraint
   \[ x_i = \hat{b}_i - \sum_{j \in \hat{N}} \hat{a}_{ij} x_j, \]
   we have \(\bar{x}_i = \hat{b}_i\) for each \(i \in \hat{B}\). Hence \(\bar{x}_e = \hat{b}_e = b_l / a_{le}\).
3. After substituting into the other constraints, we have
   \[ \bar{x}_i = \hat{b}_i = b_i - a_{ie}\hat{b}_e. \]
Questions:

- How do we determine whether a linear program is feasible?
- What do we do if the linear program is feasible, but the initial basic solution is not feasible?
- How do we determine whether a linear program is unbounded?
- How do we choose the entering and leaving variables?
Questions:

- How do we determine whether a linear program is feasible?
- What do we do if the linear program is feasible, but the initial basic solution is not feasible?
- How do we determine whether a linear program is unbounded?
- How do we choose the entering and leaving variables?

Example before was a particularly nice one!
The formal procedure SIMPLEX

\begin{algorithm}
\textbf{SIMPLEX}(A, b, c) \\
1 \quad (N, B, A, b, c, v) = \textsc{INITIALIZE-SIMPLEX}(A, b, c) \\
2 \quad \text{let } \Delta \text{ be a new vector of length } m \\
3 \quad \textbf{while} \text{ some index } j \in N \text{ has } c_j > 0 \\
4 \quad \quad \text{choose an index } e \in N \text{ for which } c_e > 0 \\
5 \quad \quad \textbf{for} \text{ each index } i \in B \\
6 \quad \quad \quad \text{if } a_{ie} > 0 \\
7 \quad \quad \quad \quad \Delta_i = b_i/a_{ie} \\
8 \quad \quad \quad \text{else } \Delta_i = \infty \\
9 \quad \quad \text{choose an index } l \in B \text{ that minimizes } \Delta_i \\
10 \quad \quad \textbf{if } \Delta_l = \infty \\
11 \quad \quad \quad \textbf{return} \text{ “unbounded”} \\
12 \quad \quad \textbf{else} (N, B, A, b, c, v) = \textsc{PIVOT}(N, B, A, b, c, v, l, e) \\
13 \quad \textbf{for } i = 1 \text{ to } n \\
14 \quad \quad \textbf{if } i \in B \\
15 \quad \quad \quad \bar{x}_i = b_i \\
16 \quad \quad \textbf{else } \bar{x}_i = 0 \\
17 \quad \textbf{return} (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \\
\end{algorithm}
The formal procedure SIMPLEX

\begin{algorithm}
\textsc{Simplex}(A, b, c)
1 \hspace{1em} (N, B, A, b, c, v) = \textsc{Initialize-Simplex}(A, b, c)
2 \hspace{1em} let \( \Delta \) be a new vector of length \( m \)
3 \hspace{1em} \textbf{while} some index \( j \in N \) has \( c_j > 0 \)
4 \hspace{2em} choose an index \( e \in N \) for which \( c_e > 0 \)
5 \hspace{2em} \textbf{for} each index \( i \in B \)
6 \hspace{3em} \textbf{if} \( a_{ie} > 0 \)
7 \hspace{4em} \( \Delta_i = b_i/a_{ie} \)
8 \hspace{3em} \textbf{else} \( \Delta_i = \infty \)
9 \hspace{2em} choose an index \( l \in B \) that minimizes \( \Delta_i \)
10 \hspace{2em} \textbf{if} \( \Delta_l = \infty \)
11 \hspace{3em} \textbf{return} “unbounded”
12 \hspace{2em} \textbf{else} \((N, B, A, b, c, v) = \textsc{Pivot}(N, B, A, b, c, v, l, e)\)
13 \hspace{2em} \textbf{for} \( i = 1 \) to \( n \)
14 \hspace{3em} \textbf{if} \( i \in B \)
15 \hspace{4em} \( \tilde{x}_i = b_i \)
16 \hspace{3em} \textbf{else} \( \tilde{x}_i = 0 \)
17 \hspace{2em} \textbf{return} \( (\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n) \)
\end{algorithm}

Returns a slack form with a feasible basic solution (if it exists)
The formal procedure SIMPLEX

\textbf{SIMPLEX}(A, b, c)

1. \((N, B, A, b, c, v) = \text{INITIALIZE-SIMPLEX}(A, b, c)\)
2. let \(\Delta\) be a new vector of length \(m\)
3. \textbf{while} some index \(j \in N\) has \(c_j > 0\)
4. \hspace{1em} choose an index \(e \in N\) for which \(c_e > 0\)
5. \hspace{1em} \textbf{for} each index \(i \in B\)
6. \hspace{2em} \textbf{if} \(a_{ie} > 0\)
7. \hspace{3em} \(\Delta_i = b_i/a_{ie}\)
8. \hspace{2em} \textbf{else} \(\Delta_i = \infty\)
9. \hspace{1em} choose an index \(l \in B\) that minimizes \(\Delta_i\)
10. \hspace{1em} \textbf{if} \(\Delta_l = \infty\)
11. \hspace{2em} \textbf{return} “unbounded”
12. \hspace{1em} \textbf{else} \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, e)\)
13. \hspace{1em} \textbf{for} \(i = 1\) to \(n\)
14. \hspace{2em} \textbf{if} \(i \in B\)
15. \hspace{3em} \(\bar{x}_i = b_i\)
16. \hspace{2em} \textbf{else} \(\bar{x}_i = 0\)
17. \hspace{1em} \textbf{return} \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)\)

Returns a slack form with a feasible basic solution (if it exists)
The formal procedure \textsc{Simplex}\( (A, b, c) \)

1. \( (N, B, A, b, c, v) = \text{\textsc{Initialize-Simplex}}(A, b, c) \)
2. let \( \Delta \) be a new vector of length \( m \)
3. while some index \( j \in N \) has \( c_j > 0 \)
4. choose an index \( e \in N \) for which \( c_e > 0 \)
5. for each index \( i \in B \)
6. \hspace{1em} if \( a_{ie} > 0 \)
7. \hspace{1em} \quad \Delta_i = b_i/a_{ie} \)
8. \hspace{1em} else \( \Delta_i = \infty \)
9. choose an index \( l \in B \) that minimizes \( \Delta_i \)
10. if \( \Delta_l = \infty \)
11. \hspace{1em} return "unbounded"
12. else \( (N, B, A, b, c, v) = \text{\textsc{Pivot}}(N, B, A, b, c, v, l, e) \)
13. for \( i = 1 \) to \( n \)
14. \hspace{1em} if \( i \in B \)
15. \hspace{1em} \quad \bar{x}_i = b_i \)
16. \hspace{1em} else \( \bar{x}_i = 0 \)
17. return \( (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \)

Returns a slack form with a feasible basic solution (if it exists)

Main Loop:
The formal procedure SIMPLEX

SIMPLEX\((A, b, c)\)

1. \((N, B, A, b, c, v) = \text{INITIALIZE-SIMPLEX}(A, b, c)\)
2. let \(\Delta\) be a new vector of length \(m\)
3. while some index \(j \in N\) has \(c_j > 0\)
4. choose an index \(e \in N\) for which \(c_e > 0\)
5. for each index \(i \in B\)
   6. if \(a_{ie} > 0\)
      7. \(\Delta_i = b_i / a_{ie}\)
   8. else \(\Delta_i = \infty\)
9. choose an index \(l \in B\) that minimizes \(\Delta_i\)
10. if \(\Delta_l = \infty\)
11. return “unbounded”
12. else \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, e)\)
13. for \(i = 1\) to \(n\)
14. if \(i \in B\)
15. \(\bar{x}_i = b_i\)
16. else \(\bar{x}_i = 0\)
17. return \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)\)

Returns a slack form with a feasible basic solution (if it exists)

Main Loop:
- terminates if all coefficients in objective function are negative
- Line 4 picks entering variable \(x_e\) with negative coefficient
- Lines 6 – 9 pick the tightest constraint, associated with \(x_l\)
- Line 11 returns “unbounded” if there are no constraints
- Line 12 calls PIVOT, switching roles of \(x_l\) and \(x_e\)
The formal procedure SIMPLEX

SIMPLEX(A, b, c)
1 (N, B, A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
2 let Δ be a new vector of length \( m \)
3 while some index \( j \in N \) has \( c_j > 0 \)
4 choose an index \( e \in N \) for which \( c_e > 0 \)
5 for each index \( i \in B \)
6 if \( a_{ie} > 0 \)
7 \( \Delta_i = b_i/a_{ie} \)
8 else \( \Delta_i = \infty \)
9 choose an index \( l \in B \) that minimizes \( \Delta_i \)
10 if \( \Delta_l = \infty \)
11 return “unbounded”
12 else \((N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, e)\)
13 for \( i = 1 \) to \( n \)
14 if \( i \in B \)
15 \( \bar{x}_i = b_i \)
16 else \( \bar{x}_i = 0 \)
17 return \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)\)

Returns a slack form with a feasible basic solution (if it exists)

Main Loop:
- terminates if all coefficients in objective function are negative
- Line 4 picks entering variable \( x_e \) with negative coefficient
- Lines 6 – 9 pick the tightest constraint, associated with \( x_l \)
- Line 11 returns “unbounded” if there are no constraints
- Line 12 calls PIVOT, switching roles of \( x_l \) and \( x_e \)

Return corresponding solution.
The formal procedure SIMPLEX

SIMPLEX\((A, b, c)\)

1. \((N, B, A, b, c, v) = \text{INITIALIZE-SIMPLEX}(A, b, c)\)
2. let \(\Delta\) be a new vector of length \(m\)
3. while some index \(j \in N\) has \(c_j > 0\)
4. choose an index \(e \in N\) for which \(c_e > 0\)
5. for each index \(i \in B\)
6. \quad if \(a_{ie} > 0\)
7. \quad \quad \Delta_i = b_i / a_{ie}
8. \quad else \(\Delta_i = \infty\)
9. choose an index \(l \in B\) that minimizes \(\Delta_i\)
10. \quad if \(\Delta_l = \infty\)
11. \quad \quad return “unbounded”
12. \quad else \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, e)\)
13. for \(i = 1\) to \(n\)
14. \quad if \(i \in B\)
15. \quad \quad \tilde{x}_i = b_i
16. \quad else \(\tilde{x}_i = 0\)
17. return \((\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n)\)

Lemma 29.2

Suppose the call to \text{INITIALIZE-SIMPLEX} in line 1 returns a slack form for which the basic solution is feasible. Then if \text{SIMPLEX} returns a solution, it is a feasible solution. If \text{SIMPLEX} returns “unbounded”, the linear program is unbounded.
The formal procedure **SIMPLEX**

\[ \text{SIMPLEX}(A, b, c) \]

1. \((N, B, A, b, c, v) = \text{INITIALIZE-SIMPLEX}(A, b, c)\)
2. let \( \Delta \) be a new vector of length \( m \)
3. \textbf{while} some index \( j \in N \) has \( c_j > 0 \)
4. choose an index \( e \in N \) for which \( c_e > 0 \)
5. for each index \( i \in B \)
6. \quad \textbf{if} \( a_{ie} > 0 \)
7. \quad \quad \Delta_i = b_i / a_{ie}
8. \quad \textbf{else} \( \Delta_i = \infty \)
9. choose an index \( l \in B \) that minimizes \( \Delta_i \)
10. \textbf{if} \( \Delta_l = \infty \)
11. \quad \textbf{return} “unbounded”

\[ \text{Proof is based on the following three-part loop invariant:} \]

- \( 1. \) the slack form is always equivalent to the one returned by \( \text{INITIALIZE-SIMPLEX} \)
- \( 2. \) for each \( i \in B \), we have \( \Delta_i \geq 0 \)
- \( 3. \) the basic solution associated with the (current) slack form is feasible.

\[ \text{Lemma 29.2} \]

Suppose the call to \( \text{INITIALIZE-SIMPLEX} \) in line 1 returns a slack form for which the basic solution is feasible. Then if \( \text{SIMPLEX} \) returns a solution, it is a feasible solution. If \( \text{SIMPLEX} \) returns “unbounded”, the linear program is unbounded.

**Returns a slack form with a feasible basic solution (if it exists)**
The formal procedure SIMPLEX

SIMPLEX\,(A,\,b,\,c)  
1  \( (N,\,B,\,A,\,b,\,c,\,\nu) = \text{INITIALIZE-SIMPLEX}\,(A,\,b,\,c) \)
2  let \( \Delta \) be a new vector of length \( m \)
3  while some index \( j \in N \) has \( c_j > 0 \)
4  choose an index \( e \in N \) for which \( c_e > 0 \)
5  for each index \( i \in B \)
6    if \( a_{ie} > 0 \)
7      \( \Delta_i = b_i / a_{ie} \)
8    else \( \Delta_i = \infty \)
9  choose an index \( l \in B \) that minimizes \( \Delta_i \)
10  if \( \Delta_l = \infty \)
11    return “unbounded”

Proof is based on the following three-part loop invariant:
1. the slack form is always equivalent to the one returned by \text{INITIALIZE-SIMPLEX},
2. for each \( i \in B \), we have \( b_i \geq 0 \),
3. the basic solution associated with the (current) slack form is feasible.

Returns a slack form with a feasible basic solution (if it exists)

Lemma 29.2

Suppose the call to \text{INITIALIZE-SIMPLEX} in line 1 returns a slack form for which the basic solution is feasible. Then if SIMPLEX returns a solution, it is a feasible solution. If SIMPLEX returns “unbounded”, the linear program is unbounded.
Termination

Degeneracy: One iteration of SIMPLEX leaves the objective value unchanged.
Termination

Degeneracy: One iteration of SIMPLEX leaves the objective value unchanged.

\[ z = x_1 + x_2 + x_3 \]
\[ x_4 = 8 - x_1 - x_2 \]
\[ x_5 = x_2 - x_3 \]
Degeneracy: One iteration of SIMPLEX leaves the objective value unchanged.

\[ z = x_1 + x_2 + x_3 \]
\[ x_4 = 8 - x_1 - x_2 \]
\[ x_5 = x_2 - x_3 \]

Pivot with \( x_1 \) entering and \( x_4 \) leaving.
Degeneracy: One iteration of SIMPLEX leaves the objective value unchanged.

\[ z = x_1 + x_2 + x_3 \]
\[ x_4 = 8 - x_1 - x_2 \]
\[ x_5 = x_2 - x_3 \]

Pivot with \( x_1 \) entering and \( x_4 \) leaving

\[ z = 8 + x_3 - x_4 \]
\[ x_1 = 8 - x_2 - x_4 \]
\[ x_5 = x_2 - x_3 \]
**Degeneracy**: One iteration of SIMPLEX leaves the objective value unchanged.

\[
\begin{align*}
  z &= x_1 + x_2 + x_3 \\
  x_4 &= 8 - x_1 - x_2 \\
  x_5 &= x_2 - x_3
\end{align*}
\]

Pivot with \(x_1\) entering and \(x_4\) leaving

\[
\begin{align*}
  z &= 8 + x_3 - x_4 \\
  x_1 &= 8 - x_2 - x_4 \\
  x_5 &= x_2 - x_3
\end{align*}
\]

Pivot with \(x_3\) entering and \(x_5\) leaving
**Degeneracy**: One iteration of SIMPLEX leaves the objective value unchanged.

\[
\begin{align*}
z &= x_1 + x_2 + x_3 \\
x_4 &= 8 - x_1 - x_2 \\
x_5 &= x_2 - x_3
\end{align*}
\]

Pivot with \(x_1\) entering and \(x_4\) leaving

\[
\begin{align*}
z &= 8 + x_3 - x_4 \\
x_1 &= 8 - x_2 - x_4 \\
x_5 &= x_2 - x_3
\end{align*}
\]

Pivot with \(x_3\) entering and \(x_5\) leaving

\[
\begin{align*}
z &= 8 + x_2 - x_4 - x_5 \\
x_1 &= 8 - x_2 - x_4 \\
x_3 &= x_2 - x_5
\end{align*}
\]
Degeneracy: One iteration of SIMPLEX leaves the objective value unchanged.

\[
z = x_1 + x_2 + x_3
\]
\[
x_4 = 8 - x_1 - x_2
\]
\[
x_5 = x_2 - x_3
\]

Pivot with \(x_1\) entering and \(x_4\) leaving

\[
z = 8 + x_3 - x_4
\]
\[
x_1 = 8 - x_2
\]
\[
x_5 = x_2 - x_3
\]

Pivot with \(x_3\) entering and \(x_5\) leaving

Cycling: If additionally slack form at two iterations are identical, SIMPLEX fails to terminate!
Exercise: Execute one more step of the Simplex Algorithm on the tableau from the previous slide.
Termination and Running Time

**Cycling**: **SIMPLEX** may fail to terminate.

1. **Bland’s rule**: Choose entering variable with smallest index.
2. **Random rule**: Choose entering variable uniformly at random.
3. **Perturbation**: Perturb the input slightly so that it is impossible to have two solutions with the same objective value.

**Anti-Cycling Strategies**

Assuming **INITIALIZE-SIMPLEX** returns a slack form for which the basic solution is feasible, **SIMPLEX** either reports that the program is unbounded or returns a feasible solution in at most \((n + m)m\) iterations.

**Lemma 29.7**

It is theoretically possible, but very rare in practice. Replace each \(b_i\) by \(\hat{b}_i = b_i + \epsilon_i\), where \(\epsilon_i \gg \epsilon_i + 1\) are all small.

Every set \(B\) of basic variables uniquely determines a slack form, and there are at most \((n + m)m\) unique slack forms.
Termination and Running Time

**Cycling**: SIMPLEX may fail to terminate.

1. Bland's rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

**Anti-Cycling Strategies**

Assuming INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that the program is unbounded or returns a feasible solution in at most \((n + m \choose m)\) iterations.

**Lemma 29.7**

It is theoretically possible, but very rare in practice. Replace each \(b_i\) by \(\hat{b}_i = b_i + \epsilon_i\), where \(\epsilon_i \gg \epsilon_i + 1\) are all small.

Every set \(B\) of basic variables uniquely determines a slack form, and there are at most \((n + m \choose m)\) unique slack forms.
Termination and Running Time

Cycling: SIMPLEX may fail to terminate.

1. Bland's rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Anti-Cycling Strategies

It is theoretically possible, but very rare in practice.
Cycling: SIMPLEX may fail to terminate.

1. Bland’s rule: Choose entering variable with smallest index

Anti-Cycling Strategies

It is theoretically possible, but very rare in practice.
Termination and Running Time

**Cycling:** SIMPLEX may fail to terminate.

1. **Bland's rule:** Choose entering variable with smallest index
2. **Random rule:** Choose entering variable uniformly at random

**Anti-Cycling Strategies**

- **Bland's rule:** Choose entering variable with smallest index
- **Random rule:** Choose entering variable uniformly at random

It is theoretically possible, but very rare in practice.
Termination and Running Time

Cycling: SIMPLEX may fail to terminate.

1. Bland’s rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Anti-Cycling Strategies

It is theoretically possible, but very rare in practice.
Termination and Running Time

Cycling: SIMPLEX may fail to terminate.

1. Bland's rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Anti-Cycling Strategies

Replace each $b_i$ by $\tilde{b}_i = b_i + \epsilon_i$, where $\epsilon_i \gg \epsilon_{i+1}$ are all small.

It is theoretically possible, but very rare in practice.
Termination and Running Time

Cycling: SIMPLEX may fail to terminate.

1. Bland’s rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Replace each $b_i$ by $\hat{b}_i = b_i + \epsilon_i$, where $\epsilon_i \gg \epsilon_{i+1}$ are all small.

Lemma 29.7

Assuming INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that the program is unbounded or returns a feasible solution in at most $\binom{n+m}{m}$ iterations.
Termination and Running Time

**Cycling**: SIMPLEX may fail to terminate.

1. Bland’s rule: Choose entering variable with smallest index
2. Random rule: Choose entering variable uniformly at random
3. Perturbation: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Anti-Cycling Strategies

**Lemma 29.7**

It is theoretically possible, but very rare in practice.

Replace each \( b_i \) by \( \hat{b}_i = b_i + \epsilon_i \), where \( \epsilon_i \gg \epsilon_{i+1} \) are all small.

Assuming \( \text{INITIALIZE-SIMPLEX} \) returns a slack form for which the basic solution is feasible, SIMPLEX either reports that the program is unbounded or returns a feasible solution in at most \( \binom{n+m}{m} \) iterations.

Every set \( B \) of basic variables uniquely determines a slack form, and there are at most \( \binom{n+m}{m} \) unique slack forms.
Outline

Introduction

Formulating Problems as Linear Programs

Standard and Slack Forms

Simplex Algorithm

Finding an Initial Solution
Finding an Initial Solution

maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Conversion into slack form
Basic solution \((x_1, x_2, x_3, x_4) = (0, 0, 2, -4)\) is not feasible!
maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 &- x_2 \leq 2 \\
x_1 &- 5x_2 \leq -4 \\
x_1, x_2 &\geq 0
\end{align*}
\]
Conversion into slack form

Basic solution \((x_1, x_2, x_3, x_4) = (0, 0, 2, -4)\) is not feasible!
Finding an Initial Solution

maximize \[ 2x_1 - x_2 \]
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

\[ z = 2x_1 - x_2 \]
\[ x_3 = 2 - 2x_1 + x_2 \]
\[ x_4 = -4 - x_1 + 5x_2 \]

Basic solution \((x_1, x_2, x_3, x_4) = (0, 0, 2, -4)\) is not feasible!

Conversion into slack form
maximize \[ 2x_1 - x_2 \]
subject to
\[ \begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*} \]
Geometric Illustration

maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Questions:
How to determine whether there is any feasible solution?
If there is one, how to determine an initial basic solution?
Geometric Illustration

maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Questions:

- How to determine whether there is any feasible solution?
- If there is one, how to determine an initial basic solution?
Formulating an Auxiliary Linear Program

maximize \( \sum_{j=1}^{n} c_j x_j \)
subject to
\[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \] for \( i = 1, 2, \ldots, m \),
\[ x_j \geq 0 \] for \( j = 1, 2, \ldots, n \)
Formulating an Auxiliary Linear Program

maximize $\sum_{j=1}^{n} c_j x_j$

subject to

$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for $i = 1, 2, \ldots, m,$

$x_j \geq 0$ for $j = 1, 2, \ldots, n$

Formulating an Auxiliary Linear Program
Formulating an Auxiliary Linear Program

maximize \( \sum_{j=1}^{n} c_j x_j \)
subject to
\[
\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m,
\]
\[
x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n
\]

maximize \(-x_0\)
subject to
\[
\sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i \quad \text{for } i = 1, 2, \ldots, m,
\]
\[
x_j \geq 0 \quad \text{for } j = 0, 1, \ldots, n
\]
Formulating an Auxiliary Linear Program

maximize $\sum_{j=1}^{n} c_j x_j$
subject to
$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for $i = 1, 2, \ldots, m,$
$x_j \geq 0$ for $j = 1, 2, \ldots, n$

maximize $-x_0$
subject to
$\sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i$ for $i = 1, 2, \ldots, m,$
$x_j \geq 0$ for $j = 0, 1, \ldots, n$

Lemma 29.11

Let $L_{aux}$ be the auxiliary LP of a linear program $L$ in standard form. Then $L$ is feasible if and only if the optimal objective value of $L_{aux}$ is 0.
Formulating an Auxiliary Linear Program

Let \( L_{aux} \) be the auxiliary LP of a linear program \( L \) in standard form. Then \( L \) is feasible if and only if the optimal objective value of \( L_{aux} \) is 0.

**Proof.**
Formulating an Auxiliary Linear Program

\[
\text{maximize } \sum_{j=1}^{n} c_j x_j \\
\text{subject to } \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m, \\
x_j \geq 0 \text{ for } j = 1, 2, \ldots, n
\]

\[
\text{Formulating an Auxiliary Linear Program}
\]

\[
\text{maximize } -x_0 \\
\text{subject to } \sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i \text{ for } i = 1, 2, \ldots, m, \\
x_j \geq 0 \text{ for } j = 0, 1, \ldots, n
\]

Lemma 29.11

Let \( L_{aux} \) be the auxiliary LP of a linear program \( L \) in standard form. Then \( L \) is feasible if and only if the optimal objective value of \( L_{aux} \) is 0.

Proof.

- “⇒”: Suppose \( L \) has a feasible solution \( \bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \)
Formulating an Auxiliary Linear Program

\[ \text{maximize} \quad \sum_{j=1}^{n} c_j x_j \]

subject to

\[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m, \]

\[ x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n \]

Let \( L_{aux} \) be the auxiliary LP of a linear program \( L \) in standard form. Then \( L \) is feasible if and only if the optimal objective value of \( L_{aux} \) is 0.

**Proof.**

- \( \Rightarrow \): Suppose \( L \) has a feasible solution \( \bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \)
- \( \bar{x}_0 = 0 \) combined with \( \bar{x} \) is a feasible solution to \( L_{aux} \) with objective value 0.
Formulating an Auxiliary Linear Program

\[
\begin{align*}
\text{maximize} & \quad \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m, \\
& \quad x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n
\end{align*}
\]

Lemma 29.11

Let \( L_{aux} \) be the auxiliary LP of a linear program \( L \) in standard form. Then \( L \) is feasible if and only if the optimal objective value of \( L_{aux} \) is 0.

Proof.

- \( \Rightarrow \) : Suppose \( L \) has a feasible solution \( \bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \)
  - \( \bar{x}_0 = 0 \) combined with \( \bar{x} \) is a feasible solution to \( L_{aux} \) with objective value 0.
  - Since \( \bar{x}_0 \geq 0 \) and the objective is to maximize \( -x_0 \), this is optimal for \( L_{aux} \)
Formulating an Auxiliary Linear Program

Let $L_{aux}$ be the auxiliary LP of a linear program $L$ in standard form. Then $L$ is feasible if and only if the optimal objective value of $L_{aux}$ is 0.

**Proof.**

- "⇒": Suppose $L$ has a feasible solution $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$
  - $\bar{x}_0 = 0$ combined with $\bar{x}$ is a feasible solution to $L_{aux}$ with objective value 0.
  - Since $\bar{x}_0 \geq 0$ and the objective is to maximize $-x_0$, this is optimal for $L_{aux}$
- "⇐": Suppose that the optimal objective value of $L_{aux}$ is 0
Formulating an Auxiliary Linear Program

Let $L_{aux}$ be the auxiliary LP of a linear program $L$ in standard form. Then $L$ is feasible if and only if the optimal objective value of $L_{aux}$ is 0.

**Proof.**

- $\Rightarrow$: Suppose $L$ has a feasible solution $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$
  - $\bar{x}_0 = 0$ combined with $\bar{x}$ is a feasible solution to $L_{aux}$ with objective value 0.
  - Since $\bar{x}_0 \geq 0$ and the objective is to maximize $-x_0$, this is optimal for $L_{aux}$
- $\Leftarrow$: Suppose that the optimal objective value of $L_{aux}$ is 0
  - Then $\bar{x}_0 = 0$, and the remaining solution values $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ satisfy $L$.  

maximize $\sum_{j=1}^{n} c_j x_j$
subject to $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for $i = 1, 2, \ldots, m,$
$x_j \geq 0$ for $j = 1, 2, \ldots, n$

maximize $-x_0$
subject to $\sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i$ for $i = 1, 2, \ldots, m,$
$x_j \geq 0$ for $j = 0, 1, \ldots, n$
Let $L_{\text{aux}}$ be the auxiliary LP of a linear program $L$ in standard form. Then $L$ is feasible if and only if the optimal objective value of $L_{\text{aux}}$ is 0.

**Proof.**

- “⇒”: Suppose $L$ has a feasible solution $\overline{x} = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$
  - $\overline{x}_0 = 0$ combined with $\overline{x}$ is a feasible solution to $L_{\text{aux}}$ with objective value 0.
  - Since $\overline{x}_0 \geq 0$ and the objective is to maximize $-x_0$, this is optimal for $L_{\text{aux}}$
- “⇐”: Suppose that the optimal objective value of $L_{\text{aux}}$ is 0
  - Then $\overline{x}_0 = 0$, and the remaining solution values $(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$ satisfy $L$. □
**INITIALIZE-SIMPLEX**

**INITIALIZE-SIMPLEX** \((A, b, c)\)

1. let \(k\) be the index of the minimum \(b_i\)
2. \(\textbf{if } b_k \geq 0 \quad //\) is the initial basic solution feasible?
3. \(\textbf{return } (\{1, 2, \ldots, n\}, \{n + 1, n + 2, \ldots, n + m\}, A, b, c, 0)\)
4. form \(L_{aux}\) by adding \(-x_0\) to the left-hand side of each constraint
   and setting the objective function to \(-x_0\)
5. let \((N, B, A, b, c, v)\) be the resulting slack form for \(L_{aux}\)
6. \(l = n + k\)
7. \(// L_{aux}\) has \(n + 1\) nonbasic variables and \(m\) basic variables.
8. \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, 0)\)
9. \(//\) The basic solution is now feasible for \(L_{aux}\).
10. iterate the **while** loop of lines 3–12 of **SIMPLEX** until an optimal solution
    to \(L_{aux}\) is found
11. \(\textbf{if }\) the optimal solution to \(L_{aux}\) sets \(x_0\) to 0
12. \(\textbf{if } x_0\) is basic
13. \(\textbf{perform one (degenerate) pivot to make it nonbasic}\)
14. \(\textbf{from the final slack form of } L_{aux}, \text{ remove } x_0 \text{ from the constraints and}\)
    \(\text{restore the original objective function of } L, \text{ but replace each basic}\)
    \(\text{variable in this objective function by the right-hand side of its}\)
    \(\text{associated constraint}\)
15. \(\textbf{return }\) the modified final slack form
16. **else return** “infeasible”
**INITIALIZE-SIMPLEX**

**INITIALIZE-SIMPLEX** \((A, b, c)\)

1. let \(k\) be the index of the minimum \(b_i\)
2. if \(b_k \geq 0\)  // is the initial basic solution feasible?
   3. return \(\{1, 2, \ldots, n\}, \{n + 1, n + 2, \ldots, n + m\}, A, b, c, 0\)
4. form \(L_{aux}\) by adding \(-x_0\) to the left-hand side of each constraint
   and setting the objective function to \(-x_0\)
5. let \((N, B, A, b, c, v)\) be the resulting slack form for \(L_{aux}\)
6. \(l = n + k\)
7. // \(L_{aux}\) has \(n + 1\) nonbasic variables and \(m\) basic variables.
8. \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, 0)\)
9. // The basic solution is now feasible for \(L_{aux}\).
10. iterate the **while** loop of lines 3–12 of SIMPLEX until an optimal solution
    to \(L_{aux}\) is found
11. if the optimal solution to \(L_{aux}\) sets \(\bar{x}_0\) to 0
12.   if \(\bar{x}_0\) is basic
13.      perform one (degenerate) pivot to make it nonbasic
14.      from the final slack form of \(L_{aux}\), remove \(x_0\) from the constraints and
      restore the original objective function of \(L\), but replace each basic
      variable in this objective function by the right-hand side of its
      associated constraint
15.     return the modified final slack form
16. else return “infeasible”

Test solution with \(N = \{1, 2, \ldots, n\}\), \(B = \{n + 1, n + 2, \ldots, n + m\}\), \(\bar{x}_i = b_i\) for \(i \in B\), \(\bar{x}_i = 0\) otherwise.
The initial basic feasible solution

\[
\text{maximize } \sum_{j=1}^{n} a_{ij} x_j \text{ subject to } \sum_{j=1}^{n} a_{ij} x_j = b_i \text{ for } i = 1, \ldots, m \quad (29.106)
\]

\[
x_j \geq 0 \text{ for } j = 0, 1, \ldots, n \quad (29.107)
\]

Then \( L \) is feasible if and only if the optimal objective value of \( L_{\text{aux}} \) is 0.

Proof

Suppose that \( L \) has a feasible solution \( x_0 \). Then the solution \( \bar{x} = 0 \) combined with \( x_0 \) is a feasible solution to \( L_{\text{aux}} \) with objective value 0. Since \( x_0 \) is a constraint of \( L_{\text{aux}} \) and the objective function is to maximize \( \sum_{j=1}^{n} \), this solution must be optimal for \( L_{\text{aux}} \).

Conversely, suppose that the optimal objective value of \( L_{\text{aux}} \) is 0. Then \( x_0 \), and the remaining solution values of \( x \) satisfy the constraints of \( L_{\text{aux}} \).

We now describe our strategy to find an initial basic feasible solution for a linear program \( L \) in standard form:

**Initialize-Simplex**

**Initialize-Simplex** \((A, b, c)\)

1. let \( k \) be the index of the minimum \( b_i \)
2. if \( b_k \geq 0 \) \hspace{1em} // is the initial basic solution feasible?
3. return \((\{1, 2, \ldots, n\}, \{n+1, n+2, \ldots, n+m\}, A, b, c, 0)\)
4. form \( L_{\text{aux}} \) by adding \(-x_0\) to the left-hand side of each constraint and setting the objective function to \(-x_0\)
5. let \((N, B, A, b, c, v)\) be the resulting slack form for \( L_{\text{aux}} \)
6. \( l = n + k \)
7. \( // L_{\text{aux}} \) has \( n + 1 \) nonbasic variables and \( m \) basic variables.
8. \((N, B, A, b, c, v) = \text{Pivot}(N, B, A, b, c, v, l, 0)\)
9. \( // \) The basic solution is now feasible for \( L_{\text{aux}} \).
10. iterate the while loop of lines 3–12 of Simplex until an optimal solution to \( L_{\text{aux}} \) is found
11. if the optimal solution to \( L_{\text{aux}} \) sets \( x_0 \) to 0
12. if \( x_0 \) is basic
13. perform one (degenerate) pivot to make it nonbasic
14. from the final slack form of \( L_{\text{aux}} \), remove \( x_0 \) from the constraints and restore the original objective function of \( L \), but replace each basic variable in this objective function by the right-hand side of its associated constraint
15. return the modified final slack form
16. else return “infeasible”
### Initialize-Simplex

**Initialize-Simplex** \((A, b, c)\)

1. Let \(k\) be the index of the minimum \(b_i\).
2. If \(b_k \geq 0\) then return \(\{1, 2, \ldots, n\}, \{n + 1, n + 2, \ldots, n + m\}, A, b, c, 0\).
3. Form \(L_{aux}\) by adding \(-x_0\) to the left-hand side of each constraint and setting the objective function to \(-x_0\).
4. Let \((N, B, A, b, c, \nu)\) be the resulting slack form for \(L_{aux}\).
5. \(l = n + k\).
6. \(\text{//} L_{aux}\) has \(n + 1\) nonbasic variables and \(m\) basic variables.
7. \((N, B, A, b, c, \nu) = \text{PIVOT}(N, B, A, b, c, \nu, l, 0)\).
8. The basic solution is now feasible for \(L_{aux}\).
9. Iterate the while loop of lines 3–12 of SIMPLEX until an optimal solution to \(L_{aux}\) is found.
10. If the optimal solution to \(L_{aux}\) sets \(x_0\) to 0 then:
    11. If \(x_0\) is basic then perform one (degenerate) pivot to make it nonbasic.
    12. From the final slack form of \(L_{aux}\), remove \(x_0\) from the constraints and restore the original objective function of \(L\), but replace each basic variable in this objective function by the right-hand side of its associated constraint.
13. Return the modified final slack form.
14. Else return “infeasible.”

---

**Test solution** with \(N = \{1, 2, \ldots, n\}, B = \{n + 1, n + 2, \ldots, n + m\}\), \(\bar{x}_i = b_i\) for \(i \in B\), \(\bar{x}_i = 0\) otherwise.

**Pivot step** with \(x_\ell\) leaving and \(x_0\) entering.

\(\ell\) will be the leaving variable so that \(x_\ell\) has the most negative value.
**INITIALIZE-SIMPLEX**

**INITIALIZE-SIMPLEX** \((A, b, c)\)

1. let \(k\) be the index of the minimum \(b_i\)
2. if \(b_k \geq 0\) \(\quad \text{// is the initial basic solution feasible?}\)
3. return \((\{1, 2, \ldots, n\}, \{n + 1, n + 2, \ldots, n + m\}, A, b, c, 0)\)
4. form \(L_{aux}\) by adding \(-x_0\) to the left-hand side of each constraint
   and setting the objective function to \(-x_0\)
5. let \((N, B, A, b, c, v)\) be the resulting slack form for \(L_{aux}\)
6. \(l = n + k\)
7. \(\quad \text{// \(L_{aux}\) has \(n + 1\) nonbasic variables and \(m\) basic variables.}\)
8. \((N, B, A, b, c, v) = \text{Pivot}(N, B, A, b, c, v, l, 0)\)
9. \(\quad \text{// The basic solution is now feasible for \(L_{aux}\).}\)
10. iterate the while loop of lines 3–12 of SIMPLEX until an optimal solution
    to \(L_{aux}\) is found
11. if the optimal solution to \(L_{aux}\) sets \(x_0\) to 0
12.    if \(x_0\) is basic
13.       perform one (degenerate) pivot to make it nonbasic
14.       from the final slack form of \(L_{aux}\), remove \(x_0\) from the constraints and
                restore the original objective function of \(L\), but replace each basic
                variable in this objective function by the right-hand side of its
                associated constraint
15.    return the modified final slack form
16. else return “infeasible”

---

Test solution with \(N = \{1, 2, \ldots, n\}, B = \{n + 1, n + 2, \ldots, n + m\}, x_i = b_i \) for \(i \in B, x_i = 0\) otherwise.

\(\ell\) will be the leaving variable so that \(x_\ell\) has the most negative value.

Pivot step with \(x_\ell\) leaving and \(x_0\) entering.

This pivot step does not change the value of any variable.
Example of INITIALIZE-SIMPLEX (1/3)

maximize \[ 2x_1 - x_2 \]
subject to
\[ 2x_1 - x_2 \leq 2 \]
\[ x_1 - 5x_2 \leq -4 \]
\[ x_1, x_2 \geq 0 \]

Formulating the auxiliary linear program
\[ z = -x_0 \]
\[ x_3 = 2 - 2x_1 + x_2 + x_0 \]
\[ x_4 = -4 - x_1 + 5x_2 + x_0 \]

Converting into slack form
Basic solution
\((0,0,0,2,-4)\) not feasible!
Example of **INITIALIZE-SIMPLEX** (1/3)

maximize \[ 2x_1 - x_2 \]

subject to

\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
-x_1 + 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Formulating the auxiliary linear program

Basic solution \( (0,0,0,2,-4) \) not feasible!
**Example of INITIALIZE-SIMPLEX (1/3)**

Maximize

\[
2x_1 - x_2
\]

Subject to

\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Formulating the auxiliary linear program

Maximize

\[-x_0\]

Subject to

\[
\begin{align*}
2x_1 - x_2 - x_0 & \leq 2 \\
x_1 - 5x_2 - x_0 & \leq -4 \\
x_1, x_2, x_0 & \geq 0
\end{align*}
\]

Basic solution

\[0, 0, 0, 2, -4\]

Not feasible!
Example of INITIALIZE-SIMPLEX (1/3)

maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Formulating the auxiliary linear program

maximize \(-x_0\)
subject to
\[
\begin{align*}
2x_1 - x_2 - x_0 & \leq 2 \\
x_1 - 5x_2 - x_0 & \leq -4 \\
x_1, x_2, x_0 & \geq 0
\end{align*}
\]

Converting into slack form

Basic solution \((0,0,0,2,−4)\) not feasible!
Example of **INITIALIZE-SIMPLEX** (1/3)

maximize \( 2x_1 - x_2 \)

subject to

\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Formulating the auxiliary linear program

maximize \( -x_0 \)

subject to

\[
\begin{align*}
2x_1 - x_2 - x_0 & \leq 2 \\
x_1 - 5x_2 - x_0 & \leq -4 \\
x_1, x_2, x_0 & \geq 0
\end{align*}
\]

Converting into slack form

\[
\begin{align*}
z & = -x_0 \\
x_3 & = 2 - 2x_1 + x_2 + x_0 \\
x_4 & = -4 - x_1 + 5x_2 + x_0
\end{align*}
\]
Example of INITIALIZE-SIMPLEX (1/3)

maximize \( 2x_1 - x_2 \)
subject to
\[
egin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0
\end{align*}
\]

Formulating the auxiliary linear program

maximize \(-x_0\)
subject to
\[
egin{align*}
2x_1 - x_2 - x_0 & \leq 2 \\
x_1 - 5x_2 - x_0 & \leq -4 \\
x_1, x_2, x_0 & \geq 0
\end{align*}
\]

Basic solution \((0, 0, 0, 2, -4)\) not feasible!

Converting into slack form

\[
egin{align*}
z &= -x_0 \\
x_3 &= 2 - 2x_1 + x_2 + x_0 \\
x_4 &= -4 - x_1 + 5x_2 + x_0
\end{align*}
\]
Example of INITIALIZE-SIMPlex (2/3)

\[ z = -x_0 \]
\[ x_3 = 2 - 2x_1 + x_2 + x_0 \]
\[ x_4 = -4 - x_1 + 5x_2 + x_0 \]

Pivot with \( x_0 \) entering and \( x_4 \) leaving
Pivot with \( x_2 \) entering and \( x_0 \) leaving

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Optimal solution has \( x_0 = 0 \), hence the initial problem was feasible!
Example of INITIALIZE-SIMPLEX (2/3)

\[ Z = -x_0 \]
\[ x_3 = 2 - 2x_1 + x_2 + x_0 \]
\[ x_4 = -4 - x_1 + 5x_2 + x_0 \]

Pivot with \( x_0 \) entering and \( x_4 \) leaving

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Optimal solution has \( x_0 = 0 \), hence the initial problem was feasible!
Example of INITIALIZE-SIMPLEX (2/3)

\[
\begin{align*}
    z & = - x_0 \\
    x_3 & = 2 - 2x_1 + x_2 + x_0 \\
    x_4 & = -4 - x_1 + 5x_2 + x_0
\end{align*}
\]

Pivot with \( x_0 \) entering and \( x_4 \) leaving

\[
\begin{align*}
    z & = -4 - x_1 + 5x_2 - x_4 \\
    x_0 & = 4 + x_1 - 5x_2 + x_4 \\
    x_3 & = 6 - x_1 - 4x_2 + x_4
\end{align*}
\]

Basic solution \((4,0,0,6,0)\) is feasible!

Optimal solution has \( x_0 = 0 \), hence the initial problem was feasible!
Example of INITIALIZE-SIMPLEX (2/3)

Basic solution (4, 0, 0, 6, 0) is feasible!
Example of INITIALIZE-SIMPLEX (2/3)

\[
\begin{align*}
    z &= -x_0 \\
    x_3 &= 2 - 2x_1 + x_2 + x_0 \\
    x_4 &= -4 - x_1 + 5x_2 + x_0 \\
\end{align*}
\]

Pivot with \(x_0\) entering and \(x_4\) leaving

\[
\begin{align*}
    z &= -4 - x_1 + 5x_2 - x_4 \\
    x_0 &= 4 + x_1 - 5x_2 + x_4 \\
    x_3 &= 6 - x_1 - 4x_2 + x_4 \\
\end{align*}
\]

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Pivot with \(x_2\) entering and \(x_0\) leaving
Example of **INITIALIZE-SIMPLEX** (2/3)

\[
\begin{align*}
    z & = -x_0 \\
    x_3 & = 2 - 2x_1 + x_2 + x_0 \\
    x_4 & = -4 - x_1 + 5x_2 + x_0
\end{align*}
\]

Pivot with \(x_0\) entering and \(x_4\) leaving

\[
\begin{align*}
    z & = -4 - x_1 + 5x_2 - x_4 \\
    x_0 & = 4 + x_1 - 5x_2 + x_4 \\
    x_3 & = 6 - x_1 - 4x_2 + x_4
\end{align*}
\]

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Pivot with \(x_2\) entering and \(x_0\) leaving

\[
\begin{align*}
    z & = -x_0 \\
    x_2 & = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
    x_3 & = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{align*}
\]
Example of INITIALIZE-SIMPLEX (2/3)

\[
\begin{align*}
  z &= 0 - 2x_1 + x_2 + x_0 \\
  x_3 &= 4 - 2x_1 + x_2 + x_0 \\
  x_4 &= 6 - x_1 - 4x_2 + x_4
\end{align*}
\]

Pivot with \( x_0 \) entering and \( x_4 \) leaving

\[
\begin{align*}
  z &= -4 - x_1 + 5x_2 - x_4 \\
  x_0 &= 4 + x_1 - 5x_2 + x_4 \\
  x_3 &= 6 - x_1 - 4x_2 + x_4
\end{align*}
\]

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Pivot with \( x_2 \) entering and \( x_0 \) leaving

\[
\begin{align*}
  z &= 0 - x_0 \\
  x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
  x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{align*}
\]

Optimal solution has \( x_0 = 0 \), hence the initial problem was feasible!
Example of INITIALIZE-SIMPLEX (3/3)

\[
\begin{align*}
    z &= -x_0 \\
    x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
    x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{align*}
\]
Example of INITIALIZE-SIMPLEX (3/3)

\[ z = -x_0 \]
\[ x_2 = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \]
\[ x_3 = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \]

Set \( x_0 = 0 \) and express objective function by non-basic variables.
Example of INITIALIZE-SIMPLEX (3/3)

\[
\begin{align*}
    z &= -x_0 \\
    x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
    x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{align*}
\]

Set \( x_0 = 0 \) and express objective function by non-basic variables:

\[
\begin{align*}
    2x_1 - x_2 &= 2x_1 - \left( \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \right) \\
    z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\
    x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
    x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{align*}
\]
Example of INITIALIZE-SIMPLEX (3/3)

\[ z = -x_0 \]
\[ x_2 = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \]
\[ x_3 = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \]

Set \( x_0 = 0 \) and express objective function by non-basic variables

\[ z = -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \]
\[ x_2 = \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \]
\[ x_3 = \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \]

Basic solution \((0, \frac{4}{5}, \frac{14}{5}, 0)\), which is feasible!
Example of **INITIALIZE-SIMPLEX** (3/3)

\[
\begin{align*}
z &= -x_0 \\
x_2 &= 4 - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
x_3 &= 14 + 4x_0 - 9x_1 + 5x_4
\end{align*}
\]

Set \( x_0 = 0 \) and express objective function by non-basic variables:

\[
2x_1 - x_2 = 2x_1 - \left( \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \right)
\]

Basic solution \((0, \frac{4}{5}, \frac{14}{5}, 0)\), which is feasible!

**Lemma 29.12**

If a linear program \( L \) has no feasible solution, then **INITIALIZE-SIMPLEX** returns “infeasible”. Otherwise, it returns a valid slack form for which the basic solution is feasible.
Theorem 29.13 (Fundamental Theorem of Linear Programming)

Any linear program $L$, given in standard form, either
1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded.

If $L$ is infeasible, SIMPLEX returns “infeasible”. If $L$ is unbounded, SIMPLEX returns “unbounded”. Otherwise, SIMPLEX returns an optimal solution with a finite objective value.
Fundamental Theorem of Linear Programming

Theorem 29.13 (Fundamental Theorem of Linear Programming)

Any linear program \( L \), given in standard form, either
1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded.

If \( L \) is infeasible, SIMPLEX returns “infeasible”. If \( L \) is unbounded, SIMPLEX returns “unbounded”. Otherwise, SIMPLEX returns an optimal solution with a finite objective value.

Proof requires the concept of duality, which is not covered in this course (for details see CLRS3, Chapter 29.4)
Workflow for Solving Linear Programs

1. Linear Program (in any form)
2. Standard Form
3. Slack Form

4. No Feasible Solution
   - INITIALIZE-SIMPLEX terminates

5. Feasible Basic Solution
   - INITIALIZE-SIMPLEX followed by SIMPLEX
     - LP unbounded
       - SIMPLEX terminates
     - LP bounded
       - SIMPLEX returns optimum
Linear Programming

Linear Programming

Linear Programming and Simplex: Summary and Outlook

extremely versatile tool for modelling problems of all kinds
basis of Integer Programming, to be discussed in later lectures

Linear Programming

In practice: usually terminates in polynomial time, i.e., $O(m + n)$
In theory: even with anti-cycling may need exponential time

Simplex Algorithm

Research Problem: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?

Interior-Point Methods: traverses the interior of the feasible set of solutions (not just vertices!)

Polynomial-Time Algorithms

II. Linear Programming Finding an Initial Solution
Linear Programming

- extremely versatile tool for modelling problems of all kinds
Linear Programming

- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures
Linear Programming

- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

Simplex Algorithm

- In practice: usually terminates in polynomial time, i.e., $O(m + n)$
Linear Programming

- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

Simplex Algorithm

- In practice: usually terminates in polynomial time, i.e., $O(m + n)$
- In theory: even with anti-cycling may need exponential time
Linear Programming

- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

Simplex Algorithm

- In practice: usually terminates in polynomial time, i.e., $O(m + n)$
- In theory: even with anti-cycling may need exponential time

Research Problem: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?
Linear Programming and Simplex: Summary and Outlook

Linear Programming
- extremely versatile tool for modelling problems of all kinds
- basis of **Integer Programming**, to be discussed in later lectures

Simplex Algorithm
- **In practice**: usually terminates in polynomial time, i.e., $O(m + n)$
- **In theory**: even with anti-cycling may need exponential time

**Research Problem**: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?

Polynomial-Time Algorithms
Linear Programming and Simplex: Summary and Outlook

Linear Programming
- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

Simplex Algorithm
- **In practice:** usually terminates in polynomial time, i.e., $O(m + n)$
- **In theory:** even with anti-cycling may need exponential time

Research Problem: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?

Polynomial-Time Algorithms
- **Interior-Point Methods:** traverses the interior of the feasible set of solutions (not just vertices!)
Linear Programming and Simplex: Summary and Outlook

Linear Programming
- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

Simplex Algorithm
- In practice: usually terminates in polynomial time, i.e., $O(m + n)$
- In theory: even with anti-cycling may need exponential time

Research Problem: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?

Polynomial-Time Algorithms
- Interior-Point Methods: traverses the interior of the feasible set of solutions (not just vertices!)
Which of the following statements are true?

1. In each iteration of the Simplex algorithm, the objective function increases.
2. There exist linear programs that have exactly two optimal solutions.
3. There exist linear programs that have infinitely many optimal solutions.
4. The Simplex algorithm always runs in worst-case polynomial time.
III. Approximation Algorithms: Covering Problems

Thomas Sauerwald

Easter 2021
Introduction

Vertex Cover

The Set-Covering Problem
Motivation

Many fundamental problems are NP-complete, yet they are too important to be abandoned.

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Strategies to cope with NP-complete problems

We will call these approximation algorithms.
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...
Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: **HAMILTON**, **3-SAT**, **VERTEX-COVER**, **KNAPSACK**, ...
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK, ...

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with **exponential running time** may be satisfactory.
2. Isolate important **special cases** which can be solved in polynomial-time.
3. Develop algorithms which find **near-optimal solutions** in polynomial-time.
Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...

**Strategies to cope with NP-complete problems**

1. If inputs (or solutions) are small, an algorithm with **exponential running time** may be satisfactory.
2. Isolate important **special cases** which can be solved in polynomial-time.
3. Develop algorithms which find **near-optimal solutions** in polynomial-time.

We will call these **approximation algorithms**.
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]
Performance Ratios for Approximation Algorithms

An algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

This covers both maximization and minimization problems.
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem:** $\frac{C^*}{C} \geq 1$

This covers both maximization and minimization problems.
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization** problem: $\frac{C^*}{C} \geq 1$
- **Minimization** problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

III. Covering Problems Introduction
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

This covers both maximization and minimization problems.

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

For many problems: tradeoff between runtime and approximation ratio.
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.
An algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

An approximation scheme is an approximation algorithm, which given any input and \( \epsilon > 0 \), is a \((1 + \epsilon)\)-approximation algorithm.
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and \( \epsilon > 0 \), is a \((1 + \epsilon)\)-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed \( \epsilon > 0 \), the runtime is polynomial in \( n \).
Performance Ratios for Approximation Algorithms

**Approximation Ratio**

An algorithm for a problem has **approximation ratio** \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

- **Maximization problem**: \( \frac{C^*}{C} \geq 1 \)
- **Minimization problem**: \( \frac{C}{C^*} \geq 1 \)

This covers both maximization and minimization problems.

**For many problems**: tradeoff between runtime and approximation ratio.

**Approximation Schemes**

An **approximation scheme** is an approximation algorithm, which given any input and \( \epsilon > 0 \), is a \((1 + \epsilon)\)-approximation algorithm.

- It is a **polynomial-time approximation scheme** (PTAS) if for any fixed \( \epsilon > 0 \), the runtime is polynomial in \( n \). For example, \( O(n^{2/\epsilon}) \).
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^{2/\epsilon}).$
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. 

III. Covering Problems

Introduction
Approximation Ratio

An algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

- Maximization problem: \( \frac{C^*}{C} \geq 1 \)
- Minimization problem: \( \frac{C}{C^*} \geq 1 \)

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and \( \epsilon > 0 \), is a \((1 + \epsilon)\)-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed \( \epsilon > 0 \), the runtime is polynomial in \( n \). For example, \( O(n^2/\epsilon) \).
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both \( 1/\epsilon \) and \( n \). For example, \( O((1/\epsilon)^2 \cdot n^3) \).

III. Covering Problems

Introduction
Outline

Introduction

Vertex Cover

The Set-Covering Problem
The Vertex-Cover Problem

Vertex Cover Problem

- **Given**: Undirected graph \( G = (V, E) \)
- **Goal**: Find a minimum-cardinality subset \( V' \subseteq V \) such that if \( (u, v) \in E(G) \), then \( u \in V' \) or \( v \in V' \).
The Vertex-Cover Problem

Vertex Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$. 

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task.
- Perform all tasks with the minimal amount of resources.

Extensions: weighted vertices or hypergraphs ($\Rightarrow$ Set-Covering Problem).
### The Vertex-Cover Problem

**Given:** Undirected graph $G = (V, E)$

**Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

---

**Applications:**

- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources

**Extensions:** weighted vertices or hypergraphs ($\rightarrow$ Set-Covering Problem)

---

Vertex Cover Problem

- **Given:** Undirected graph $G = (V, E)$
- **Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$. 

![Graph Example](image-url)
The Vertex-Cover Problem

Given: Undirected graph $G = (V, E)$

Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

We are covering edges by picking vertices!

Vertex Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.
The Vertex-Cover Problem

**Given:** Undirected graph $G = (V, E)$

**Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

We are covering edges by picking vertices!
The Vertex-Cover Problem

We are covering edges by picking vertices!

**Vertex Cover Problem**

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

**Applications:**

- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
- Extensions: weighted vertices or hypergraphs (⇝ Set-Covering Problem)
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task.
The Vertex-Cover Problem

Given: Undirected graph \( G = (V, E) \)

Goal: Find a minimum-cardinality subset \( V' \subseteq V \) such that if \((u, v) \in E(G)\), then \( u \in V' \) or \( v \in V' \).

This is an NP-hard problem.

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources

We are covering edges by picking vertices!
The Vertex-Cover Problem

Given: Undirected graph $G = (V, E)$
Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
- Extensions: weighted vertices or hypergraphs (\(\rightarrow\) Set-Covering Problem)
Exercise: Be creative and design your own algorithm for VERTEX-COVER!
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \( (G) \)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. **while** \( E' \neq \emptyset \)
4. \hspace{1em} let \((u, v)\) be an arbitrary edge of \( E' \)
5. \hspace{1em} \( C = C \cup \{u, v\} \)
6. \hspace{1em} remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. **return** \( C \)
An Approximation Algorithm based on Greedy

**35.1 The vertex-cover problem 1109**

Figure 35.1 illustrates how A\textsc{PRROX-VERTEX-COVER} operates on an example graph. The variable $C$ contains the vertex cover being constructed. Line 1 initializes $C$ to the empty set. Line 2 sets $E'$ to be a copy of the edge set $G:E$ of the graph. The loop of lines 3–6 repeatedly picks an edge $(u, v)$ from $E'$, adds it to $C$, and removes from $E'$ every edge incident on either $u$ or $v$. Line 7 returns $C$.

**A\textsc{PRROX-VERTEX-COVER}(G)**

1. $C = \emptyset$
2. $E' = G:E$
3. while $E' \neq \emptyset$
4. let $(u, v)$ be an arbitrary edge of $E'$
5. $C = C \cup \{u, v\}$
6. remove from $E'$ every edge incident on either $u$ or $v$
7. return $C$

III. Covering Problems

**Vertex Cover**
An Approximation Algorithm based on Greedy

**Approx-Vertex-Cover** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. while \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. return \(C\)

III. Covering Problems
Vertex Cover
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. **while** \( E' \neq \emptyset \)
   4. let \( (u, v) \) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
4. **return** \( C \)

III. Covering Problems
Vertex Cover
An Approximation Algorithm based on Greedy

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable $C$ contains the vertex cover being constructed. Line 1 initializes $C$ to the empty set. Line 2 sets $E'$ to be a copy of the edge set $G: E$ of the graph. The loop of lines 3–6 repeatedly picks an edge $(u, v)$ from $E'$, adds it to $C$, and removes from $E'$ every edge incident on either $u$ or $v$. Line 7 returns $C$.

**APPROX-VERTEX-COVER**($G$)

1. $C = \emptyset$
2. $E' = G: E$
3. **while** $E' \neq \emptyset$
   4. let $(u, v)$ be an arbitrary edge of $E'$
   5. $C = C \cup \{u, v\}$
   6. remove from $E'$ every edge incident on either $u$ or $v$
7. **return** $C$

III. Covering Problems

Vertex Cover
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. \(\text{while } E' \neq \emptyset\)
   4. \(\text{let } (u, v) \text{ be an arbitrary edge of } E'\)
   5. \(C = C \cup \{u, v\}\)
   6. \(\text{remove from } E' \text{ every edge incident on either } u \text{ or } v\)
4. \(\text{return } C\)

**Figure 35.1** Illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \(C\) contains the vertex cover being constructed. Line 1 initializes \(C\) to the empty set. Line 2 sets \(E'\) to be a copy of the edge set \(G.E\) of the graph. The loop of lines 3–6 repeatedly picks an edge \((u, v)\) from \(E'\), adds it to \(C\), and removes from \(E'\) every edge incident on either \(u\) or \(v\).
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. while \( E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
4. return \( C \)

III. Covering Problems

Vertex Cover
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G \cdot E\)
3. **while** \(E' \neq \emptyset\)
4. 
   let \((u, v)\) be an arbitrary edge of \(E'\)
5. \(C = C \cup \{u, v\}\)
6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. **return** \(C\)
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. while \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. return \(C\)

**III. Covering Problems**

**Vertex Cover**

**Figure 35.1** illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \(C\) contains the vertex cover being constructed. Line 1 initializes \(C\) to the empty set. Line 2 sets \(E'\) to be a copy of the edge set \(G.E\) of the graph. The loop of lines 3–6 repeatedly picks an edge \((u, v)\) from \(E'\), adds it to \(C\), and removes from \(E'\) every edge incident on either \(u\) or \(v\). Line 7 returns \(C\).
An Approximation Algorithm based on Greedy

**Algorithm:**

\text{APPROX-VERTEX-COVER}(G)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \textbf{while} \( E' \neq \emptyset \)
4. \hspace{1em} let \((u, v)\) be an arbitrary edge of \( E' \)
5. \hspace{1em} \( C = C \cup \{u, v\} \)
6. \hspace{1em} remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. \textbf{return} \( C \)

---

**Figure 35.1:**

The operation of \text{APPROX-VERTEX-COVER}.

(a) The input graph \( G \), which has 7 vertices and 8 edges.
(b) The edge \([b, c]\), shown heavy, is the first edge chosen by \text{APPROX-VERTEX-COVER}. Vertices \( b \) and \( c \), shown lightly shaded, are added to the set \( C \) containing the vertex cover being created.
(c) Edges \([a, b]\), \([c, e]\), and \([c, d]\), shown dashed, are removed since the year are covered by some vertex in \( C \).
(d) Edge \([e, f]\) is chosen; vertices \( e \) and \( f \) are added to \( C \).
(e) Edge \([d, g]\) is chosen; vertices \( d \) and \( g \) are added to \( C \).
(f) The vertex cover resulting from \text{APPROX-VERTEX-COVER} contains the vertices \( b, c, d, e, f, g \).

The optimal solution has size 3.
Analysis of Greedy for Vertex Cover

**Algorithm Approx-Vertex-Cover**($G$)

1. $C = \emptyset$
2. $E' = G.E$
3. **while** $E' \neq \emptyset$
4.   let $(u, v)$ be an arbitrary edge of $E'$
5.   $C = C \cup \{u, v\}$
6.   remove from $E'$ every edge incident on either $u$ or $v$
7. **return** $C$

---

**Theorem 35.1**

**Proof:**

Running time is $O(V + E)$ (using adjacency lists to represent $E'$).

Let $A \subseteq E$ denote the set of edges picked in line 4.

Key Observation: $A$ is a set of vertex-disjoint edges, i.e., $A$ is a matching.

⇒ Every optimal cover $C^*$ must include at least one endpoint:

$|C^*| \geq |A|$

Every edge in $A$ contributes 2 vertices to $|C|$:

$|C| = 2|A| \leq 2|C^*|$. 

We can bound the size of the returned solution without knowing the (size of an) optimal solution!

A "vertex-based" Greedy that adds one vertex at each iteration fails to achieve an approximation ratio of 2 (Supervision Exercise)!
Analysis of Greedy for Vertex Cover

**APPROX-VERTEX-COVER** (*G*)

1. $C = \emptyset$
2. $E' = G.E$
3. while $E' \neq \emptyset$
   4. let $(u, v)$ be an arbitrary edge of $E'$
   5. $C = C \cup \{u, v\}$
   6. remove from $E'$ every edge incident on either $u$ or $v$
7. return $C$

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.
Analysis of Greedy for Vertex Cover

\textsc{Approx-Vertex-Cover} (G)

1. \( C = \emptyset \)
2. \( E' = G . E \)
3. \textbf{while} \( E' \neq \emptyset \)
   4. \textbf{let} \((u, v)\) \textbf{be an arbitrary edge of }\( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. \textbf{remove} \textbf{from} \( E' \) \textbf{every} \textbf{edge} \textbf{incident} \textbf{on} \textbf{either} \( u \) \textbf{or} \( v \)
7. \textbf{return} \( C \)

\textbf{Theorem 35.1}

\textsc{Approx-Vertex-Cover} \textbf{is a} \textbf{poly-time} \textbf{2-approximation} \textbf{algorithm.}

\textbf{Proof:}
Analysis of Greedy for Vertex Cover

**Algorithm Approx-Vertex-Cover(G)**

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. \( \text{while } E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)

**Theorem 35.1**

**Approx-Vertex-Cover** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
Analysis of Greedy for Vertex Cover

**Algorithm**

\[
\textsc{Approx-Vertex-Cover}(G)
\]

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \( \textbf{while } E' \neq \emptyset \)
4. \( \text{let } (u, v) \text{ be an arbitrary edge of } E' \)
5. \( C = C \cup \{u, v\} \)
6. \( \text{remove from } E' \text{ every edge incident on either } u \text{ or } v \)
7. \( \textbf{return } C \)

---

**Theorem 35.1**

\textsc{Approx-Vertex-Cover} is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
**Analysis of Greedy for Vertex Cover**

**Algorithm APPROX-VERTEX-COVER**

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. while \( E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
4. return \( C \)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- **Key Observation:** \( A \) is a set of vertex-disjoint edges, i.e., \( A \) is a matching
Analysis of Greedy for Vertex Cover

**Algorithm**

\[\text{APPROX-VERTEX-COVER}(G)\]

1. \[C = \emptyset\]
2. \[E' = G \cdot E\]
3. \textbf{while} \(E' \neq \emptyset\) \textbf{do}
   4. \textbf{let} \((u, v)\) be an arbitrary edge of \(E'\)
   5. \[C = C \cup \{u, v\}\]
   6. \textbf{remove from} \(E'\) \textbf{every edge incident on either} \(u\) \textbf{or} \(v\)
4. \textbf{return} \(C\)

**Theorem 35.1**

\[\text{APPROX-VERTEX-COVER}\] is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- **Let** \(A \subseteq E\) denote the set of edges picked in line 4
- **Key Observation:** \(A\) is a set of vertex-disjoint edges, i.e., \(A\) is a matching

\[\Rightarrow\] **Every optimal cover** \(C^*\) **must include at least one endpoint:**
### Analysis of Greedy for Vertex Cover

**Algorithm 35.1**

\[
\text{APPROX-VERTEX-COVER}(G) \\
1 \quad C = \emptyset \\
2 \quad E' = G \cdot E \\
3 \quad \text{while } E' \neq \emptyset \\
4 \quad \quad \text{let } (u, v) \text{ be an arbitrary edge of } E' \\
5 \quad \quad C = C \cup \{u, v\} \\
6 \quad \quad \text{remove from } E' \text{ every edge incident on either } u \text{ or } v \\
7 \quad \text{return } C
\]

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- **Let** \(A \subseteq E\) **denote the set of edges picked in line 4**
- **Key Observation:** \(A\) is a set of vertex-disjoint edges, i.e., \(A\) is a matching
- \(\Rightarrow\) **Every optimal cover** \(C^*\) **must include at least one endpoint:**

\[|C^*| \geq |A|\]
Algorithm \textsc{Approx-Vertex-Cover}($G$)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \textbf{while} \( E' \neq \emptyset \)
4. \hspace{1em} let \((u, v)\) be an arbitrary edge of \( E' \)
5. \hspace{1em} \( C = C \cup \{u, v\} \)
6. \hspace{1em} remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. \textbf{return} \( C \)

\textbf{Theorem 35.1}

\textsc{Approx-Vertex-Cover} is a poly-time 2-approximation algorithm.

\textbf{Proof:}

- \textbf{Running time} is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- \textbf{Key Observation:} \( A \) is a set of vertex-disjoint edges, i.e., \( A \) is a matching
  \[ |C^*| \geq |A| \]
- \textbf{Every optimal cover} \( C^* \) must include at least one endpoint:
- \textbf{Every edge in} \( A \) contributes 2 vertices to \( |C| \):
Analysis of Greedy for Vertex Cover

**Algorithm 35.1  APPROX-VERTEX-COVER**

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. **while** \( E' \neq \emptyset \)
   4. let \( (u, v) \) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. **return** \( C \)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- **Key Observation:** \( A \) is a set of vertex-disjoint edges, i.e., \( A \) is a matching
  \( \Rightarrow \) Every optimal cover \( C^* \) must include at least one endpoint: \( |C^*| \geq |A| \)
- Every edge in \( A \) contributes 2 vertices to \( |C| \): \( |C| = 2|A| \)
Analysis of Greedy for Vertex Cover

**Algorithm**

\[
\text{APPROX-VERTEX-COVER}(G)
\]

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. \(\text{while } E' \neq \emptyset\)
   - let \((u, v)\) be an arbitrary edge of \(E'\)
   - \(C = C \cup \{u, v\}\)
   - remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. \(\text{return } C\)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- **Let** \(A \subseteq E\) **denote** the set of edges picked in line 4
- **Key Observation:** \(A\) is a set of vertex-disjoint edges, i.e., \(A\) is a **matching**
  
  \(\Rightarrow\) **Every optimal cover** \(C^*\) **must include** at least one endpoint: \(|C^*| \geq |A|\)
  
  - **Every edge in** \(A\) **contributes** 2 **vertices** to \(|C|\): \(|C| = 2|A| \leq 2|C^*|\).
Analysis of Greedy for Vertex Cover

**Algorithm 35.1: APPROX-VERTEX-COVER \((G)\)**

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. **while** \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. **return** \(C\)

---

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- Let \(A \subseteq E\) denote the set of edges picked in line 4
- **Key Observation:** \(A\) is a set of vertex-disjoint edges, i.e., \(A\) is a matching
  - **⇒** Every optimal cover \(C^*\) must include at least one endpoint: \(|C^*| \geq |A|\)
  - Every edge in \(A\) contributes 2 vertices to \(|C|\): \(|C| = 2|A| \leq 2|C^*|\)
Analysis of Greedy for Vertex Cover

\textsc{Approx-Vertex-Cover}(G)

1. \( C = \emptyset \)
2. \( E' = G. E \)
3. \textbf{while} \( E' \neq \emptyset \)
   - let \((u, v)\) be an arbitrary edge of \( E' \)
4. \( C = C \cup \{u, v\} \)
5. \textbf{remove from} \( E' \) \textbf{every edge incident on either} \( u \) \textbf{or} \( v \)
6. \textbf{return} \( C \)

\textbf{We can bound the size of the returned solution without knowing the (size of an) optimal solution!}

\textbf{Theorem 35.1}
\textsc{Approx-Vertex-Cover} is a poly-time 2-approximation algorithm.

\textbf{Proof:}

- \textbf{Running time} is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- \textbf{Key Observation:} \( A \) is a set of vertex-disjoint edges, i.e., \( A \) is a matching
  \( \Rightarrow \) Every optimal cover \( C^* \) must include at least one endpoint: \( |C^*| \geq |A| \)
- Every edge in \( A \) contributes 2 vertices to \( |C| \): \( |C| = 2|A| \leq 2|C^*| \). \( \square \)
**Analysis of Greedy for Vertex Cover**

`APPROX-VERTEX-COVER(G)`

1. `C = ∅`
2. `E' = G\cdot E`
3. while `E' ≠ ∅`
   4. let `(u, v)` be an arbitrary edge of `E'`
   5. `C = C ∪ {u, v}`
   6. remove from `E'` every edge incident on either `u` or `v`
4. return `C`

A "vertex-based" Greedy that adds **one** vertex at each iteration fails to achieve an approximation ratio of 2 (Supervision Exercise)!

We can bound the size of the returned solution **without knowing** the (size of an) optimal solution!

**Theorem 35.1**

`APPROX-VERTEX-COVER` is a poly-time **2-approximation** algorithm.

**Proof:**

- **Running time** is `O(V + E)` (using adjacency lists to represent `E'`)
- Let `A ⊆ E` denote the set of edges picked in line 4
- **Key Observation:** `A` is a set of vertex-disjoint edges, i.e., `A` is a matching
  ⇒ Every optimal cover `C*` must include at least one endpoint: `|C*| ≥ |A|`
- **Every edge in `A` contributes 2 vertices to `|C|`:**
  - `|C| = 2|A| ≤ 2|C*|`
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with \textit{exponential running time} may be satisfactory.
2. Isolate important \textit{special cases} which can be solved in polynomial-time.
3. Develop algorithms which find \textit{near-optimal} solutions in polynomial-time.
Solving Special Cases

1. If inputs are small, an algorithm with exponential running time may be satisfactory.

2. Isolate important special cases which can be solved in polynomial-time.

3. Develop algorithms which find near-optimal solutions in polynomial-time.

Strategies to cope with NP-complete problems

III. Covering Problems

Vertex Cover
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.
There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.
Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.

**Exchange-Argument:** Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.
There exists an optimal vertex cover which does not include any leaves.

**VERTEX-COVER-TREES(G)**

1: \( C = \emptyset \)
2: **while** \( \exists \) leaves in \( G \)
3: \quad Add all parents to \( C \)
4: \quad Remove all leaves and their parents from \( G \)
5: **return** \( C \)
Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

\[ \text{\textsc{vertex-cover-trees}(G)} \]
1: \( C = \emptyset \)
2: \textbf{while} \( \exists \) leaves in \( G \)
3: \quad \text{Add all parents to } C
4: \quad \text{Remove all leaves and their parents from } G
5: \quad \textbf{return} \ C

Clear: Running time is \( O(V) \), and the returned solution is a vertex cover.
Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

Algorithm: 

\textsc{Vertex-Cover-Trees}(G)

1: \( C = \emptyset \)
2: \textbf{while} \exists \text{ leaves in } G \\
3: \quad \text{Add all parents to } C \\
4: \quad \text{Remove all leaves and their parents from } G \\
5: \textbf{return} \ C \\

Clear: Running time is \( O(V) \), and the returned solution is a vertex cover.

Solution is also optimal. (Use inductively the existence of an optimal vertex cover without leaves)
Execution on a Small Example

\texttt{VERTEX-COVER-TREES(G)}

1: \hspace{0.5cm} \texttt{C} = \emptyset
2: \hspace{0.5cm} \textbf{while} \ \exists \text{ leaves in } G
3: \hspace{1cm} \text{Add all parents to } C
4: \hspace{1cm} \text{Remove all leaves and their parents from } G
5: \hspace{0.5cm} \textbf{return} \ C
Execution on a Small Example

```
VERTEX-COVER-TREES(G)
1: C = ∅
2: while ∃ leaves in G
3: Add all parents to C
4: Remove all leaves and their parents from G
5: return C
```

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.
**Execution on a Small Example**

\[ \text{VERTEX-COVER-TREES}(G) \]

1. \( C = \emptyset \)
2. \textbf{while} \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. \textbf{return} \( C \)
**Execution on a Small Example**

**VERTEX-COVER-TREES(G)**

1: \( C = \emptyset \)
2: **while** \( \exists \) leaves in \( G \)
3: Add all parents to \( C \)
4: Remove all leaves and their parents from \( G \)
5: **return** \( C \)
Execution on a Small Example

**Vertex-Cover-Trees(G)**

1: \( C = \emptyset \)
2: **while** \exists\ leaves in \( G \)
3: \hspace{1cm} Add all parents to \( C \)
4: \hspace{1cm} Remove all leaves and their parents from \( G \)
5: **return** \( C \)
**Execution on a Small Example**

\[ \text{VERTEX-COVER-TREES}(G) \]

1. \( C = \emptyset \)
2. \textbf{while} \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. \textbf{return} \( C \)
Execution on a Small Example

VERTEX-COVER-TREES(G)
1: $C = \emptyset$
2: while $\exists$ leaves in $G$
3: Add all parents to $C$
4: Remove all leaves and their parents from $G$
5: return $C$

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.

III. Covering Problems
Vertex Cover
Execution on a Small Example

\textbf{VERTEX-COVER-TREES}(G)

1: \( C = \emptyset \)
2: \textbf{while} \( \exists \) leaves in \( G \)
3: \hspace{1em} Add all parents to \( C \)
4: \hspace{1em} Remove all leaves and their parents from \( G \)
5: \hspace{1em} \textbf{return} \( C \)

Problem can be also solved on \textit{bipartite graphs}, using Max-Flows and Min-Cuts.
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory

2. Isolate important special cases which can be solved in polynomial-time.

3. Develop algorithms which find near-optimal solutions in polynomial-time.
Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Such algorithms are called exact algorithms.
Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Focus on instances where the minimum vertex cover is small, that is, less or equal than some given integer $k$. Such algorithms are called exact algorithms.
Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Such algorithms are called exact algorithms.

Focus on instances where the minimum vertex cover is small, that is, less or equal than some given integer $k$.

Simple Brute-Force Search would take $\approx \binom{n}{k} = \Theta(n^k)$ time.
Consider a graph \( G = (V, E) \), edge \( \{u, v\} \in E(G) \) and integer \( k \geq 1 \). Let \( G_u \) be the graph obtained by deleting \( u \) and its incident edges (\( G_v \) is defined similarly). Then \( G \) has a vertex cover of size \( k \) if and only if \( G_u \) or \( G_v \) (or both) have a vertex cover of size \( k - 1 \).
Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Reminiscent of Dynamic Programming.
Towards a more efficient Search

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\iff$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. 
Towards a more efficient Search

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\iff$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. 

\begin{enumerate}
\item[$\Leftarrow$] Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$.
Towards a more efficient Search

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$.
Towards a more efficient Search

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$.
Adding $u$ yields a vertex cover of $G$ which is of size $k$

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. 

---

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$.
Adding $u$ yields a vertex cover of $G$ which is of size $k$

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. 

---

III. Covering Problems

Vertex Cover
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$.

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. Removing $u$ from $C$ yields a vertex cover of $G_u$ which is of size $k - 1$. ⬜
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)

1. **if** \(E = \emptyset\) **return** \(\emptyset\)
2. **if** \(k = 0\) and \(E \neq \emptyset\) **return** \(\bot\)
3. Pick an arbitrary edge \((u, v) \in E\)
4. \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5. \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6. **if** \(S_1 \neq \bot\) **return** \(S_1 \cup \{u\}\)
7. **if** \(S_2 \neq \bot\) **return** \(S_2 \cup \{v\}\)
8. **return** \(\bot\)

Correctness follows by the Substructure Lemma and induction.

Running time:
- Depth \(k\), branching factor 2
- Total number of calls is \(O(2^k)\)
- Worst-case time for one call (computing \(G_u\) or \(G_v\) could take \(\Theta(E)\) !)

Total runtime:
- \(O(2^k \cdot E)\)
- exponential in \(k\), but much better than \(\Theta(n^k)\) (i.e., still polynomial for \(k = \mathcal{O}(\log n)\))
A More Efficient Search Algorithm

\[ \text{VERTEX-COVER-SEARCH}(G, k) \]
1: if \( E = \emptyset \) return \( \emptyset \)
2: if \( k = 0 \) and \( E \neq \emptyset \) return ⊥
3: Pick an arbitrary edge \((u, v) \in E\)
4: \( S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1) \)
5: \( S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1) \)
6: if \( S_1 \neq \perp \) return \( S_1 \cup \{u\} \)
7: if \( S_2 \neq \perp \) return \( S_2 \cup \{v\} \)
8: return ⊥

Correctness follows by the Substructure Lemma and induction.
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)
1. **if** \(E = \emptyset\) **return** \(\emptyset\)
2. **if** \(k = 0\) **and** \(E \neq \emptyset\) **return** \(\perp\)
3. **Pick** an arbitrary edge \((u, v) \in E\)
4. \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5. \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6. **if** \(S_1 \neq \perp\) **return** \(S_1 \cup \{u\}\)
7. **if** \(S_2 \neq \perp\) **return** \(S_2 \cup \{v\}\)
8. **return** \(\perp\)

**Running time:**

O\((2^k \cdot E)\) worst-case time for one call (computing \(G_u\) or \(G_v\) could take \(\Theta(E)\)!)
A More Efficient Search Algorithm

\[ \text{VERTEX-COVER-SEARCH}(G, k) \]

1: \textbf{if} \( E = \emptyset \) \textbf{return} \( \emptyset \)
2: \textbf{if} \( k = 0 \) \textbf{and} \( E \neq \emptyset \) \textbf{return} \( \bot \)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \( S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1) \)
5: \( S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1) \)
6: \textbf{if} \( S_1 \neq \bot \) \textbf{return} \( S_1 \cup \{u\} \)
7: \textbf{if} \( S_2 \neq \bot \) \textbf{return} \( S_2 \cup \{v\} \)
8: \textbf{return} \( \bot \)

Running time:
- Depth \( k \), branching factor 2

III. Covering Problems

Vertex Cover
A More Efficient Search Algorithm

\[ \text{VERTEX-COVER-SEARCH}(G, k) \]

1: \textbf{if} \( E = \emptyset \) \textbf{return} \( \emptyset \)
2: \textbf{if} \( k = 0 \) and \( E \neq \emptyset \) \textbf{return} \( \perp \)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \( S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1) \)
5: \( S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1) \)
6: \textbf{if} \( S_1 \neq \perp \) \textbf{return} \( S_1 \cup \{u\} \)
7: \textbf{if} \( S_2 \neq \perp \) \textbf{return} \( S_2 \cup \{v\} \)
8: \textbf{return} \( \perp \)

Running time:
- Depth \( k \), branching factor 2 \( \Rightarrow \) total number of calls is \( O(2^k) \)
A More Efficient Search Algorithm

VERTEX-COVER-SEARCH\((G, k)\)

1: if \(E = \emptyset\) return \(\emptyset\)
2: if \(k = 0\) and \(E \neq \emptyset\) return \(\bot\)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5: \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6: if \(S_1 \neq \bot\) return \(S_1 \cup \{u\}\)
7: if \(S_2 \neq \bot\) return \(S_2 \cup \{v\}\)
8: return \(\bot\)

Running time:

- Depth \(k\), branching factor 2 \(\Rightarrow\) total number of calls is \(O(2^k)\)
- \(O(E)\) worst-case time for one call (computing \(G_u\) or \(G_v\) could take \(\Theta(E)\)!)

III. Covering Problems

Vertex Cover
A More Efficient Search Algorithm

\textsc{Vertex-Cover-Search}(G, k)
1: \textbf{if} \( E = \emptyset \) \textbf{return} \( \emptyset \)
2: \textbf{if} \( k = 0 \) and \( E \neq \emptyset \) \textbf{return} \( \perp \)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \( S_1 = \textsc{Vertex-Cover-Search}(G_u, k - 1) \)
5: \( S_2 = \textsc{Vertex-Cover-Search}(G_v, k - 1) \)
6: \textbf{if} \( S_1 \neq \perp \) \textbf{return} \( S_1 \cup \{u\} \)
7: \textbf{if} \( S_2 \neq \perp \) \textbf{return} \( S_2 \cup \{v\} \)
8: \textbf{return} \( \perp \)

Running time:
- Depth \( k \), branching factor 2 \( \Rightarrow \) total number of calls is \( O(2^k) \)
- \( O(E) \) worst-case time for one call (computing \( G_u \) or \( G_v \) could take \( \Theta(E) \)!) 
- Total runtime: \( O(2^k \cdot E) \).
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)

1. if \(E = \emptyset\) return \(\emptyset\)
2. if \(k = 0\) and \(E \neq \emptyset\) return \(\bot\)
3. Pick an arbitrary edge \((u, v) \in E\)
4. \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5. \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6. if \(S_1 \neq \bot\) return \(S_1 \cup \{u\}\)
7. if \(S_2 \neq \bot\) return \(S_2 \cup \{v\}\)
8. return \(\bot\)

Running time:

- Depth \(k\), branching factor 2 ⇒ total number of calls is \(O(2^k)\)
- \(O(E)\) worst-case time for one call (computing \(G_u\) or \(G_v\) could take \(\Theta(E)\)!) 
- Total runtime: \(O(2^k \cdot E)\).

exponential in \(k\), but much better than \(\Theta(n^k)\) (i.e., still polynomial for \(k = O(\log n)\))
Outline

Introduction

Vertex Cover

The Set-Covering Problem
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

$$\text{s.t. } X = \bigcup_{S \in C} S.$$
The Set-Covering Problem

Set Cover Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[
\text{s.t. } X = \bigcup_{S \in C} S.
\]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

\[
X = \bigcup_{S \in C} S.
\]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!
The Set-Covering Problem

Given: set \( X \) of size \( n \) and family of subsets \( \mathcal{F} \)

Goal: Find a minimum-size subset \( C \subseteq \mathcal{F} \)

\[
\text{s.t.} \quad X = \bigcup_{S \in C} S.
\]

Only solvable if \( \bigcup_{S \in \mathcal{F}} S = X \! \)

Remarks:
- Generalisation of the vertex-cover problem and hence also NP-hard.
- Models resource allocation problems, e.g., wireless coverage.

Set Cover Problem

- Given: set \( X \) of size \( n \) and family of subsets \( \mathcal{F} \)
- Goal: Find a minimum-size subset \( C \subseteq \mathcal{F} \)

Number of sets (and not elements)
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in \mathcal{C}} S$.

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in \mathcal{C}} S$

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!

Remarks:
- Generalization of the vertex-cover problem and hence also NP-hard.
- Models resource allocation problems, e.g., wireless coverage.
The Set-Covering Problem

Given: set $X$ of size $n$ and family of subsets $\mathcal{F}$

Goal: Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in \mathcal{C}} S.$

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X!$

Remarks:
- Generalisation of the vertex-cover problem and hence also NP-hard.
- Models resource allocation problems, e.g., wireless coverage.
The Set-Covering Problem

Given: set $X$ of size $n$ and family of subsets $\mathcal{F}$

Goal: Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in \mathcal{C}} S$

Number of sets (and not elements)

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X!$
The Set-Covering Problem

Given: set $X$ of size $n$ and family of subsets $\mathcal{F}$

Goal: Find a minimum-size subset $C \subseteq \mathcal{F}$

such that $X = \bigcup_{S \in C} S$.

Number of sets (and not elements)

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!

Remarks: generalisation of the vertex-cover problem and hence also NP-hard. Models resource allocation problems, e.g., wireless coverage.

III. Covering Problems

The Set-Covering Problem
The Set-Covering Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

\[
\text{s.t.} \quad X = \bigcup_{S \in C} S.
\]

Number of sets (and not elements)

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!
The Set-Covering Problem

Set Cover Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[
X = \bigcup_{S \in C} S.
\]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!

Remarks:
The Set-Covering Problem

**Set Cover Problem**

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in C} S. \]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!

**Remarks:**
- generalisation of the vertex-cover problem and hence also NP-hard.
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

\[
\text{s.t. } X = \bigcup_{S \in \mathcal{C}} S.
\]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X!$

Remarks:

- generalisation of the vertex-cover problem and hence also NP-hard.
- models resource allocation problems, e.g., wireless coverage
Greedy

Strategy: Pick the set $S$ that covers the largest number of uncovered elements.

III. Covering Problems The Set-Covering Problem 19
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**GREEDY-SET-COVER** ($X$, $F$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in F$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$

---

III. Covering Problems

The Set-Covering Problem
Greedy

Strategy: Pick the set \( S \) that covers the largest number of uncovered elements.

**Greedy-Set-Cover** \((X, \mathcal{F})\)

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. \textbf{while} \( U \neq \emptyset \)
4. \hspace{1em} select an \( S \in \mathcal{F} \) that maximizes \( |S \cap U| \)
5. \hspace{1em} \( U = U - S \)
6. \hspace{1em} \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
7. \textbf{return} \( \mathcal{C} \)
Greedy

**Strategic: Pick the set $S$ that covers the largest number of uncovered elements.**

**Greedy-Set-Cover**($X$, $F$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in F$ that maximizes $|S \cap U|$  
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$
Greedy

**Strategy:** Pick the set \( S \) that covers the largest number of uncovered elements.

**Greedy-Set-Cover** \((X, \mathcal{F})\)

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. \( \textbf{while } U \neq \emptyset \)
4. \( \text{select an } S \in \mathcal{F} \text{ that maximizes } |S \cap U| \)
5. \( U = U - S \)
6. \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
7. \( \textbf{return } \mathcal{C} \)

In the example of Figure 35.3, Greedy-Set-Cover adds, in order, the sets \( S_1, S_4, \) and \( S_5 \), followed by either \( S_3 \) or \( S_6 \).

The algorithm works as follows. The set \( U \) contains, at each stage, the set of remaining uncovered elements. The set \( \mathcal{C} \) contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset \( S \) that covers as many uncovered elements as possible (breaking ties arbitrarily). After \( S \) is selected, line 5 removes its elements from \( U \), and line 6 places \( S \) into \( \mathcal{C} \). When the algorithm terminates, the set \( \mathcal{C} \) contains a subfamily of \( \mathcal{F} \) that covers \( X \).

We can easily implement Greedy-Set-Cover to run in time polynomial in \(|X|\) and \(|\mathcal{F}|\). Since the number of iterations of the loop in lines 3–6 is bounded from above by \( \min\{|X|, |\mathcal{F}|\} \), a simple implementation runs in time \( O(|X| |\mathcal{F}|) \), as a simple implementation runs in time \( O(|X| |\mathcal{F}| \min\{|X|, |\mathcal{F}|\}) \).

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the \( d \text{th} \) harmonic number \( H_d \) (see Section A.1) by \( H(d) \). As a boundary condition, we define \( H(0) = 0 \).

**Theorem 35.4**

Greedy-Set-Cover is a polynomial-time \( \mathcal{O}(n) \)-approximation algorithm, where \( /\mathcal{O}(n) = H(\max\{|S| \mid S \in \mathcal{F} \}) \):

**Proof**

We have already shown that Greedy-Set-Cover runs in polynomial time.
**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, \mathcal{F})$

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$
Greedy

Strategy: Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, \mathcal{F})$  
1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
   4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
   5. $U = U - S$
   6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$
Greedy

Strategy: Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, \mathcal{F})$

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$ 
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

Optimal cover is $C = \{S_3, S_4, S_5\}$

Greedy chooses $S_1, S_4, S_5$ and $S_3$ (or $S_6$), which is a cover of size 4.
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, F)$

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in F$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$

In the example of Figure 35.3, Greedy-Set-Cover adds to $\mathcal{C}$, in order, the sets $S_1$, $S_4$, and $S_5$, followed by either $S_3$ or $S_6$. The algorithm works as follows. The set $U$ contains, at each stage, the set of remaining uncovered elements. The set $\mathcal{C}$ contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset $S$ that covers as many uncovered elements as possible (breaking ties arbitrarily). After $S$ is selected, line 5 removes its elements from $U$, and line 6 places $S$ into $\mathcal{C}$. When the algorithm terminates, the set $\mathcal{C}$ contains a subfamily of $F$ that covers $X$. We can easily implement Greedy-Set-Cover to run in time polynomial in $|X|$ and $|F|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min \{ |X|, |F| \}$, we can implement the loop in time $O(\min \{ |X|, |F| \})$.

**Exercise 35.3-3** asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the $d$th harmonic number $H_d = \sum_{i=1}^{d} \frac{1}{i}$ (see Section A.1) by $H(d)$. As a boundary condition, we define $H(0) = 0$.

**Theorem 35.4**

Greedy-Set-Cover is a polynomial-time $\mathcal{O}(n)$-approximation algorithm, where $\mathcal{O}(n) = H(\max \{ |S| : W \subseteq F \})$.

**Proof**

We have already shown that Greedy-Set-Cover runs in polynomial time.
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, F)$

1. $U = X$
2. $C = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in F$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $C = C \cup \{S\}$
7. **return** $C$

Can be easily implemented to run in time polynomial in $|X|$ and $|F|$.
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** ($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

Can be easily implemented to run in time polynomial in $|X|$ and $|\mathcal{F}|$

How good is the approximation ratio?

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\mathcal{O}(\log |X|)$-approximation algorithm, where

$\mathcal{O}(\log |X|) = H_{\max \{ |S| \mid S \in \mathcal{F} \}}$

**Proof**

We have already shown that **GREEDY-SET-COVER** runs in polynomial time.

### Example

In the example of Figure 35.3, **GREEDY-SET-COVER** adds, in order, the sets $S_1$, $S_4$, and $S_5$, followed by either $S_3$ or $S_6$.

The algorithm works as follows. The set $U$ contains, at each stage, the set of remaining uncovered elements. The set $\mathcal{C}$ contains the cover being constructed.

Line 4 is the greedy decision-making step, choosing a subset $S$ that covers as many uncovered elements as possible (breaking ties arbitrarily). After $S$ is selected, line 5 removes its elements from $U$, and line 6 places $S$ into $\mathcal{C}$.

When the algorithm terminates, the set $\mathcal{C}$ contains a subfamily of $\mathcal{F}$ that covers $X$.

We can easily implement **GREEDY-SET-COVER** to run in time polynomial in $|X|$ and $|\mathcal{F}|$.

Since the number of iterations on lines 3–6 is bounded from above by $\min \{ |X|, |\mathcal{F}| \}$, and we can implement the loop in time $O(|X| |\mathcal{F}| \min \{ |X|, |\mathcal{F}| \})$.

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the $d$th harmonic number $H_d$ (see Section A.1) by $H_d$. As a boundary condition, we define $H_{0} = 0$.

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\mathcal{O}(\log |X|)$-approximation algorithm, where

$\mathcal{O}(\log |X|) = H_{\max \{ |S| \mid S \in \mathcal{F} \}}$

**Proof**

We have already shown that **GREEDY-SET-COVER** runs in polynomial time.
Theorem 35.4

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$$
Approximation Ratio of Greedy

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$$

**Idea:** Distribute cost of 1 for each added set over newly covered elements.

If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.

**Definition of cost**

Notice that in the mathematical analysis, $S_i$ is the set chosen in iteration $i$—not to be confused with the sets $S_1, S_2, \ldots, S_6$ in the example.

III. Covering Problems

The Set-Covering Problem
Approximation Ratio of Greedy

**Theorem 35.4**

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$  

**Idea:** Distribute the cost of 1 for each added set over newly covered elements. If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

**Definition of cost**

Notice that in the mathematical analysis, $S_i$ is the set chosen in iteration $i$ — not to be confused with the sets $S_1, S_2, ..., S_6$ in the example.

III. Covering Problems

The Set-Covering Problem
**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in F\}) \leq \ln(n) + 1.$$ 

Idea: Distribute cost of 1 for each added set over newly covered elements.
Approximation Ratio of Greedy

Theorem 35.4

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$ 

$$H(k) := \sum_{i=1}^{k} \frac{1}{i} \leq \ln(k) + 1$$

Idea: Distribute cost of 1 for each added set over newly covered elements.

Definition of cost

If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then

$$c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$
Approximation Ratio of Greedy

**Theorem 35.4**

**Greedy-Set-Cover** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$ 

**Idea:** Distribute cost of 1 for each added set over newly covered elements.

**Definition of cost**

If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then

$$c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$ 

Notice that in the mathematical analysis, $S_i$ is the set chosen in iteration $i$ - not to be confused with the sets $S_1, S_2, \ldots, S_6$ in the example.
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$

III. Covering Problems

The Set-Covering Problem
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$

III. Covering Problems

The Set-Covering Problem
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$

III. Covering Problems

The Set-Covering Problem
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i\setminus(S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$. 

Proof of Theorem 35.4 (1/2)
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$

Proof.

- Each step of the algorithm assigns one unit of cost, so

\begin{equation}
(1)
\end{equation}
Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x$$  \hspace{1cm} (1)
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \tag{1}
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|)
\]

\[
\leq |C^*| \cdot H(\max\{|S|: S \in F\})
\]

Key Inequality:

\[
\sum_{x \in S} c_x \leq H(|S|).
\]
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so
  \[
  |C| = \sum_{x \in X} c_x \quad (1)
  \]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so
  \[
  \sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)
  \]
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x$$ (1)

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$$ (2)

- Combining 1 and 2 gives
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \quad (1)
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)
\]

- Combining 1 and 2 gives

\[
|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x
\]
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \tag{1}
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \tag{2}
\]

- Combining 1 and 2 gives

\[
|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x
\]

Key Inequality: \( \sum_{x \in S} c_x \leq H(|S|) \).
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x \quad (1)$$

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)$$

- Combining 1 and 2 gives

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|)$$

Key Inequality: \(\sum_{x \in S} c_x \leq H(|S|)\).
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}.$

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x \quad (1)$$

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)$$

- Combining 1 and 2 gives

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\})$$

Key Inequality: $\sum_{x \in S} c_x \leq H(|S|).$
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

For any \( S \in F \) and \( i = 1, 2, \ldots, |C| = k \) let

\[ u_i = |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \]

\[ u_0 \geq u_1 \geq \cdots \geq u_k = 0 \]

and \( u_i - u_{i-1} \) counts the items in \( S \) covered first time by \( S_i \).

\[ \Rightarrow \sum_{x \in S} c_x \leq k \sum_{i=1}^{|C|} (u_i - u_{i-1}) \cdot 1 \]

Further, by definition of the \( \text{G\_REEDY\_SET\_COVER} \):

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_i - u_{i-1} \cdot 1 \]

Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq k \sum_{i=1}^{|C|} (u_i - u_{i-1}) \cdot 1 \]

\[ = k \sum_{i=1}^{|C|} u_i - u_{i-1} \sum_{j=1}^{u_i} 1 \]

\[ \leq k \sum_{i=1}^{|C|} (H(u_i - u_{i-1}) - H(u_i)) \]

\[ = H(u_0) - H(u_k) \]

\[ = H(|S|) \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality
\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

Remaining uncovered elements in \( S \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

Sets chosen by the algorithm

III. Covering Problems The Set-Covering Problem 23
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

Sets chosen by the algorithm

Remaining uncovered elements in \( S \) Sets chosen by the algorithm

III. Covering Problems The Set-Covering Problem 23
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]

\[ \Rightarrow \sum_{x \in S} c_x \leq k \sum_{i=1}^{u_0} (u_i - 1) \sum_{j=1}^{u_i} 1 \leq k \sum_{i=1}^{u_0} u_i - H(u_k) = H(|S|). \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \]

and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]

\[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \]

and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the \textbf{GREEDY-SET-COVER}:

III. Covering Problems

The Set-Covering Problem
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \(\Rightarrow\) \( |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).
  \(\Rightarrow\)
  \[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \( \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

  \[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:
  \[ |S_{i} \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \]
  and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

  \[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:
  \[ \sum_{x \in S} c_x \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ \Rightarrow |S| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \quad \text{and} \quad u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]
  \[ \Rightarrow \]
  \[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:
  \[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \quad \text{and} \quad u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \( \Rightarrow \) \( |S| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).
  \( \Rightarrow \)

\[
\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}
\]

- Further, by definition of the GREEDY-SET-COVER:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:
  \[
  \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}
  \]
  \[
  \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j}
  \]
  \[= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \]

III. Covering Problems

The Set-Covering Problem 23
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the GREEDY-SET-COVER:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \]

\[ \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \]

\[ = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i \colonequals |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |S| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \]

\[ \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \]

\[ = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) = H(|S|). \]

\[ \square \]
Set-Covering Problem (Summary)

Theorem 35.4

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$
The same approach also gives an approximation ratio of $O(\ln(n))$ if there exists a cost function $c : \mathcal{F} \rightarrow \mathbb{R}^+$.

**Theorem 35.4**

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$
Theorem 35.4

**Greedy-Set-Cover** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.$$

---

- Is the bound on the approximation ratio in Theorem 35.4 tight?
- Is there a better algorithm?
**Theorem 35.4**

GREEDY-SET-COVER is a polynomial-time \( \rho(n) \)-algorithm, where

\[
\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.
\]

**Is the bound on the approximation ratio in Theorem 35.4 tight?**

**Is there a better algorithm?**

**Lower Bound**

Unless P=NP, there is no \( c \cdot \ln(n) \) polynomial-time approximation algorithm for some constant \( 0 < c < 1 \).
Example where the solution of Greedy is bad

Given any integer $k \geq 3$

<table>
<thead>
<tr>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Given any integer $k \geq 3$</td>
</tr>
</tbody>
</table>
## Example where the solution of Greedy is bad

<table>
<thead>
<tr>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Given any integer $k \geq 3$</td>
</tr>
<tr>
<td>- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)</td>
</tr>
</tbody>
</table>
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)

\[ k = 4, \quad n = 30: \]

\[ \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \]

\[ \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \]
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30$:
Example where the solution of Greedy is bad

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30$:
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30:$

III. Covering Problems

The Set-Covering Problem
Example where the solution of Greedy is bad

**Instance**

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements

\[ k = 4, n = 30: \]

\[ S_1 \quad \quad S_2 \quad \quad S_3 \]

III. Covering Problems

The Set-Covering Problem
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements

\( k = 4, n = 30: \)

III. Covering Problems
The Set-Covering Problem
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

\[
\begin{align*}
S_1 & \quad S_2 & \quad S_3 & \quad S_4 \\
\text{(Sample Sets)} & \quad \text{(Sample Sets)} & \quad \text{(Sample Sets)} & \quad \text{(Sample Sets)}
\end{align*}
\]
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:
Example where the solution of Greedy is bad

- **Instance**
  - Given any integer $k \geq 3$
  - There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
  - Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
  - Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:
Example where the solution of Greedy is bad

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[ k = 4, \; n = 30: \]

\[
\begin{array}{cccc}
S_1 & S_2 & S_3 & S_4 \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

\[
\begin{array}{cccc}
T_1 & T_2 \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$: 

III. Covering Problems

The Set-Covering Problem
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30:$
Example where the solution of Greedy is bad

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

Solution of **Greedy** consists of $k$ sets.
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

Solution of Greedy consists of $k$ sets.
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30:$

Solution of Greedy consists of $k$ sets.
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[
k = 4, \quad n = 30:
\]

Solution of Greedy consists of \( k \) sets.
Example where the solution of Greedy is bad

**Instance**

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30:$

Solution of **Greedy** consists of $k$ sets.  
Optimum consists of 2 sets.
Exercise: Consider the vertex cover problem, restricted to a graph where every vertex has exactly 3 neighbours. Which approximation ratio can we obtain?

1. 1 (i.e., I can solve it exactly!!!)
2. 2
3. \( \frac{11}{6} = 2 - \frac{1}{6} \)
4. \( H(n) \leq \log(n) \)
IV. Approximation Algorithms via Exact Algorithms

Thomas Sauerwald

Easter 2021
The Subset-Sum Problem

Parallel Machine Scheduling

Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).
The Subset-Sum Problem

- **Given:** Set of positive integers $S = \{x_1, x_2, \ldots, x_n\}$ and positive integer $t$
- **Goal:** Find a subset $S' \subseteq S$ which maximizes $\sum_{i: x_i \in S'} x_i \leq t$.

This problem is **NP-hard**
The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

\[
\begin{align*}
  x_1 &= 10 \\
  x_2 &= 4 \\
  x_3 &= 5 \\
  x_4 &= 6 \\
  x_5 &= 1 \\
  t &= 13 \text{ tons}
\end{align*}
\]
The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

\[ x_1 = 10 \]
\[ x_2 = 4 \]
\[ x_3 = 5 \]
\[ x_4 = 6 \]
\[ x_5 = 1 \]

\( t = 13 \) tons
The Subset-Sum Problem

- **Given:** Set of positive integers $S = \{x_1, x_2, \ldots, x_n\}$ and positive integer $t$
- **Goal:** Find a subset $S' \subseteq S$ which maximizes $\sum_{i: x_i \in S'} x_i \leq t$.

### Example
- $x_1 = 10$
- $x_2 = 4$
- $x_3 = 5$
- $x_4 = 6$
- $x_5 = 1$

$t = 13$ tons
The Subset-Sum Problem

Given: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
Goal: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

\[
\begin{align*}
\hspace{1cm} t &= 13 \text{ tons} \\
\hspace{1cm} x_1 &= 10 \\
\hspace{1cm} x_2 &= 4 \\
\hspace{1cm} x_3 &= 5 \\
\hspace{1cm} x_4 &= 6 \\
\hspace{1cm} x_5 &= 1 \\
\hspace{1cm} x_1 + x_5 &= 11
\end{align*}
\]
The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

\[
\begin{align*}
x_1 &= 10 \\
x_2 &= 4 \\
x_3 &= 5 \\
x_4 &= 6 \\
x_5 &= 1 \\
\end{align*}
\]

\( t = 13 \text{ tons} \)
The Subset-Sum Problem

Given: Set of positive integers \( S = \{ x_1, x_2, \ldots, x_n \} \) and positive integer \( t \)

Goal: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).
Dynamic Programming: Compute bottom-up all possible sums $\leq t$
Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum($S, t$)**

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
   4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
   5. remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$
**An Exact (Exponential-Time) Algorithm**

### Dynamic Programming: Compute bottom-up all possible sums \( \leq t \)

**Exact-Subset-Sum** \((S, t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
   4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \quad S + x := \{s + x : s \in S\} \)
   5. remove from \( L_i \) every element that is greater than \( t \)
4. **return** the largest element in \( L_n \)
Exact-Subset-Sum($S, t$)

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
4.   $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
5.     $S + x := \{s + x : s \in S\}$
6.     remove from $L_i$ every element that is greater than $t$
7. return the largest element in $L_n$

Dynamic Programming: Compute bottom-up all possible sums $\leq t$
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum** $(S, t)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
   4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
   5. remove from $L_i$ every element that is greater than $t$
4. return the largest element in $L_n$

**Returns the merged list (in sorted order and without duplicates)**

Implementable in time $O(|L_{i-1}|)$ (like Merge-Sort)
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum** ($S, t$)

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
4. \hspace{1em} $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
5. \hspace{1em} remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$

Example:
**An Exact (Exponential-Time) Algorithm**

**Dynamic Programming:** Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum** $(S, t)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
   4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
   5. remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$

**Example:**
- $S = \{1, 4, 5\}$, $t = 10$

---

IV. Approximation via Exact Algorithms

The Subset-Sum Problem
An Exact (Exponential-Time) Algorithm

### Dynamic Programming: Compute bottom-up all possible sums ≤ t

**Exact-Subset-Sum(S, t)**

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
   4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
   5. remove from \( L_i \) every element that is greater than \( t \)
6. **return** the largest element in \( L_n \)

**Example:**
- \( S = \{1, 4, 5\} \), \( t = 10 \)
- \( L_0 = \langle 0 \rangle \)
Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum** $(S, t)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. **for** $i = 1$ **to** $n$
4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
5. **remove** from $L_i$ every element that is greater than $t$
6. **return** the largest element in $L_n$

Example:
- $S = \{1, 4, 5\}$, $t = 10$
- $L_0 = \langle 0 \rangle$
- $L_1 = \langle 0, 1 \rangle$
Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum** $(S, t)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
4. \hspace{1em} $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
5. \hspace{1em} remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$

Example:
- $S = \{1, 4, 5\}$, $t = 10$
- $L_0 = \langle 0 \rangle$
- $L_1 = \langle 0, 1 \rangle$
- $L_2 = \langle 0, 1, 4, 5 \rangle$
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums ≤ t

**Exact-Subset-Sum** \((S, t)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. for \(i = 1\) to \(n\)
4. \(L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)\)
5. remove from \(L_i\) every element that is greater than \(t\)
6. return the largest element in \(L_n\)

Example:
- \(S = \{1, 4, 5\}, t = 10\)
- \(L_0 = \langle 0 \rangle\)
- \(L_1 = \langle 0, 1 \rangle\)
- \(L_2 = \langle 0, 1, 4, 5 \rangle\)
- \(L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle\)
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums ≤ \( t \)

**Exact-Subset-Sum** \((S, t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. for \( i = 1 \) to \( n \)
   4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
5. remove from \( L_i \) every element that is greater than \( t \)
6. return the largest element in \( L_n \)

Example:
- \( S = \{1, 4, 5\}, \ t = 10 \)
- \( L_0 = \langle 0 \rangle \)
- \( L_1 = \langle 0, 1 \rangle \)
- \( L_2 = \langle 0, 1, 4, 5 \rangle \)
- \( L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle \)
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums \( \leq t \)

**Exact-Subset-Sum** \((S, t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
   4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
   5. remove from \( L_i \) every element that is greater than \( t \)
6. **return** the largest element in \( L_n \)

**Correctness:** \( L_n \) contains all sums of \( \{x_1, x_2, \ldots, x_n\} \)

Example:

- \( S = \{1, 4, 5\}, \ t = 10 \)
- \( L_0 = \langle 0 \rangle \)
- \( L_1 = \langle 0, 1 \rangle \)
- \( L_2 = \langle 0, 1, 4, 5 \rangle \)
- \( L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle \)
An Exact (Exponential-Time) Algorithm

**Dynamic Programming:** Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum**($S, t$)

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. **for** $i = 1$ **to** $n$
   4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1}, L_i \cup x_i)$
5. remove from $L_i$ every element that exceeds $t$
6. **return** the largest element in $L_n$

- **Correctness:** $L_n$ contains all sums of $\{x_1, x_2, \ldots, x_n\}$

Example:
- $S = \{1, 4, 5\}, t = 10$
- $L_0 = \langle 0 \rangle$
- $L_1 = \langle 0, 1 \rangle$
- $L_2 = \langle 0, 1, 4, 5 \rangle$
- $L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle$
**An Exact (Exponential-Time) Algorithm**

**Dynamic Programming:** Compute bottom-up all possible sums \( \leq t \)

\[
\text{EXACT-SUBSET-SUM}(S, t)
\]

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. \( \text{for } i = 1 \text{ to } n \)
   4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
5. remove from \( L_i \) every element that is greater than \( t \)
6. \( \text{return} \) the largest element in \( L_n \)

- **Correctness:** \( L_n \) contains all sums of \( \{x_1, x_2, \ldots, x_n\} \)
- **Runtime:** \( O(2^1 + 2^2 + \cdots + 2^n) = O(2^n) \)

**Example:**
- \( S = \{1, 4, 5\}, \ t = 10 \)
- \( L_0 = \langle 0 \rangle \)
- \( L_1 = \langle 0, 1 \rangle \)
- \( L_2 = \langle 0, 1, 4, 5 \rangle \)
- \( L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle \)
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums \( \leq t \)

**Exact-Subset-Sum** \((S, t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
5. remove from \( L_i \) every element that is greater than \( t \)
6. **return** the largest element in \( L_n \)

- **Correctness:** \( L_n \) contains all sums of \( \{x_1, x_2, \ldots, x_n\} \)
- **Runtime:** \( O(2^1 + 2^2 + \cdots + 2^n) = O(2^n) \)

**Example:**
- \( S = \{1, 4, 5\} \)
- \( L_0 = \langle 0 \rangle \)
- \( L_1 = \langle 0, 1 \rangle \)
- \( L_2 = \langle 0, 1, 4, 5 \rangle \)
- \( L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle \)

There are \( 2^i \) subsets of \( \{x_1, x_2, \ldots, x_i\} \).
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums ≤ t

**Exact-Subset-Sum** \((S, t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
4. \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
5. remove from \( L_i \) every element that is greater than \( t \)
6. **return** the largest element in \( L_n \)

**Example:**
- \( S = \{1, 4, 5\} \)
- \( L_0 = \langle 0 \rangle \)
- \( L_1 = \langle 0, 1 \rangle \)
- \( L_2 = \langle 0, 1, 4, 5 \rangle \)
- \( L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle \)

**Correctness:** \( L_n \) contains all sums of \( \{x_1, x_2, \ldots, x_n\} \)

**Runtime:** \( O(2^1 + 2^2 + \cdots + 2^n) = O(2^n) \)

There are \( 2^i \) subsets of \( \{x_1, x_2, \ldots, x_i\} \).

Better runtime if \( t \) and/or \( |L_i| \) are small.
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

Trimming a List

- Given a trimming parameter $0 < \delta < 1$
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

Trimming a List

- Given a trimming parameter $0 < \delta < 1$
- Trimming $L$ yields smaller sublist $L'$ so that for every $y \in L$: $\exists z \in L'$:

$$\frac{y}{1 + \delta} \leq z \leq y.$$
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

**Trimming a List**

- Given a **trimming parameter** $0 < \delta < 1$
- **Trimming** $L$ yields **smaller** sublist $L'$ so that for every $y \in L$: $\exists z \in L'$:
  \[
  \frac{y}{1 + \delta} \leq z \leq y.
  \]

**Example:**

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

TRIM works in time $\Theta(m)$, if $L$ is given in sorted order.

IV. Approximation via Exact Algorithms

The Subset-Sum Problem
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

Trimming a List

- Given a trimming parameter $0 < \delta < 1$
-Trimming $L$ yields smaller sublist $L'$ so that for every $y \in L$: $\exists z \in L'$:

$$\frac{y}{1 + \delta} \leq z \leq y.$$ 

- $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$
- $\delta = 0.1$
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

Trimming a List

- Given a trimming parameter $0 < \delta < 1$
- Trimming $L$ yields smaller sublist $L'$ so that for every $y \in L$: $\exists z \in L'$:

\[
\frac{y}{1 + \delta} \leq z \leq y.
\]

- $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$
- $\delta = 0.1$
- $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$

Trimming a List

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

$\delta = 0.1$

$L' = \langle 10, 12, 15, 20, 23, 29 \rangle$
Towards a FPTAS

Idea: Don’t need to maintain two values in $L$ which are close to each other.

Trimming a List

- Given a trimming parameter $0 < \delta < 1$
- Trimming $L$ yields smaller sublist $L'$ so that for every $y \in L$: $\exists z \in L'$:

$$\frac{y}{1 + \delta} \leq z \leq y.$$ 

$\text{TRIM}(L, \delta)$

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. $\text{last} = y_1$
4. for $i = 2$ to $m$
5. \hspace{1em} if $y_i > \text{last} \cdot (1 + \delta)$ \hspace{1em} \Comment{y_i \geq \text{last} because L is sorted}
6. \hspace{1em} append $y_i$ onto the end of $L'$
7. \hspace{1em} $\text{last} = y_i$
8. return $L'$
**Towards a FPTAS**

**Idea:** Don’t need to maintain two values in \( L \) which are close to each other.

**Trimming a List**

- Given a trimming parameter \( 0 < \delta < 1 \)
- Trimming \( L \) yields smaller sublist \( L' \) so that for every \( y \in L \):
  \[
  \frac{y}{1 + \delta} \leq z \leq y.
  \]

**TRIM\((L, \delta)\)**

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( last = y_1 \)
4. for \( i = 2 \) to \( m \)
   5. if \( y_i > last \cdot (1 + \delta) \) \( \text{ // } y_i \geq last \) because \( L \) is sorted
   6. append \( y_i \) onto the end of \( L' \)
   7. \( last = y_i \)
5. return \( L' \)

**TRIM works in time \( \Theta(m) \), if \( L \) is given in sorted order.**
that if two values in \( L \) are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter \( \delta \) such that \( 0 < \delta < 1 \). When we trim a list \( L \) by \( \delta \), we remove as many elements from \( L \) as possible, in such a way that if \( L_0 \) is the result of trimming \( L \), therefore element \( y \) that was removed from \( L \), there is an element \( \tilde{y} \) still in \( L_0 \) that approximates \( y \), that is, \( y \leq \delta \cdot \tilde{y} \).

We can think of such a \( \tilde{y} \) as "representing" \( y \) in the new list \( L_0 \). Each removed element \( y \) is represented by a remaining element \( \tilde{y} \) satisfying inequality (35.24).

For example, if \( \delta = 0.1 \) and \( L = \{10, 11, 12, 15, 20, 21, 22, 23, 24, 29\} \), then we can trim \( L \) to obtain \( L_0 = \{10, 12, 15, 20, 23, 29\} \), where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list \( L = \{y_1, y_2, \ldots, y_m\} \) in time \( O(m) \), given \( L \) and \( \delta \), assuming that \( L \) is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

\[ \text{TRIM}(L, \delta) \]

1. let \( m \) be the length of \( L \)
2. \( L' = \{y_1\} \)
3. \( \text{last} = y_1 \)
4. \( \text{for } i = 2 \text{ to } m \)
5. \( \quad \text{if } y_i > \text{last} \cdot (1 + \delta) \quad \text{// } y_i \geq \text{last} \text{ because } L \text{ is sorted} \)
6. \( \quad \quad \text{append } y_i \text{ onto the end of } L' \)
7. \( \quad \quad \text{last} = y_i \)
8. return \( L' \)
Illustration of the Trim Operation

**TRIM**(*L*, *δ*)
1. let *m* be the length of *L*
2. *L’* = ⟨*y*<sub>1</sub⟩
3. last = *y*<sub>1</sub>
4. for *i* = 2 to *m*
   5. if *y*<sub>*i*</sub> > last · (1 + *δ*) // *y*<sub>*i*</sub> ≥ last because *L* is sorted
      6. append *y*<sub>*i*</sub> onto the end of *L’*
      7. last = *y*<sub>*i*</sub>
6. return *L’*

**δ** = 0.1

*L* = ⟨10, 11, 12, 15, 20, 21, 22, 23, 24, 29⟩

*L’* = ⟨⟩
that if two values in $L$ are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter $\delta$ such that $0 < \delta < 1$. When we trim a list $L$ by $\delta$, we remove as many elements from $L$ as possible, in such a way that if $L_0$ is the result of trimming $L$, then for every element $y$ that was removed from $L$, there is an element $\tilde{y}$ still in $L_0$ that approximates $y$, that is, $y \leq \tilde{y} \leq \delta y$.

We can think of such a $\tilde{y}$ as "representing" $y$ in the new list $L_0$. Each removed element $y$ is represented by a remaining element $\tilde{y}$ satisfying inequality (35.24).

For example, if $\delta = 0.1$ and $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$; then we can trim $L$ to obtain $L_0 = \langle 10, 12, 15, 20, 23, 29 \rangle$; where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list $L = \langle y_1; y_2; \ldots; y_m \rangle$ in time $O(m)$, given $L$ and $\delta$, assuming that $L$ is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

TRIM($L, \delta$)

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. $last = y_1$
4. for $i = 2$ to $m$
   5. if $y_i > last \cdot (1 + \delta) \quad // \; y_i \geq last$ because $L$ is sorted
   6. append $y_i$ onto the end of $L'$
   7. $last = y_i$
8. return $L'$

$$\delta = 0.1$$

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$$

$$L' = \langle 10 \rangle$$
Illustration of the Trim Operation

\[ \text{TRIM}(L, \delta) \]
\begin{algorithm}
1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. for \( i = 2 \) to \( m \)
5. \hspace{1em} if \( y_i > \text{last} \cdot (1 + \delta) \) // \( y_i \geq \text{last} \) because \( L \) is sorted
6. \hspace{2em} append \( y_i \) onto the end of \( L' \)
7. \hspace{1em} \( \text{last} = y_i \)
8. return \( L' \)
\end{algorithm}

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10 \rangle \]
Illustration of the Trim Operation

**TRIM(L, δ)**

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. last = $y_1$
4. for $i = 2$ to $m$
   5. if $y_i > \text{last} \cdot (1 + \delta)$  // $y_i \geq \text{last}$ because $L$ is sorted
      6. append $y_i$ onto the end of $L'$
      7. last = $y_i$
6. return $L'$

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10 \rangle \]
Illustration of the Trim Operation

\text{T} \text{RIM}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. for \( i = 2 \) to \( m \)
5. \hspace{1em} if \( y_i > \text{last} \cdot (1 + \delta) \) \hspace{1em} // \( y_i \geq \text{last} \) because \( L \) is sorted
6. \hspace{2em} append \( y_i \) onto the end of \( L' \)
7. \hspace{2em} \( \text{last} = y_i \)
8. return \( L' \)

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10 \rangle \]
Illustration of the Trim Operation

**Trim (L, δ)**

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. for \( i = 2 \) to \( m \)
   5. if \( y_i > \text{last} \cdot (1 + \delta) \) \quad // \( y_i \geq \text{last} \) because \( L \) is sorted
   6. append \( y_i \) onto the end of \( L' \)
   7. \( \text{last} = y_i \)
5. return \( L' \)

\( \delta = 0.1 \)

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12 \rangle \]
Illustration of the Trim Operation

\textsc{Trim}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. \textbf{for } \( i = 2 \) \textbf{ to } \( m \)
5. \hspace{1em} \textbf{if } \( y_i > \text{last} \cdot (1 + \delta) \) \quad \text{\( \text{// } y_i \geq \text{last} \) because \( L \) is sorted}
6. \hspace{2em} \text{append } y_i \text{ onto the end of } L'
7. \hspace{1em} \text{last} = y_i
8. \textbf{return } L'

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12 \rangle \]
Illustration of the Trim Operation

\textbf{TRIM}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. \textbf{for} \( i = 2 \) \textbf{to} \( m \)
5. \quad \textbf{if} \( y_i > \text{last} \cdot (1 + \delta) \) \hspace{1em} \text{//} \ y_i \geq \text{last} \text{ because } L \text{ is sorted}
6. \quad \text{append } y_i \text{ onto the end of } L'
7. \quad \text{last} = y_i
8. \textbf{return} \ L'

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12 \rangle \]
Illustration of the Trim Operation

TRIM($L, \delta$)
1 let $m$ be the length of $L$
2 $L' = \langle y_1 \rangle$
3 last = $y_1$
4 for $i = 2$ to $m$
5 \quad if $y_i > \text{last} \cdot (1 + \delta)$ \quad // $y_i \geq \text{last}$ because $L$ is sorted
6 \quad \quad append $y_i$ onto the end of $L'$
7 \quad \quad last = $y_i$
8 return $L'$

$\delta = 0.1$

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

$L' = \langle 10, 12, 15 \rangle$
Illustration of the Trim Operation

**TRIM**(L, δ)
1. let m be the length of L
2. L' = \{y_1\}
3. last = y_1
4. for i = 2 to m
5. \hspace{1em} if y_i > last \cdot (1 + δ) \hspace{1em} \text{\texttt{// y_i ≥ last because L is sorted}}
6. \hspace{2em} append y_i onto the end of L'
7. \hspace{2em} last = y_i
8. return L'

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12, 15 \rangle \]
that if two values in $L$ are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter $\delta$ such that $0 < \delta < 1$. When we trim a list $L$ by $\delta$, we remove as many elements from $L$ as possible, in such a way that if $L_0$ is the result of trimming $L$, therefore element $y$ that was removed from $L$, there is an element $\hat{y}$ still in $L_0$ that approximates $y$, that is, $y \leq \frac{1}{\delta} \hat{y}$. (35.24) We can think of such a $\hat{y}$ as "representing" $y$ in the new list $L_0$. Each removed element $y$ is represented by a remaining element $\hat{y}$ satisfying inequality (35.24).

For example, if $\delta = 0.1$ and $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$; then we can trim $L$ to obtain $L_0 = \langle 10, 12, 15 \rangle$, where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in time $O(m)$, given $L$ and $\delta$, and assuming that $L$ is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

**TRIM($L$, $\delta$)**

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. last = $y_1$
4. for $i = 2$ to $m$
5.   if $y_i > last \cdot (1 + \delta)$ // $y_i \geq last$ because $L$ is sorted
6.     append $y_i$ onto the end of $L'$
7.     last = $y_i$
8. return $L'$

$\delta = 0.1$

\[
L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle
\]

$\hat{y} = \langle 10, 12, 15 \rangle$
Illustration of the Trim Operation

**Trim**($L, \delta$)

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. $last = y_1$
4. for $i = 2$ to $m$
5.   if $y_i > last \cdot (1 + \delta)$ \hspace{1cm} // $y_i \geq last$ because $L$ is sorted
6.     append $y_i$ onto the end of $L'$
7.     $last = y_i$
8. return $L'$

\[ \delta = 0.1 \]

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

$L' = \langle 10, 12, 15, 20 \rangle$
Illustration of the Trim Operation

\text{TRIM}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. \textbf{for} \( i = 2 \) \textbf{to} \( m \)
5. \quad \textbf{if} \( y_i > \text{last} \cdot (1 + \delta) \) \quad // \( y_i \geq \text{last} \) because \( L \) is sorted
6. \quad \quad \text{append} \ y_i \ \text{onto the end of} \ L'
7. \quad \quad \text{last} = y_i
8. \textbf{return} \ L'

\( \delta = 0.1 \)

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ \text{last} \]

\[ i \]

\[ L' = \langle 10, 12, 15, 20 \rangle \]
Illustration of the Trim Operation

**Trim**($L, \delta$)

1. let $m$ be the length of $L$
2. $L' = \langle y_1 \rangle$
3. last = $y_1$
4. for $i = 2$ to $m$
   5. if $y_i > last \cdot (1 + \delta)$  // $y_i \geq last$ because $L$ is sorted
      6. append $y_i$ onto the end of $L'$
      7. last = $y_i$
6. return $L'$

$\delta = 0.1$

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

$L' = \langle 10, 12, 15, 20 \rangle$
Illustration of the Trim Operation

**TRIM**(\(L, \delta\))

1. let \(m\) be the length of \(L\)
2. \(L' = \langle y_1 \rangle\)
3. last = \(y_1\)
4. for \(i = 2\) to \(m\)
5. if \(y_i > \text{last} \cdot (1 + \delta)\) \(// y_i \geq \text{last}\) because \(L\) is sorted
6. append \(y_i\) onto the end of \(L'\)
7. last = \(y_i\)
8. return \(L'\)

\(\delta = 0.1\)

\(L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle\)

\(L' = \langle 10, 12, 15, 20 \rangle\)
Illustration of the Trim Operation

\textbf{Trim}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. for \( i = 2 \) to \( m \)
5. \hspace{1em} if \( y_i > \text{last} \cdot (1 + \delta) \) \hspace{1em} // \( y_i \geq \text{last} \) because \( L \) is sorted
6. \hspace{2em} append \( y_i \) onto the end of \( L' \)
7. \hspace{2em} \( \text{last} = y_i \)
8. return \( L' \)

\( \delta = 0.1 \)

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12, 15, 20 \rangle \]
Illustration of the Trim Operation

**Trim Operation**

Let \( L \) be a list of integers sorted in non-decreasing order.

1. Let \( m \) be the length of \( L \).
2. Initialize \( L' = \{y_1\} \).
3. Initialize \( \text{last} = y_1 \).
4. For \( i = 2 \) to \( m \):
   - If \( y_i > \text{last} \cdot (1 + \delta) \) (\( \delta = 0.1 \)):
     - Append \( y_i \) to \( L' \).
     - Update \( \text{last} = y_i \).
5. Return \( L' \).

**Example**

Let \( L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \) and \( \delta = 0.1 \).

After trimming:

\[ L' = \langle 10, 12, 15, 20, 23 \rangle \]
Illustration of the Trim Operation

**Trim** \((L, \delta)\)

1. let \(m\) be the length of \(L\)
2. \(L' = \langle y_1 \rangle\)
3. \(last = y_1\)
4. for \(i = 2\) to \(m\)
   - if \(y_i > last \cdot (1 + \delta)\) \(\quad // y_i \geq last\) because \(L\) is sorted
     - append \(y_i\) onto the end of \(L'\)
     - \(last = y_i\)
5. return \(L'\)

\[
\delta = 0.1
\]

\[
L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle
\]

\[
L' = \langle 10, 12, 15, 20, 23 \rangle
\]
Illustration of the Trim Operation

\textbf{Trim}(L, \delta)
1 let \( m \) be the length of \( L \)
2 \( L' = \langle y_1 \rangle \)
3 \( \text{last} = y_1 \)
4 \textbf{for} \( i = 2 \) \textbf{to} \( m \)
5 \textbf{if} \( y_i > \text{last} \cdot (1 + \delta) \) \quad \ll  y_i \geq \text{last} \) because \( L \) is sorted
6 \quad \text{append} \( y_i \) onto the end of \( L' \)
7 \quad \text{last} = y_i
8 \textbf{return} \( L' \)

\[ \delta = 0.1 \]

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12, 15, 20, 23 \rangle \]
Illustration of the Trim Operation

TRIM($L, \delta$)
1 let $m$ be the length of $L$
2 $L' = \langle y_1 \rangle$
3 last = $y_1$
4 for $i = 2$ to $m$
5     if $y_i > \text{last} \cdot (1 + \delta)$ /* $y_i \geq \text{last}$ because $L$ is sorted */
6         append $y_i$ onto the end of $L'$
7         last = $y_i$
8 return $L'$

$\delta = 0.1$

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

$L' = \langle 10, 12, 15, 20, 23 \rangle$
that if two values in \( L \) are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter \( \delta \) such that \( 0 < \delta < 1 \). When we trim a list \( L \) by \( \delta \), we remove as many elements from \( L \) as possible, in such a way that if \( L_0 \) is the result of trimming \( L \), therefore every element \( y \) that was removed from \( L \) there is an element \( y' \) still in \( L_0 \) that approximates \( y \), that is, \( y \leq C \delta / DC \leq y' \):

\[
(35.24)
\]

We can think of such a \( y' \) as “representing” \( y \) in the new list \( L_0 \). Each removed element \( y \) is represented by a remaining element \( y' \) satisfying inequality (35.24).

For example, if \( \delta = 0.1 \) and \( L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \); then we can trim \( L \) to obtain \( L_0 = \langle 10, 12, 15, 20, 23, 29 \rangle \), where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list \( L = \langle y_1, y_2, \ldots, y_m \rangle \) in time \( O(m) \), given \( L \) and \( \delta \), assuming that \( L \) is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

\[
\text{TRIM}(L, \delta)
\]

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. last = \( y_1 \)
4. for \( i = 2 \) to \( m \)
5. if \( y_i > last \cdot (1 + \delta) \) \( \text{// } y_i \geq last \) because \( L \) is sorted
6. append \( y_i \) onto the end of \( L' \)
7. last = \( y_i \)
8. return \( L' \)

\[
\delta = 0.1
\]

\[
L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle
\]

\[
L' = \langle 10, 12, 15, 20, 23, 29 \rangle
\]
Illustration of the Trim Operation

**Trim** \((L, \delta)\)

1. let \(m\) be the length of \(L\)
2. \(L' = \langle y_1 \rangle\)
3. last = \(y_1\)
4. for \(i = 2\) to \(m\)
5. \(\text{if } y_i > \text{last} \cdot (1 + \delta) \quad \text{// } y_i \geq \text{last} \text{ because } L \text{ is sorted}\)
6. append \(y_i\) onto the end of \(L'\)
7. last = \(y_i\)
8. return \(L'\)

\[\delta = 0.1\]

\(L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle\)

\(L' = \langle 10, 12, 15, 20, 23, 29 \rangle\)
The FPTAS

**APPROX-SUBSET-SUM** \((S, t, \epsilon)\)

1.  \(n = |S|\)
2.  \(L_0 = \langle 0 \rangle\)
3.  \(\text{for } i = 1 \text{ to } n\)
4.    \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5.    \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6.    remove from \(L_i\) every element that is greater than \(t\)
7.  let \(z^*\) be the largest value in \(L_n\)
8.  return \(z^*\)
The FPTAS

**APPROX-SUBSET-SUM** \((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \{0\}\)
3. for \(i = 1\) to \(n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. remove from \(L_i\) every element that is greater than \(t\)
7. let \(z^*\) be the largest value in \(L_n\)
8. return \(z^*\)

**EXACT-SUBSET-SUM** \((S, t)\)

1. \(n = |S|\)
2. \(L_0 = \{0\}\)
3. for \(i = 1\) to \(n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. remove from \(L_i\) every element that is greater than \(t\)
6. return the largest element in \(L_n\)
The FPTAS

**APPROX-SUBSET-SUM** \((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. \(\text{for } i = 1 \text{ to } n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. remove from \(L_i\) every element that is greater than \(t\)
7. let \(z^*\) be the largest value in \(L_n\)
8. return \(z^*\)

**EXACT-SUBSET-SUM** \((S, t)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. \(\text{for } i = 1 \text{ to } n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. remove from \(L_i\) every element that is greater than \(t\)
6. return the largest element in \(L_n\)

Repeated application of TRIM to make sure \(L_i\)’s remain short.
The FPTAS

**Approx-Subset-Sum** \((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \{0\}\)
3. for \(i = 1\) to \(n\) 
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. remove from \(L_i\) every element that is greater than \(t\)
7. let \(z^*\) be the largest value in \(L_n\)
8. return \(z^*\)

**Exact-Subset-Sum** \((S, t)\)

1. \(n = |S|\)
2. \(L_0 = \{0\}\)
3. for \(i = 1\) to \(n\) 
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. remove from \(L_i\) every element that is greater than \(t\)
6. return the largest element in \(L_n\)

Repeated application of \(\text{TRIM}\) to make sure \(L_i\)'s remain short.

- We must bound the inaccuracy introduced by repeated trimming
The FPTAS

**Approx-Subset-Sum** $(S, t, \epsilon)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
4. $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
5. $L_i = \text{TRIM}(L_i, \epsilon/2n)$
6. remove from $L_i$ every element that is greater than $t$
7. let $z^*$ be the largest value in $L_n$
8. return $z^*$

Repeating application of TRIM to make sure $L_i$’s remain short.

**Exact-Subset-Sum** $(S, t)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
4. $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
5. remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$

- We must bound the inaccuracy introduced by repeated trimming
- We must show that the algorithm is polynomial time
The FPTAS

Approx-Subset-Sum\((S,t,\epsilon)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. for \( i = 1 \) to \( n \)
   4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
   5. \( L_i = \text{TRIM}(L_i, \epsilon/2n) \)
4. remove from \( L_i \) every element that is greater than \( t \)
6. let \( z^* \) be the largest value in \( L_n \)
7. return \( z^* \)

Exact-Subset-Sum\((S,t)\)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. for \( i = 1 \) to \( n \)
   4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
   5. remove from \( L_i \) every element that is greater than \( t \)
6. return the largest element in \( L_n \)

Repeated application of TRIM to make sure \( L_i \)'s remain short.

- We must bound the inaccuracy introduced by repeated trimming
- We must show that the algorithm is polynomial time

Solution is a careful choice of \( \delta! \)
Running through an Example (CLRS3)

\textbf{APPROX-SUBSET-SUM}(S, t, \epsilon)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. \textbf{for} \( i = 1 \) \textbf{to} \( n \)
4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
5. \( L_i = \text{TRIM}(L_i, \epsilon/2n) \)
6. remove from \( L_i \) every element that is greater than \( t \)
7. let \( z^* \) be the largest value in \( L_n \)
8. \textbf{return} \( z^* \)
Running through an Example (CLRS3)

\textbf{APPROX-SUBSET-SUM}(S, t, \epsilon)

1. \hspace{1em} n = |S|
2. \hspace{1em} L_0 = \langle 0 \rangle
3. \hspace{1em} \textbf{for} i = 1 \textbf{ to } n
4. \hspace{2em} L_i = \textsc{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)
5. \hspace{2em} L_i = \textsc{Trim}(L_i, \epsilon / 2n)
6. \hspace{2em} \text{remove from } L_i \text{ every element that is greater than } t
7. \hspace{1em} \text{let } z^* \text{ be the largest value in } L_n
8. \hspace{1em} \textbf{return} z^*

- **Input:** $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM**\((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. **for** \(i = 1\) **to** \(n\)
   4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
   5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. remove from \(L_i\) every element that is greater than \(t\)
7. let \(z^*\) be the largest value in \(L_n\)
8. **return** \(z^*\)

- **Input:** \(S = \langle 104, 102, 201, 101 \rangle, \ t = 308, \ \epsilon = 0.4\)
- **⇒** **Trimming parameter:** \(\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05\)
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM**(*S*, *t*, *ε*)

1.  
   \[n = |S|\]
2.  
   \[L_0 = \langle 0 \rangle\]
3.  
   for *i* = 1 to *n*
4.  
   \[L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\]
5.  
   \[L_i = \text{TRIM}(L_i, \epsilon/2n)\]
6.  
   remove from *L* *i* every element that is greater than *t*
7.  
   let *z* *\ast* be the largest value in *L* *n*
8.  
   return *z* *\ast*  

- **Input:** *S* = \langle 104, 102, 201, 101 \rangle, *t* = 308, *ε* = 0.4

\[\Rightarrow\]**Trimming parameter:** \[\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05\]

- **line 2:** *L* *0* = \langle 0 \rangle
Running through an Example (CLRS3)

\textbf{APPROX-SUBSET-SUM}(S, t, \epsilon)

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. \textbf{for} \( i = 1 \) \textbf{to} \( n \)
4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
5. \( L_i = \text{TRIM}(L_i, \epsilon / 2n) \)
6. remove from \( L_i \) every element that is greater than \( t \)
7. let \( z^* \) be the largest value in \( L_n \)
8. \textbf{return} \( z^* \)

- \textbf{Input}: \( S = \langle 104, 102, 201, 101 \rangle, t = 308, \epsilon = 0.4 \)
- \textbf{Trimming parameter}: \( \delta = \epsilon / (2 \cdot n) = \epsilon / 8 = 0.05 \)
  - line 2: \( L_0 = \langle 0 \rangle \)
  - line 4: \( L_1 = \langle 0, 104 \rangle \)
Approx-Sum-Sum(δ, t, ϵ)
1  \( n = |S| \)
2  \( L_0 = \langle 0 \rangle \)
3  for \( i = 1 \) to \( n \)
4    \( L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i) \)
5    \( L_i = \text{Trim}(L_i, \epsilon/2n) \)
6      remove from \( L_i \) every element that is greater than \( t \)
7  let \( z^* \) be the largest value in \( L_n \)
8  return \( z^* \)

- Input: \( S = \langle 104, 102, 201, 101 \rangle \), \( t = 308 \), \( \epsilon = 0.4 \)
- Trimming parameter: \( \delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05 \)

  - line 2: \( L_0 = \langle 0 \rangle \)
  - line 4: \( L_1 = \langle 0, 104 \rangle \)
  - line 5: \( L_1 = \langle 0, 104 \rangle \)
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM**($S$, $t$, $\epsilon$)

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. **for** $i = 1$ **to** $n$
4. \hspace{1em} $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
5. \hspace{1em} $L_i = \text{TRIM}(L_i, \epsilon/2n)$
6. \hspace{1em} remove from $L_i$ every element that is greater than $t$
7. let $z^*$ be the largest value in $L_n$
8. **return** $z^*$

- **Input:** $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$
- **Trimming parameter:** $\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05$

- **line 2:** $L_0 = \langle 0 \rangle$
- **line 4:** $L_1 = \langle 0, 104 \rangle$
- **line 5:** $L_1 = \langle 0, 104 \rangle$
- **line 6:** $L_1 = \langle 0, 104 \rangle$
Running through an Example (CLRS3)

\[
\text{APPX-\textsc{subset-sum}}(S, t, \epsilon)
\]

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. \(\text{for } i = 1 \text{ to } n\)
4. \(L_i = \text{MERGE-\textsc{lists}}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. \(\text{remove from } L_i \text{ every element that is greater than } t\)
7. \(\text{let } z^* \text{ be the largest value in } L_n\)
8. \(\text{return } z^*\)

- **Input**: \(S = \langle 104, 102, 201, 101 \rangle, t = 308, \epsilon = 0.4\)
- **Trimming parameter**: \(\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05\)
  - line 2: \(L_0 = \langle 0 \rangle\)
  - line 4: \(L_1 = \langle 0, 104 \rangle\)
  - line 5: \(L_1 = \langle 0, 104 \rangle\)
  - line 6: \(L_1 = \langle 0, 104 \rangle\)
  - line 4: \(L_2 = \langle 0, 102, 104, 206 \rangle\)
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM**\((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. **for** \(i = 1\) **to** \(n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. **remove** from \(L_i\) **every** element that is greater than \(t\)
7. **let** \(z^*\) **be** the largest value in \(L_n\)
8. **return** \(z^*\)

- **Input:** \(S = \langle 104, 102, 201, 101 \rangle\), \(t = 308\), \(\epsilon = 0.4\)

\(\Rightarrow\) **Trimming parameter:** \(\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05\)

- **line 2:** \(L_0 = \langle 0 \rangle\)
- **line 4:** \(L_1 = \langle 0, 104 \rangle\)
- **line 5:** \(L_1 = \langle 0, 104 \rangle\)
- **line 6:** \(L_1 = \langle 0, 104 \rangle\)
- **line 4:** \(L_2 = \langle 0, 102, 104, 206 \rangle\)
- **line 5:** \(L_2 = \langle 0, 102, 206 \rangle\)
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM**(*S*, *t*, *ε*)

1.  
   \[ n = |S| \]
2.  
   \[ L_0 = \{0\} \]
3.  
   **for** *i* = 1 **to** *n*
4.  
   \[ L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \]
5.  
   \[ L_i = \text{TRIM}(L_i, \epsilon/2n) \]
6.  
   **remove** from *L* *i* **every** element **that** is greater than *t*
7.  
   **let** *z*\(^*\) **be** the largest **value** in *L* *n*
8.  
   **return** *z*\(^*\)

- **Input:** *S* = \{104, 102, 201, 101\}, *t* = 308, *ε* = 0.4

⇒ **Trimming parameter:** \[ \delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05 \]

- **line** 2: *L* 0 = \{0\}
- **line** 4: *L* 1 = \{0, 104\}
- **line** 5: *L* 1 = \{0, 104\}
- **line** 6: *L* 1 = \{0, 104\}
- **line** 4: *L* 2 = \{0, 102, 104, 206\}
- **line** 5: *L* 2 = \{0, 102, 206\}
- **line** 6: *L* 2 = \{0, 102, 206\}
### Running through an Example (CLRS3)

**APPROX-SUBSET-SUM** ($S, t, \epsilon$)

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
   4. $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
   5. $L_i = \text{TRIM}(L_i, \epsilon/2n)$
   6. remove from $L_i$ every element that is greater than $t$
4. let $z^*$ be the largest value in $L_n$
5. return $z^*$

- **Input:** $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$

  - **Trimming parameter:** $\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05$

- line 2: $L_0 = \langle 0 \rangle$
- line 4: $L_1 = \langle 0, 104 \rangle$
- line 5: $L_1 = \langle 0, 104 \rangle$
- line 6: $L_1 = \langle 0, 104 \rangle$
- line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
- line 5: $L_2 = \langle 0, 102, 206 \rangle$
- line 6: $L_2 = \langle 0, 102, 206 \rangle$
- line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
Running through an Example (CLRS3)

\textsc{Approx-Subset-Sum}(S, t, \epsilon)

1 \quad n = |S|
2 \quad L_0 = \langle 0 \rangle
3 \quad \text{for } i = 1 \text{ to } n
4 \quad \quad L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)
5 \quad \quad L_i = \text{Trim}(L_i, \epsilon/2n)
6 \quad \text{remove from } L_i \text{ every element that is greater than } t
7 \quad \text{let } z^\ast \text{ be the largest value in } L_n
8 \quad \text{return } z^\ast

- \textbf{Input: } S = \langle 104, 102, 201, 101 \rangle, \ t = 308, \ \epsilon = 0.4

\Rightarrow \textbf{Trimming parameter: } \delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05

- line 2: \ L_0 = \langle 0 \rangle
- line 4: \ L_1 = \langle 0, 104 \rangle
- line 5: \ L_1 = \langle 0, 104 \rangle
- line 6: \ L_1 = \langle 0, 104 \rangle
- line 4: \ L_2 = \langle 0, 102, 104, 206 \rangle
- line 5: \ L_2 = \langle 0, 102, 206 \rangle
- line 6: \ L_2 = \langle 0, 102, 206 \rangle
- line 4: \ L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle
- line 5: \ L_3 = \langle 0, 102, 201, 303, 407 \rangle
Running through an Example (CLRS3)

\textsc{approx-subset-sum}(S, t, \epsilon)

1 \quad n = |S|
2 \quad L_0 = \langle 0 \rangle
3 \quad \textbf{for} i = 1 \textbf{ to } n
4 \quad \quad L_i = \textsc{merge-lists}(L_{i-1}, L_{i-1} + x_i)
5 \quad \quad L_i = \textsc{trim}(L_i, \epsilon/2n)
6 \quad \quad \text{remove from } L_i \text{ every element that is greater than } t
7 \quad \text{let } z^* \text{ be the largest value in } L_n
8 \quad \textbf{return } z^*

- \textbf{Input: } S = \langle 104, 102, 201, 101 \rangle, \ t = 308, \ \epsilon = 0.4

⇒ \textbf{Trimming parameter: } \delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05

- line 2: \ \text{ } L_0 = \langle 0 \rangle
- line 4: \ \text{ } L_1 = \langle 0, 104 \rangle
- line 5: \ \text{ } L_1 = \langle 0, 104 \rangle
- line 6: \ \text{ } L_1 = \langle 0, 104 \rangle
- line 4: \ \text{ } L_2 = \langle 0, 102, 104, 206 \rangle
- line 5: \ \text{ } L_2 = \langle 0, 102, 206 \rangle
- line 6: \ \text{ } L_2 = \langle 0, 102, 206 \rangle
- line 4: \ \text{ } L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle
- line 5: \ \text{ } L_3 = \langle 0, 102, 201, 303, 407 \rangle
- line 6: \ \text{ } L_3 = \langle 0, 102, 201, 303 \rangle
Running through an Example (CLRS3)

**APPROX-SUBSET-SUM(S, t, ε)**

1. \( n = |S| \)
2. \( L_0 = \langle 0 \rangle \)
3. **for** \( i = 1 \) **to** \( n \)
4. \( L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \)
5. \( L_i = \text{TRIM}(L_i, \epsilon/2n) \)
6. remove from \( L_i \) every element that is greater than \( t \)
7. let \( z^* \) be the largest value in \( L_n \)
8. **return** \( z^* \)

- **Input:** \( S = \langle 104, 102, 201, 101 \rangle, t = 308, \epsilon = 0.4 \)

**⇒ Trimming parameter:** \( \delta = \epsilon / (2 \cdot n) = \epsilon / 8 = 0.05 \)

- line 2: \( L_0 = \langle 0 \rangle \)
- line 4: \( L_1 = \langle 0, 104 \rangle \)
- line 5: \( L_1 = \langle 0, 104 \rangle \)
- line 6: \( L_1 = \langle 0, 104 \rangle \)
- line 4: \( L_2 = \langle 0, 102, 104, 206 \rangle \)
- line 5: \( L_2 = \langle 0, 102, 206 \rangle \)
- line 6: \( L_2 = \langle 0, 102, 206 \rangle \)
- line 4: \( L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle \)
- line 5: \( L_3 = \langle 0, 102, 201, 303, 407 \rangle \)
- line 6: \( L_3 = \langle 0, 102, 201, 303 \rangle \)
- line 4: \( L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle \)
Running through an Example (CLRS3)

Approx-Subset-Sum(S, t, \epsilon)

1  n = |S|
2  L_0 = \langle 0 \rangle
3  for i = 1 to n
4      L_i = Merge-Lists(L_{i-1}, L_{i-1} + x_i)
5      L_i = Trim(L_i, \epsilon/2n)
6      remove from L_i every element that is greater than t
7  let z* be the largest value in L_n
8  return z*

- Input: S = \langle 104, 102, 201, 101 \rangle, t = 308, \epsilon = 0.4

⇒ Trimming parameter: \delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05

- line 2: L_0 = \langle 0 \rangle
- line 4: L_1 = \langle 0, 104 \rangle
- line 5: L_1 = \langle 0, 104 \rangle
- line 6: L_1 = \langle 0, 104 \rangle
- line 4: L_2 = \langle 0, 102, 104, 206 \rangle
- line 5: L_2 = \langle 0, 102, 206 \rangle
- line 6: L_2 = \langle 0, 102, 206 \rangle
- line 4: L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle
- line 5: L_3 = \langle 0, 102, 201, 303, 407 \rangle
- line 6: L_3 = \langle 0, 102, 201, 303 \rangle
- line 4: L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle
- line 5: L_4 = \langle 0, 101, 201, 302, 404 \rangle
Running through an Example (CLRS3)

\textsc{approx-subset-sum}(S, t, \epsilon)
\begin{enumerate}
\item $n = |S|$
\item $L_0 = \langle 0 \rangle$
\item \textbf{for} $i = 1$ \textbf{to} $n$
\item \hspace{1em} $L_i = \text{merge-lists}(L_{i-1}, L_{i-1} + x_i)$
\item \hspace{1em} $L_i = \text{trim}(L_i, \epsilon/2n)$
\item \hspace{1em} remove from $L_i$ every element that is greater than $t$
\item \hspace{1em} let $z^*$ be the largest value in $L_n$
\item \hspace{1em} \textbf{return} $z^*$
\end{enumerate}

- \textbf{Input: } $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$

\Rightarrow \textbf{Trimming parameter: } $\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05$

\begin{itemize}
\item line 2: $L_0 = \langle 0 \rangle$
\item line 4: $L_1 = \langle 0, 104 \rangle$
\item line 5: $L_1 = \langle 0, 104 \rangle$
\item line 6: $L_1 = \langle 0, 104 \rangle$
\item line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
\item line 5: $L_2 = \langle 0, 102, 206 \rangle$
\item line 6: $L_2 = \langle 0, 102, 206 \rangle$
\item line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
\item line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$
\item line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$
\item line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
\item line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$
\item line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$
\end{itemize}
### Running through an Example (CLRS3)

**APPROX-SUBSET-SUM** $(S, t, \epsilon)$

1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. **for** $i = 1$ to $n$
4. $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$
5. $L_i = \text{TRIM}(L_i, \epsilon/2n)$
6. remove from $L_i$ every element that is greater than $t$
7. let $z^*$ be the largest value in $L_n$
8. **return** $z^*$

- **Input:** $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$

⇒ **Trimming parameter:** $\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05$

- line 2: $L_0 = \langle 0 \rangle$
- line 4: $L_1 = \langle 0, 104 \rangle$
- line 5: $L_1 = \langle 0, 104 \rangle$
- line 6: $L_1 = \langle 0, 104 \rangle$
- line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
- line 5: $L_2 = \langle 0, 102, 206 \rangle$
- line 6: $L_2 = \langle 0, 102, 206 \rangle$
- line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
- line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$
- line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$
- line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
- line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$
- line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$

 Returned solution $z^* = 302$, which is 2% within the optimum $307 = 104 + 102 + 101$
Reminder: Performance Ratios for Approximation Algorithms

### Approximation Ratio

An algorithm for a problem has **approximation ratio** $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

For many problems: **tradeoff between runtime and approximation ratio.**

### Approximation Schemes

An **approximation scheme** is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a **polynomial-time approximation scheme** (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. **For example, $O(n^2/\epsilon)$.**
- It is a **fully polynomial-time approximation scheme** (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. **For example, $O((1/\epsilon)^2 \cdot n^3)$.**
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

Can be shown by induction on $i$

Taylor approximation of $e$
Theorem 35.8

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution $\checkmark$

IV. Approximation via Exact Algorithms

The Subset-Sum Problem
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution $\checkmark$
- Let $y^*$ denote an optimal solution
**Analysis of APPROX-SUBSET-SUM**

**Theorem 35.8**

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

**Proof (Approximation Ratio):**

- Returned solution $z^*$ is a valid solution $✓$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.
Theorem 35.8

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution ✓
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y$$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution ✓
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y$$

Can be shown by induction on $i$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):
- Returned solution $z^*$ is a valid solution $✓$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y=y^*, i=n$$

Can be shown by induction on $i$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution \( z^* \) is a valid solution \( \checkmark \)
- Let \( y^* \) denote an optimal solution
- For every possible sum \( y \leq t \) of \( x_1, \ldots, x_i \), there exists an element \( z \in L_i' \) s.t.:

\[
\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y = y^*, i = n \quad \Rightarrow \quad \frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*
\]

Can be shown by induction on \( i \)
Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution $✓$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

\[
\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y=y^*, i=n \Rightarrow \quad \frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*
\]

\[
\frac{y^*}{z} \leq \left(1 + \frac{\epsilon}{2n}\right)^n,
\]

Can be shown by induction on $i$.
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution ✓
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y=y^*, i=n \quad \frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*$$

Can be shown by induction on $i$

and now using the fact that $(1 + \frac{\epsilon}{2n})^n \xrightarrow{n \to \infty} e^{\epsilon/2}$ yields
Analysis of APPROX-SUBSET-SUM

**Theorem 35.8**

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

**Proof (Approximation Ratio):**

- Returned solution $z^*$ is a valid solution $\checkmark$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

\[
\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y = y^*, i = n
\]

\[
\frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*
\]

Can be shown by induction on $i$

and now using the fact that $\left(1 + \frac{\epsilon}{2n}\right)^n \xrightarrow{n \to \infty} e^{\frac{\epsilon}{2}}$ yields

\[
\frac{y^*}{z} \leq e^{\frac{\epsilon}{2}}
\]
Analysis of APPROX-SUBSET-SUM

**Theorem 35.8**

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

**Proof (Approximation Ratio):**

- Returned solution $z^*$ is a valid solution $\checkmark$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

\[
\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad y=y^*, i=n \quad \frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*
\]

Can be shown by induction on $i$

and now using the fact that

\[
\left(1 + \frac{\epsilon/2}{n}\right)^n \xrightarrow{n \to \infty} e^{\epsilon/2} \quad \text{yields}
\]

\[
\frac{y^*}{z} \leq e^{\epsilon/2} \quad \text{Taylor approximation of } e
\]
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution ✓
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad \Rightarrow \quad \frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*$$

Can be shown by induction on $i$

and now using the fact that $\left(1 + \frac{\epsilon/2}{n}\right)^n \nrightarrow e^{\epsilon/2}$ yields

$$\frac{y^*}{z} \leq e^{\epsilon/2} \quad \text{Taylor approximation of } e$$

$$\leq 1 + \frac{\epsilon}{2} + (\frac{\epsilon}{2})^2$$
Analysis of **APPROX-SUBSET-SUM**

**Theorem 35.8**

**APPROX-SUBSET-SUM** is a FPTAS for the subset-sum problem.

Proof (Approximation Ratio):

- Returned solution $z^*$ is a valid solution $✓$
- Let $y^*$ denote an optimal solution
- For every possible sum $y \leq t$ of $x_1, \ldots, x_i$, there exists an element $z \in L'_i$ s.t.:

$$\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y$$

$y = y^*, i=n$  

$$\frac{y^*}{(1 + \epsilon/(2n))^n} \leq z \leq y^*$$

Can be shown by induction on $i$

and now using the fact that 

$$\left(1 + \frac{\epsilon/2}{n}\right)^n \xrightarrow{n \to \infty} e^{\epsilon/2}$$

yields

$$\frac{y^*}{z} \leq e^{\epsilon/2}$$  

**Taylor approximation of $e$**

$$\leq 1 + \epsilon/2 + (\epsilon/2)^2 \leq 1 + \epsilon$$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

...
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

- **Strategy**: Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

- **Strategy:** Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$
Analysis of **APPROX-SUBSET-SUM**

**Theorem 35.8**

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

**Proof (Running Time):**
- **Strategy:** Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$
  
  $\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\left\lfloor \log_{1+\epsilon/(2n)} t \right\rfloor$ additional values.
Analysis of APPROX-SUBSET-SUM

Theorem 35.8
APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):
- **Strategy**: Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\lceil \log_{1+\epsilon/(2n)} t \rceil$ additional values.

Hence,

$$\log_{1+\epsilon/(2n)} t + 2 =$$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

- **Strategy**: Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\lceil \log_{1+\epsilon/(2n)} t \rceil$ additional values. Hence,

$$\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2$$
Analysis of APPROX-SUBSET-SUM

**Theorem 35.8**

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

**Proof (Running Time):**

- **Strategy:** Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\lfloor \log_{1+\epsilon/(2n)} t \rfloor$ additional values. Hence,

$$
\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2
$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

- **Strategy**: Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\lfloor \log_{1+\epsilon/(2n)} t \rfloor$ additional values.

Hence,

$$\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2$$

$$\leq \frac{2n(1 + \epsilon/(2n)) \ln t}{\epsilon} + 2$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8
APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):
- **Strategy**: Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\lceil \log_{1+\epsilon/(2n)} t \rceil$ additional values. Hence,

$$\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2$$

$$\leq \frac{2n(1 + \epsilon/(2n)) \ln t}{\epsilon} + 2$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$

$$< \frac{3n \ln t}{\epsilon} + 2.$$
Analysis of APPROX-SUBSET-SUM

Theorem 35.8

APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):

- **Strategy:** Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

$\Rightarrow$ Possible Values after trimming are 0, 1, and up to $\left\lfloor \log_{1+\epsilon/(2n)} t \right\rfloor$ additional values. Hence,

$$\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2 \leq \frac{2n(1 + \epsilon/(2n)) \ln t}{\epsilon} + 2$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$

- This bound on $|L_i|$ is polynomial in the size of the input and in $1/\epsilon$. 

IV. Approximation via Exact Algorithms The Subset-Sum Problem
Analysis of APPROX-SUBSET-SUM

Theorem 35.8
APPROX-SUBSET-SUM is a FPTAS for the subset-sum problem.

Proof (Running Time):
- **Strategy:** Derive a bound on $|L_i|$ (running time is linear in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

⇒ Possible Values after trimming are 0, 1, and up to $\lceil \log_{1+\epsilon/(2n)} t \rceil$ additional values.

Hence,

$$
\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2 \\
\leq \frac{2n(1 + \epsilon/(2n)) \ln t}{\epsilon} + 2
$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$

$$
\ln(1 + x) \geq \frac{x}{1+x} < \frac{3n \ln t}{\epsilon} + 2.
$$

- This bound on $|L_i|$ is polynomial in the size of the input and in $1/\epsilon$.

Need $\log(t)$ bits to represent $t$ and $n$ bits to represent $S$.
Concluding Remarks

The Subset-Sum Problem

- **Given**: Set of positive integers $S = \{x_1, x_2, \ldots, x_n\}$ and positive integer $t$
- **Goal**: Find a subset $S' \subseteq S$ which maximizes $\sum_{i: x_i \in S'} x_i \leq t$. 

The Knapsack Problem

A more general problem than Subset-Sum

There is a FPTAS for the Knapsack problem.

Theorem

Algorithm very similar to APPROX-SUBSET-SUM
The Subset-Sum Problem

- **Given**: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

**Theorem 35.8**

\( \text{APPROX-SUBSET-SUM} \) is a **FPTAS** for the subset-sum problem.
Concluding Remarks

The Subset-Sum Problem

- **Given**: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

Theorem 35.8

**APPROX-SUBSET-SUM** is a FPTAS for the subset-sum problem.

The Knapsack Problem

- **Given**: Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
**Concluding Remarks**

---

**The Subset-Sum Problem**

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

---

**Theorem 35.8**

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

---

**The Knapsack Problem**

- **Given:** Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which
Concluding Remarks

The Subset-Sum Problem

- **Given**: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

Theorem 35.8

\textsc{Approx-Subset-Sum} is a \textsc{FPTAS} for the subset-sum problem.

The Knapsack Problem

- **Given**: Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which
  1. maximizes \( \sum_{i \in S'} v_i \)
  2. satisfies \( \sum_{i \in S'} w_i \leq t \).
Concluding Remarks

The Subset-Sum Problem

- **Given**: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

Theorem 35.8

**APPROX-SUBSET-SUM** is a FPTAS for the subset-sum problem.

The Knapsack Problem

- **Given**: Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which
  1. maximizes \( \sum_{i \in S'} v_i \)
  2. satisfies \( \sum_{i \in S'} w_i \leq t \)

A more general problem than Subset-Sum
Concluding Remarks

The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

Theorem 35.8

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

The Knapsack Problem

- **Given:** Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which
  1. maximizes \( \sum_{i \in S'} v_i \)
  2. satisfies \( \sum_{i \in S'} w_i \leq t \)

Theorem

There is a **FPTAS** for the Knapsack problem.
The Subset-Sum Problem

- **Given**: Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i\in S'} x_i \leq t \).

**Theorem 35.8**

**APPROX-SUBSET-SUM** is a **FPTAS** for the subset-sum problem.

The Knapsack Problem

- **Given**: Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal**: Find a subset \( S' \subseteq S \) which
  1. maximizes \( \sum_{i\in S'} v_i \)
  2. satisfies \( \sum_{i\in S'} w_i \leq t \)

**Theorem**

There is a **FPTAS** for the Knapsack problem.
Outline

The Subset-Sum Problem

Parallel Machine Scheduling

Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Parallel Machine Scheduling

Machine Scheduling Problem

- **Given:** $n$ jobs $J_1, J_2, \ldots, J_n$ with processing times $p_1, p_2, \ldots, p_n$, and $m$ identical machines $M_1, M_2, \ldots, M_m$
Parallel Machine Scheduling

Machine Scheduling Problem

- **Given:** $n$ jobs $J_1, J_2, \ldots, J_n$ with processing times $p_1, p_2, \ldots, p_n$, and $m$ identical machines $M_1, M_2, \ldots, M_m$

- **Goal:** Schedule the jobs on the machines minimizing the makespan $C_{\text{max}} = \max_{1 \leq j \leq n} C_j$, where $C_k$ is the completion time of job $J_k$. 

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
Parallel Machine Scheduling

- **Given**: \( n \) jobs \( J_1, J_2, \ldots, J_n \) with processing times \( p_1, p_2, \ldots, p_n \), and \( m \) identical machines \( M_1, M_2, \ldots, M_m \)
- **Goal**: Schedule the jobs on the machines minimizing the makespan \( C_{\text{max}} = \max_{1 \leq j \leq n} C_j \), where \( C_k \) is the completion time of job \( J_k \).

- \( J_1: p_1 = 2 \)
- \( J_2: p_2 = 12 \)
- \( J_3: p_3 = 6 \)
- \( J_4: p_4 = 4 \)
Parallel Machine Scheduling

Given: \( n \) jobs \( J_1, J_2, \ldots, J_n \) with processing times \( p_1, p_2, \ldots, p_n \), and \( m \) identical machines \( M_1, M_2, \ldots, M_m \)

Goal: Schedule the jobs on the machines minimizing the makespan \( C_{\text{max}} = \max_{1 \leq j \leq n} C_j \), where \( C_k \) is the completion time of job \( J_k \).

- \( J_1: p_1 = 2 \)
- \( J_2: p_2 = 12 \)
- \( J_3: p_3 = 6 \)
- \( J_4: p_4 = 4 \)
Parallel Machine Scheduling

Machine Scheduling Problem

- **Given:** \( n \) jobs \( J_1, J_2, \ldots, J_n \) with processing times \( p_1, p_2, \ldots, p_n \), and \( m \) identical machines \( M_1, M_2, \ldots, M_m \)
- **Goal:** Schedule the jobs on the machines minimizing the makespan \( C_{\text{max}} = \max_{1 \leq j \leq n} C_j \), where \( C_k \) is the completion time of job \( J_k \).

- \( J_1: p_1 = 2 \)
- \( J_2: p_2 = 12 \)
- \( J_3: p_3 = 6 \)
- \( J_4: p_4 = 4 \)
Parallel Machine Scheduling

Machine Scheduling Problem

- **Given:** $n$ jobs $J_1, J_2, \ldots, J_n$ with processing times $p_1, p_2, \ldots, p_n$, and $m$ identical machines $M_1, M_2, \ldots, M_m$

- **Goal:** Schedule the jobs on the machines minimizing the makespan $C_{\text{max}} = \max_{1 \leq j \leq n} C_j$, where $C_k$ is the completion time of job $J_k$.

For the analysis, it will be convenient to denote by $C_i$ the completion time of a machine $i$.
NP-Completeness of Parallel Machine Scheduling

Lemma

Parallel Machine Scheduling is NP-complete even if there are only two machines.

Proof Idea: Polynomial time reduction from NUMBER-PARTITIONING.
Parallel Machine Scheduling is NP-complete even if there are only two machines.

**Proof Idea:** Polynomial time reduction from NUMBER-PARTITIONING.
NP-Completeness of Parallel Machine Scheduling

**Lemma**

Parallel Machine Scheduling is NP-complete even if there are only two machines.

**Proof Idea:** Polynomial time reduction from NUMBER-PARTITIONING.

List Scheduling \((J_1, J_2, \ldots, J_n, m)\)

1. **while** there exists an unassigned job
2. Schedule job on the machine with the least load
Parallel Machine Scheduling is NP-complete even if there are only two machines.

Proof Idea: Polynomial time reduction from NUMBER-PARTITIONING.

Equivalent to the following Online Algorithm [CLRS3]:
Whenever a machine is idle, schedule the next job on that machine.

List Scheduling($J_1, J_2, \ldots, J_n, m$)
1: while there exists an unassigned job
2: Schedule job on the machine with the least load
Parallel Machine Scheduling is NP-complete even if there are only two machines.

**Proof Idea:** Polynomial time reduction from \textsc{Number-Partitioning}.

Equivalent to the following \textit{Online Algorithm} [CLRS3]:
Whenever a machine is idle, schedule the next job on that machine.

**List Scheduling** ($J_1, J_2, \ldots, J_n, m$)
1. \textbf{while} there exists an unassigned job
2. Schedule job on the machine with the least load

How good is this most basic Greedy Approach?
List Scheduling Analysis (Observations)

a. The optimal makespan is at least as large as the greatest processing time, that is, 
   \[ C^*_{\max} \geq \max_{1 \leq k \leq n} p_k. \]

b. The optimal makespan is at least as large as the average machine load, that is, 
   \[ C^*_{\max} \geq \frac{1}{m} \sum_{k=1}^{n} p_k. \]

Proof:

b. The total processing times of all \( n \) jobs equals 
   \[ \sum_{k=1}^{n} p_k \Rightarrow \text{One machine must have a load of at least} \]
   \[ \frac{1}{m} \cdot \sum_{k=1}^{n} p_k. \]
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

\[ C^*_\text{max} \geq \max_{1 \leq k \leq n} p_k. \]

Ex 35-5 a.&b.
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_\text{max} \geq \max_{1 \leq k \leq n} p_k.$$ 

b. The optimal makespan is at least as large as the average machine load, that is,

$$C^*_\text{max} \geq \frac{1}{m} \sum_{k=1}^{n} p_k.$$
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

\[ C^*_\text{max} \geq \max_{1 \leq k \leq n} p_k. \]

b. The optimal makespan is at least as large as the average machine load, that is,

\[ C^*_\text{max} \geq \frac{1}{m} \sum_{k=1}^{n} p_k. \]

Proof:
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

\[ C^*_\text{max} \geq \max_{1 \leq k \leq n} p_k. \]

b. The optimal makespan is at least as large as the average machine load, that is,

\[ C^*_\text{max} \geq \frac{1}{m} \sum_{k=1}^{n} p_k. \]

Proof:

b. The total processing times of all \( n \) jobs equals \( \sum_{k=1}^{n} p_k \)
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

\[ C^*_{\text{max}} \geq \max_{1 \leq k \leq n} p_k. \]

b. The optimal makespan is at least as large as the average machine load, that is,

\[ C^*_{\text{max}} \geq \frac{1}{m} \sum_{k=1}^{n} p_k. \]

Proof:

b. The total processing times of all \( n \) jobs equals \( \sum_{k=1}^{n} p_k \)

\[ \Rightarrow \text{One machine must have a load of at least} \quad \frac{1}{m} \cdot \sum_{k=1}^{n} p_k \]
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.
Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \).
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)

![Diagram showing machine scheduling with last job \( J_i \) scheduled on machine \( M_j \).]
Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[
C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k.
\]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \]

Using Ex 35-5 a. & b.

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k \]
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k = \frac{1}{m} \sum_{k=1}^{n} p_k \]

Using Ex 35-5 a. & b.
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:

- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k = \frac{1}{m} \sum_{k=1}^{n} p_k \implies C_j \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k
\]
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:

- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k = \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k \]

Using Ex 35-5 a. & b.
For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

**Proof:**

- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k = \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k \leq 2 \cdot C_{\text{\scriptsize max}} \]

Using Ex 35-5 a. & b.
Analysis can be shown to be almost tight. Is there a better algorithm?
Improving Greedy

Analysis can be shown to be almost tight. Is there a better algorithm?

The problem of the List-Scheduling Approach were the large jobs.
Improving Greedy

Analysis can be shown to be almost tight. Is there a better algorithm?

The problem of the List-Scheduling Approach were the large jobs

LEAST PROCESSING TIME($J_1, J_2, \ldots, J_n, m$)
1: Sort jobs decreasingly in their processing times
2: for $i = 1$ to $m$
3: \hspace{1em} $C_i = 0$
4: \hspace{1em} $S_i = \emptyset$
5: \hspace{1em} end for
6: for $j = 1$ to $n$
7: \hspace{1em} $i = \operatorname{argmin}_{1 \leq k \leq m} C_k$
8: \hspace{1em} $S_i = S_i \cup \{j\}, C_i = C_i + p_j$
9: \hspace{1em} end for
10: return $S_1, \ldots, S_m$
Analysis can be shown to be almost tight. Is there a better algorithm?

**Least Processing Time** \((J_1, J_2, \ldots, J_n, m)\)

1. Sort jobs decreasingly in their processing times
2. for \(i = 1\) to \(m\)
3. \(C_i = 0\)
4. \(S_i = \emptyset\)
5. end for
6. for \(j = 1\) to \(n\)
7. \(i = \arg\min_{1 \leq k \leq m} C_k\)
8. \(S_i = S_i \cup \{j\}, C_i = C_i + p_j\)
9. end for
10. return \(S_1, \ldots, S_m\)

**Runtime:**

\(O(n \log n)\) for sorting
\(O(n \log m)\) for extracting (and re-inserting) the minimum (use priority queue).
Improving Greedy

Analysis can be shown to be almost tight. Is there a better algorithm?

Least Processing Time ($J_1, J_2, \ldots, J_n, m$)

1: Sort jobs decreasingly in their processing times
2: for $i = 1$ to $m$
3: \hspace{1em} $C_i = 0$
4: \hspace{1em} $S_i = \emptyset$
5: end for
6: for $j = 1$ to $n$
7: \hspace{1em} $i = \arg\min_{1 \leq k \leq m} C_k$
8: \hspace{1em} $S_i = S_i \cup \{j\}$, $C_i = C_i + p_j$
9: end for
10: return $S_1, \ldots, S_m$

Runtime:

- $O(n \log n)$ for sorting

The problem of the List-Scheduling Approach were the large jobs
Improving Greedy

The problem of the List-Scheduling Approach were the large jobs.

Analysis can be shown to be almost tight. Is there a better algorithm?

**Least Processing Time** \( (J_1, J_2, \ldots, J_n, m) \)

1. Sort jobs decreasingly in their processing times
2. for \( i = 1 \) to \( m \)
3. \( C_i = 0 \)
4. \( S_i = \emptyset \)
5. end for
6. for \( j = 1 \) to \( n \)
7. \( i = \arg\min_{1 \leq k \leq m} C_k \)
8. \( S_i = S_i \cup \{j\}, C_i = C_i + p_j \)
9. end for
10. return \( S_1, \ldots, S_m \)

**Runtime:**
- \( O(n \log n) \) for sorting
- \( O(n \log m) \) for extracting (and re-inserting) the minimum (use priority queue).

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
Analysis of Improved Greedy

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

This can be shown to be tight (see next slide).
Analysis of Improved Greedy

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof (of approximation ratio 3/2).
The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof (of approximation ratio $3/2$).

- **Observation 1**: If there are at most $m$ jobs, then the solution is optimal.
Analysis of Improved Greedy

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$. Graham 1966

Proof (of approximation ratio 3/2).

- **Observation 1**: If there are at most $m$ jobs, then the solution is optimal.
- **Observation 2**: If there are more than $m$ jobs, then $C_{\text{max}}^* \geq 2 \cdot p_{m+1}$.
Analysis of Improved Greedy

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{(3m)}$.

Proof (of approximation ratio $3/2$).

- Observation 1: If there are at most $m$ jobs, then the solution is optimal.
- Observation 2: If there are more than $m$ jobs, then $C^*_\text{max} \geq 2 \cdot p_{m+1}$.
- As in the analysis for list scheduling

![Diagram](image)
Analysis of Improved Greedy

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof (of approximation ratio 3/2).

- **Observation 1**: If there are at most $m$ jobs, then the solution is optimal.
- **Observation 2**: If there are more than $m$ jobs, then $C^*_{\text{max}} \geq 2 \cdot p_{m+1}$.
- As in the analysis for list scheduling, we have

$$C_{\text{max}} = C_j = (C_j - p_i) + p_i$$
Analysis of Improved Greedy

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof (of approximation ratio $3/2$).

- **Observation 1:** If there are at most $m$ jobs, then the solution is optimal.
- **Observation 2:** If there are more than $m$ jobs, then $C^*_\text{max} \geq 2 \cdot p_{m+1}$.
- As in the analysis for list scheduling, we have

  \[ C^*_\text{max} = C_j = (C_j - p_i) + p_i \leq C^*_\text{max} + \frac{1}{2} C^*_\text{max} \]

This is for the case $i \geq m + 1$ (otherwise, an even stronger inequality holds)
Analysis of Improved Greedy

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof (of approximation ratio 3/2).
- **Observation 1**: If there are at most $m$ jobs, then the solution is optimal.
- **Observation 2**: If there are more than $m$ jobs, then $C^*_\text{max} \geq 2 \cdot p_{m+1}$.
- As in the analysis for list scheduling, we have

\[
C^*_\text{max} = C_j = (C_j - p_i) + p_i \leq C^*_\text{max} + \frac{1}{2} C^*_\text{max} = \frac{3}{2} C^*_\text{max}.
\]

IV. Approximation via Exact Algorithms
Parallel Machine Scheduling

18
The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$. 

IV. Approximation via Exact Algorithms 
Parallel Machine Scheduling
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Proof of an instance which shows tightness:
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$. 

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

```
M_5
M_4
M_3 8
M_2 9
M_1 9
```

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{(3m)} \).

Proof of an instance which shows tightness:
- \( m \) machines and \( n = 2m + 1 \) jobs:
  - two of length \( 2m - 1 \), \( 2m - 2 \), \ldots, \( m \) and one extra job of length \( m \)

\[
m = 5, \quad n = 11:
\]

<table>
<thead>
<tr>
<th>Machine</th>
<th>Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_5 )</td>
<td>7</td>
</tr>
<tr>
<td>( M_4 )</td>
<td>8</td>
</tr>
<tr>
<td>( M_3 )</td>
<td>8</td>
</tr>
<tr>
<td>( M_2 )</td>
<td>9</td>
</tr>
<tr>
<td>( M_1 )</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ C^\text{max} = 15 \]

\[ C^* \text{max} = 19 \]

IV. Approximation via Exact Algorithms
Parallel Machine Scheduling
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$. Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

M_5: \hspace{1cm} M_4: \hspace{1cm} M_3: \hspace{1cm} M_2: \hspace{1cm} M_1:

\begin{array}{c|c|c|c|c}
0 & 1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 \\
15 & 16 & 17 & 18 & 19 \\
20 & & & & \\
\end{array}

C^{\ast\max} = 15, C^{\max} = 19

LPT gives $C^{\max} = 19$

Optimum is $C^{\ast\max} = 15$
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

\[ C_{\text{max}} = 19 \]

Optimum is $C^*_{\text{max}} = 15$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Graham 1966

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[ m = 5, \ n = 11 : \]

```
M_5: 7 7
M_4: 8 6
M_3: 8 6
M_2: 9 5
M_1: 9
```

IV. Approximation via Exact Algorithms
Parallel Machine Scheduling
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

```
M_5: 7 7
M_4: 8 6
M_3: 8 6
M_2: 9 5
M_1: 9 5
```
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of \(\frac{4}{3} - \frac{1}{3m}\).

Proof of an instance which shows tightness:

- \(m\) machines and \(n = 2m + 1\) jobs:
- two of length \(2m - 1, 2m - 2, \ldots, m\) and one extra job of length \(m\)

\[m = 5, \ n = 11:\]

\[M_1 \quad M_2 \quad M_3 \quad M_4 \quad M_5\]

<table>
<thead>
<tr>
<th>Time</th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
<th>Job 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11:$

```
M_5: 7 7
M_4: 8 6
M_3: 8 6
M_2: 9 5
M_1: 9 5
```

$C_{max} = 19$
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{(3m)} \).

Graham 1966

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[
\begin{align*}
m = 5, \ n = 11 : \\
LPT \ gives \ C_{\text{max}} = 19
\end{align*}
\]
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{(3m)}$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Graham 1966

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[
m = 5, \ n = 11:
\]

LPT gives \( C_{\text{max}} = 19 \)

\[
M_5
M_4
M_3
M_2
M_1
\]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

7 7 6 6 5 5 5
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Graham 1966

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^{\ast}_{\text{max}} = 15$
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[ m = 5, \ n = 11 : \]

LPT gives \( C_{\text{max}} = 19 \)

Optimum is \( C^*_{\text{max}} = 15 \)

IV. Approximation via Exact Algorithms
Parallel Machine Scheduling
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:
- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^{*}_{\text{max}} = 15$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Graham 1966

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[
m = 5, \ n = 11 : \quad \text{LPT gives } C_{\text{max}} = 19
\]
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \( \frac{4}{3} - \frac{1}{3m} \).

Graham 1966

Proof of an instance which shows tightness:

- \( m \) machines and \( n = 2m + 1 \) jobs:
- two of length \( 2m - 1, 2m - 2, \ldots, m \) and one extra job of length \( m \)

\[
m = 5, \ n = 11:
\]

LPT gives \( C_{\text{max}} = 19 \)

IV. Approximation via Exact Algorithms

Parallel Machine Scheduling
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of \(4/3 - 1/(3m)\).

Graham 1966

The LPT algorithm has an approximation ratio of \(4/3 - 1/(3m)\).

Proof of an instance which shows tightness:

- \(m\) machines and \(n = 2m + 1\) jobs:
- two of length \(2m - 1, 2m - 2, \ldots, m\) and one extra job of length \(m\)

\[
m = 5, n = 11: \quad \text{LPT gives } C_{\text{max}} = 19
\]
Tightness of the Bound for LPT

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Graham 1966

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$$m = 5, \ n = 11 : \quad \text{LPT gives } C_{\text{max}} = 19$$

$$C^*_{\text{max}} = 15$$
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of \(4/3 - 1/(3m)\).

Proof of an instance which shows tightness:

- \(m\) machines and \(n = 2m + 1\) jobs:
- two of length \(2m - 1, 2m - 2, \ldots, m\) and one extra job of length \(m\)

\[m = 5, n = 11:\]

LPT gives \(C_{\text{max}} = 19\)
Optimum is \(C^*_{\text{max}} = 15\)

![Diagram showing scheduling with machines and jobs](image-url)
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines and $n = 2m + 1$ jobs:
- two of length $2m - 1, 2m - 2, \ldots, m$ and one extra job of length $m$

$m = 5$, $n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
Conclusion

List scheduling has an approximation ratio of 2.

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

IV. Approximation via Exact Algorithms

Parallel Machine Scheduling
**Conclusion**

- **Graham 1966**
  - List scheduling has an approximation ratio of 2.

- **Graham 1966**
  - The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

- **Theorem (Hochbaum, Shmoys’87)**
  - There exists a PTAS for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$. 
Conclusion

Graham 1966

List scheduling has an approximation ratio of 2.

Graham 1966

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{(3m)}$.

Theorem (Hochbaum, Shmoys’87)

There exists a PTAS for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$.

Can we find a FPTAS (for polynomially bounded processing times)?
Conclusion

List scheduling has an approximation ratio of 2.

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Graham 1966

There exists a PTAS for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$.

Theorem (Hochbaum, Shmoys’87)

Can we find a FPTAS (for polynomially bounded processing times)?

No!

IV. Approximation via Exact Algorithms Parallel Machine Scheduling
**Conclusion**

- **Graham 1966**
  - List scheduling has an approximation ratio of 2.

- **Graham 1966**
  - The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

- **Theorem (Hochbaum, Shmoys’87)**
  - There exists a PTAS for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$.

Can we find a FPTAS (for polynomially bounded processing times)? **No!**

Because for sufficiently small approximation ratio $1 + \epsilon$, the computed solution has to be optimal, and Parallel Machine Scheduling is strongly NP-hard.
Exercise (easy): Run the LPT algorithm on three machines and jobs having processing times \{3, 4, 4, 3, 5, 3, 5\}. Which allocation do you get?

1. [3, 3, 5], [4, 5], [4, 3]
2. [5, 3], [5, 4], [4, 3, 3]
3. [3, 3, 3], [5, 4], [5, 4]
Outline

The Subset-Sum Problem

Parallel Machine Scheduling

Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Basic Idea: For \((1 + \epsilon)-\)approximation, don’t have to work with exact \(p_k\)’s.
A PTAS for Parallel Machine Scheduling

Basic Idea: For $(1 + \epsilon)$-approximation, don’t have to work with exact $p_k$’s.

SUBROUTINE($J_1, J_2, \ldots, J_n, m, T$)
1: Either: Return a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$
2: Or: Return there is no solution with makespan $< T$

IV. Approximation via Exact Algorithms
Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

**SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C_{\text{max}}^*\}\)
2. Or: **Return** there is no solution with makespan \(< T\)

**Key Lemma**

**SUBROUTINE** can be implemented in time \(n^{O(1/\epsilon^2)}\).
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

\textbf{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)

1: Either: \textbf{Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_\text{max}\}\)

2: Or: \textbf{Return} there is no solution with makespan \(< T\)

Key Lemma

We will prove this on the next slides.

\textbf{SUBROUTINE} can be implemented in time \(n^{O(1/\epsilon^2)}\).
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

**SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\)
1. Either: Return a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C_{\text{max}}^{*}\}\)
2. Or: Return there is no solution with makespan \(< T\)

**Key Lemma**

We will prove this on the next slides.

**SUBROUTINE** can be implemented in time \(n^{O(1/\epsilon^2)}\).

**Theorem (Hochbaum, Shmoys’87)**

There exists a **PTAS** for Parallel Machine Scheduling which runs in time 
\(O(n^{O(1/\epsilon^2)} \cdot \log P)\), where \(P := \sum_{k=1}^{n} p_k\).
A PTAS for Parallel Machine Scheduling

Basic Idea: For $(1 + \epsilon)$-approximation, don’t have to work with exact $p_k$'s.

**SUBROUTINE**($J_1, J_2, \ldots, J_n, m, T$)
1: Either: **Return** a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$
2: Or: **Return** there is no solution with makespan $< T$

**Key Lemma**

We will prove this on the next slides.

SUBROUTINE can be implemented in time $n^{O(1/\epsilon^2)}$.

**Theorem (Hochbaum, Shmoys'87)**

There exists a **PTAS** for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$.

**Proof (using Key Lemma):**

PTAS($J_1, J_2, \ldots, J_n, m$)
1: Do binary search to find smallest $T$ s.t. $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$.
2: **Return** solution computed by **SUBROUTINE**($J_1, J_2, \ldots, J_n, m, T$)
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

**SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_\text{max}\}\)
2. Or: **Return** there is no solution with makespan \(< T\)

Key Lemma

We will prove this on the next slides.

**SUBROUTINE** can be implemented in time \(n^{O(1/\epsilon^2)}\).

**Theorem (Hochbaum, Shmoys’87)**

There exists a PTAS for Parallel Machine Scheduling which runs in time \(O(n^{O(1/\epsilon^2)} \cdot \log P)\), where \(P := \sum_{k=1}^{n} p_k\).

Proof (using Key Lemma):

**PTAS**\((J_1, J_2, \ldots, J_n, m)\)

1. Do binary search to find smallest \(T\) s.t. \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_\text{max}\}\).
2. **Return** solution computed by **SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\).

Since \(0 \leq C^*_\text{max} \leq P\) and \(C^*_\text{max}\) is integral, binary search terminates after \(O(\log P)\) steps.
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

**SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)
2. Or: **Return** there is no solution with makespan < \(T\)

**Key Lemma**

We will prove this on the next slides.

**SUBROUTINE** can be implemented in time \(n^{O(1/\epsilon^2)}\).

**Theorem (Hochbaum, Shmoys’87)**

There exists a **PTAS** for Parallel Machine Scheduling which runs in time \(O(n^{O(1/\epsilon^2)} \cdot \log P)\), where \(P := \sum_{k=1}^{n} p_k\).

**Proof (using Key Lemma):**

**PTAS**\((J_1, J_2, \ldots, J_n, m)\)

1. Do binary search to find smallest \(T\) s.t. \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).
2. **Return** solution computed by **SUBROUTINE**\((J_1, J_2, \ldots, J_n, m, T)\)

Since \(0 \leq C^*_{\text{max}} \leq P\) and \(C^*_{\text{max}}\) is integral, binary search terminates after \(O(\log P)\) steps.
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)\)-approximation, don’t have to work with exact \(p_k\)’s.

\text{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)
1: Either: \textbf{Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)
2: Or: \textbf{Return} there is no solution with makespan < \(T\)

Key Lemma
We will prove this on the next slides.

\text{SUBROUTINE} can be implemented in time \(n^{O(1/\epsilon^2)}\).

Theorem (Hochbaum, Shmoys’87)
There exists a PTAS for Parallel Machine Scheduling which runs in time \(O(n^{O(1/\epsilon^2)} \cdot \log P)\), where \(P := \sum_{k=1}^{n} p_k\).

Proof (using Key Lemma):
PTAS\((J_1, J_2, \ldots, J_n, m)\)
1: Do binary search to find smallest \(T\) s.t. \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).
2: \textbf{Return} solution computed by \text{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)

Since \(0 \leq C^*_{\text{max}} \leq P\) and \(C^*_{\text{max}}\) is integral, binary search terminates after \(O(\log P)\) steps.
SUBROUTINE($J_1, J_2, \ldots, J_n, m, T$)

1: Either: Return a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^{*}_{\text{max}}\}$

2: Or: Return there is no solution with makespan $< T$
SUBROUTINE \((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: \textbf{Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)
2. Or: \textbf{Return} there is no solution with makespan < \(T\)

**Observation**

Divide jobs into two groups: \(J_{\text{small}} = \{i: p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = [n] \setminus J_{\text{small}}\). Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).
Implementation of Subroutine

\textbf{SUBROUTINE} \((J_1, J_2, \ldots, J_n, m, T)\)

1: \textbf{Either: Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)

2: \textbf{Or: Return} there is no solution with makespan \(< T\)

\textbf{Observation}

\textbf{Divide} jobs into two groups: \(J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = [n] \setminus J_{\text{small}}\). Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then \textbf{greedily placing} \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).

\textbf{Proof:}
Implementation of Subroutine

\textbf{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)

1: Either: \textbf{Return} a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$
2: Or: \textbf{Return} there is no solution with makespan $< T$

Observation

\textbf{Divide} jobs into two groups: $J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}$ and $J_{\text{large}} = [n] \setminus J_{\text{small}}$. Given a solution for $J_{\text{large}}$ only with makespan $(1 + \epsilon) \cdot T$, then \textbf{greedily} placing $J_{\text{small}}$ yields a solution with makespan $(1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$.

Proof:

- Let $M_j$ be the machine with largest load
Implementation of Subroutine

**SUBROUTINE**$(J_1, J_2, \ldots, J_n, m, T)$

1. Either: **Return** a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$
2. Or: **Return** there is no solution with makespan $< T$

---

**Observation**

**Divide** jobs into two groups: $J_{\text{small}} = \{i: p_i \leq \epsilon \cdot T\}$ and $J_{\text{large}} = [n] \setminus J_{\text{small}}$. Given a solution for $J_{\text{large}}$ only with makespan $(1 + \epsilon) \cdot T$, then **greedily** placing $J_{\text{small}}$ yields a solution with makespan $(1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$.

**Proof:**

- Let $M_j$ be the machine with largest load
- If there are no jobs from $J_{\text{small}}$, then makespan is at most $(1 + \epsilon) \cdot T$. 

---

IV. Approximation via Exact Algorithms Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Implementation of Subroutine

\textbf{SUBROUTINE}\((J_1, J_2, \ldots, J_n, m, T)\)

1: Either: \textbf{Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)

2: Or: \textbf{Return} there is no solution with makespan \(< T\)

\begin{tcolorbox}
\underline{Observation} \hspace{1cm}

Divide jobs into two groups: \(J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = [n] \setminus J_{\text{small}}\). Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).

\end{tcolorbox}

\textbf{Proof:}

- Let \(M_j\) be the machine with largest load
- If there are no jobs from \(J_{\text{small}}\), then makespan is at most \((1 + \epsilon) \cdot T\).
- Otherwise, let \(i \in J_{\text{small}}\) be the last job added to \(M_j\).


**Implementation of Subroutine**

\[
\text{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)
\]

1. **Either**: \textbf{Return} a solution with \( C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\} \)
2. **Or**: \textbf{Return} there is no solution with makespan \( < T \)

---

**Observation**

Divide jobs into two groups: \( J_{\text{small}} = \{ i : p_i \leq \epsilon \cdot T \} \) and \( J_{\text{large}} = [n] \setminus J_{\text{small}} \). Given a solution for \( J_{\text{large}} \) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \( J_{\text{small}} \) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).  

**Proof:**

- Let \( M_j \) be the machine with largest load
- If there are no jobs from \( J_{\text{small}} \), then makespan is at most \((1 + \epsilon) \cdot T\).
- Otherwise, let \( i \in J_{\text{small}} \) be the last job added to \( M_j \).

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k
\]

the “well-known” formula
Implementation of Subroutine

**SUBROUTINE (J₁, J₂, ..., Jₙ, m, T)**

1. Either: **Return** a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$
2. Or: **Return** there is no solution with makespan $< T$

---

**Observation**

Divide jobs into two groups: $J_{\text{small}} = \{i: p_i \leq \epsilon \cdot T\}$ and $J_{\text{large}} = [n] \setminus J_{\text{small}}$. Given a solution for $J_{\text{large}}$ only with makespan $(1 + \epsilon) \cdot T$, then greedily placing $J_{\text{small}}$ yields a solution with makespan $(1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$.

---

**Proof:**

- Let $M_j$ be the machine with largest load
- If there are no jobs from $J_{\text{small}}$, then makespan is at most $(1 + \epsilon) \cdot T$.
- Otherwise, let $i \in J_{\text{small}}$ be the last job added to $M_j$.

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow
\]

the “well-known” formula
Implementation of Subroutine

**SUBROUTINE** \((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)
2. Or: **Return** there is no solution with makespan < \(T\)

---

**Observation**

Divide jobs into two groups: \(J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = [n] \setminus J_{\text{small}}\). Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).

---

**Proof:**

- Let \(M_J\) be the machine with largest load
- If there are no jobs from \(J_{\text{small}}\), then makespan is at most \((1 + \epsilon) \cdot T\).
- Otherwise, let \(i \in J_{\text{small}}\) be the last job added to \(M_J\).

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq p_i + \frac{1}{m} \sum_{k=1}^{n} p_k
\]

the “well-known” formula
Implementation of Subroutine

\textbf{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)

1: Either: \textbf{Return} a solution with $C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$

2: Or: \textbf{Return} there is no solution with makespan $< T$

---

Observation

Divide jobs into two groups: $J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}$ and $J_{\text{large}} = [n] \setminus J_{\text{small}}$. Given a solution for $J_{\text{large}}$ only with makespan $(1 + \epsilon) \cdot T$, then greedily placing $J_{\text{small}}$ yields a solution with makespan $(1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}$.

---

Proof:

- Let $M_j$ be the machine with largest load
- If there are no jobs from $J_{\text{small}}$, then makespan is at most $(1 + \epsilon) \cdot T$.
- Otherwise, let $i \in J_{\text{small}}$ be the last job added to $M_j$.

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq p_i + \frac{1}{m} \sum_{k=1}^{n} p_k \leq \epsilon \cdot T + C^*_{\text{max}}
\]

the “well-known” formula
Implementation of Subroutine

SUBROUTINE(\(J_1, J_2, \ldots, J_n, m, T\))

1: Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C_{\text{max}}^*\}\)

2: Or: **Return** there is no solution with makespan \(< T\)

---

**Observation**

Divide jobs into two groups: \(J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = [n] \setminus J_{\text{small}}\).

Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C_{\text{max}}^*\}\).

**Proof:**

- Let \(M_j\) be the machine with largest load
- If there are no jobs from \(J_{\text{small}}\), then makespan is at most \((1 + \epsilon) \cdot T\).
- Otherwise, let \(i \in J_{\text{small}}\) be the last job added to \(M_j\).

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k 
\Rightarrow 
C_j \leq p_i + \frac{1}{m} \sum_{k=1}^{n} p_k
\leq \epsilon \cdot T + C_{\text{max}}^*
\leq (1 + \epsilon) \cdot \max\{T, C_{\text{max}}^*\}
\]

the “well-known” formula
Implementation of Subroutine

\texttt{SUBROUTINE}(J_1, J_2, \ldots, J_n, m, T)

1: Either: \textbf{Return} a solution with \( C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\} \)
2: Or: \textbf{Return} there is no solution with makespan < \( T \)

---

**Observation**

Divide jobs into two groups: \( J_{\text{small}} = \{i : p_i \leq \epsilon \cdot T\} \) and \( J_{\text{large}} = [n] \setminus J_{\text{small}} \).

Given a solution for \( J_{\text{large}} \) only with makespan \( (1 + \epsilon) \cdot T \), then greedily placing \( J_{\text{small}} \) yields a solution with makespan \( (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\} \).

**Proof:**

- Let \( M_j \) be the machine with largest load
- If there are no jobs from \( J_{\text{small}} \), then makespan is at most \( (1 + \epsilon) \cdot T \).
- Otherwise, let \( i \in J_{\text{small}} \) be the last job added to \( M_j \).

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq p_i + \frac{1}{m} \sum_{k=1}^{n} p_k \\
\leq \epsilon \cdot T + C^*_{\text{max}} \\
\leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}
\]

the “well-known” formula
Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$. 

Can assume there are no jobs with $p_j \geq T$!

Assign some jobs to one machine, and then use as few machines as possible for the rest.
Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. 

IV. Approximation via Exact Algorithms Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$
Proof of Key Lemma (non-examinable)

Use **Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$.

\[\begin{align*}
\epsilon &= 0.5 \\
b &= 2
\end{align*}\]
Proof of Key Lemma (non-examinable)

Use **Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$.

```latex
\begin{align*}
\epsilon &= 0.5 \\
\text{b} &= 2
\end{align*}
```

- Assign some jobs to one machine, and then use as few machines as possible for the rest.
Use Dynamic Programming to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T\).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left\lceil \frac{p_i b^2}{T} \right\rceil \cdot \frac{T}{b^2} \).
Proof of Key Lemma (non-examinable)

Use **Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$.
Use **Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$.
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$

$\Rightarrow$ Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$

Can assume there are no jobs with $p_j \geq T!$

IV. Approximation via Exact Algorithms Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T\).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left\lfloor \frac{p_i b^2}{T} \right\rfloor \cdot \frac{T}{b^2} \)

\[ \Rightarrow \] Every \( p'_i = \alpha \cdot \frac{T}{b^2} \) for \( \alpha = b, b+1, \ldots, b^2 \)

- Let \( C \) be all \( (s_b, s_{b+1}, \ldots, s_{b^2}) \) with \( \sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T \).

IV. Approximation via Exact Algorithms

Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)

\[ \begin{align*}
1.5 \cdot T & \quad 1.5 \cdot T \\
1.25 \cdot T & \quad 1.25 \cdot T \\
1 \cdot T & \quad 1 \cdot T \\
0.75 \cdot T & \quad 0.75 \cdot T \\
0.5 \cdot T & \quad 0.5 \cdot T \\
0.25 \cdot T & \quad 0.25 \cdot T \\
0 & \quad 0
\end{align*} \]

\( J_{\text{large}} \) \hspace{2cm} \( J_{\text{small}} \)

\( \epsilon = 0.5 \)
\( b = 2 \)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \lceil \frac{p_i b^2}{T} \rceil \cdot \frac{T}{b^2}$

  ⇒ Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$

- Let $\mathcal{C}$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T$.

Assignments to one machine with makespan $\leq T$.

IV. Approximation via Exact Algorithms Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Proof of Key Lemma (non-examinable)

Use **Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p_i' = \left\lceil \frac{p_i b^2}{T} \right\rceil \cdot \frac{T}{b^2}$
  - Every $p_i' = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$
- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T$.
- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the **minimum number of machines** required to schedule all jobs with makespan $\leq T$:

![Diagram](image-url)

- $\epsilon = 0.5$
- $b = 2$

IV. Approximation via Exact Algorithms  
**Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)**
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_i b^2}{T} \right\rceil \cdot \frac{T}{b^2}$

  $\Rightarrow$ Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b+1, \ldots, b^2$

- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T$.

- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the minimum number of machines required to schedule all jobs with makespan $\leq T$:

  $f(0, 0, \ldots, 0) = 0$
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T\).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2} \)

\[ \Rightarrow \] Every \( p'_i = \alpha \cdot \frac{T}{b^2} \) for \( \alpha = b, b+1, \ldots, b^2 \)

- Let \( C \) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \( \sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T \).

- Let \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \) be the minimum number of machines required to schedule all jobs with makespan \( \leq T \):

\[
f(0, 0, \ldots, 0) = 0 \]

\[
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

IV. Approximation via Exact Algorithms  
Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p_i' = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$

  $\Rightarrow$ Every $p_i' = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$

- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T$.

- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the minimum number of machines required to schedule all jobs with makespan $\leq T$:

  
  
  \[
  f(0, 0, \ldots, 0) = 0
  \]

  \[
  f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
  \]

IV. Approximation via Exact Algorithms

Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule \(J_{\text{large}}\) with makespan \((1 + \epsilon) \cdot T\).

- Let \(b\) be the smallest integer with \(1/b \leq \epsilon\). Define processing times \(p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}\)

\[\Rightarrow\quad \text{Every } p'_i = \alpha \cdot \frac{T}{b^2} \text{ for } \alpha = b, b + 1, \ldots, b^2\]

- Let \(C\) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \(\sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T\).

- Let \(f(n_b, n_{b+1}, \ldots, n_{b^2})\) be the minimum number of machines required to schedule all jobs with makespan \(\leq T\):

\[
f(0, 0, \ldots, 0) = 0
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

- Number of table entries is at most \(n^{b^2}\), hence filling all entries takes \(n^{O(b^2)}\)
Proof of Key Lemma (non-examinable)

Use Dynamic Programming to schedule \( J_{\text{large}} \) with makespan \( (1 + \epsilon) \cdot T \).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left[ \frac{p_i b^2}{T} \right] \cdot \frac{T}{b^2} \).

\[ \Rightarrow \] Every \( p'_i = \alpha \cdot \frac{T}{b^2} \) for \( \alpha = b, b + 1, \ldots, b^2 \).

- Let \( C \) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \( \sum_{i=1}^{b^2} s_i \cdot j \cdot \frac{T}{b^2} \leq T \).

- Let \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \) be the minimum number of machines required to schedule all jobs with makespan \( \leq T \):

\[
f(0, 0, \ldots, 0) = 0 \\
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

- The number of table entries is at most \( n^{b^2} \), hence filling all entries takes \( n^{O(b^2)} \).

- If \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m \) (for the jobs with \( p' \)), then return yes, otherwise no.

IV. Approximation via Exact Algorithms Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable)
Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2}$
- Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$
- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{j=1}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T$.
- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the minimum number of machines required to schedule all jobs with makespan $\leq T$:
  \[
  f(0, 0, \ldots, 0) = 0 \\
  f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
  \]
- Number of table entries is at most $n^{b^2}$, hence filling all entries takes $n^{O(b^2)}$
- If $f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m$ (for the jobs with $p'$), then return yes, otherwise no.
- As every machine is assigned at most $b$ jobs ($p'_i \geq \frac{T}{b}$) and the makespan is $\leq T$, the proof is complete.
Proof of Key Lemma (non-examinable)

Use **Dynamic Programming** to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T\).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p_i' = \left\lceil \frac{p_j b^2}{T} \right\rceil \cdot \frac{T}{b^2} \)

\[ \Rightarrow \] Every \( p_i' = \alpha \cdot \frac{T}{b^2} \) for \( \alpha = b, b + 1, \ldots, b^2 \)

- Let \( C \) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \( \sum_{j=s_b}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T \).

- Let \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \) be the minimum number of machines required to schedule all jobs with makespan \( \leq T \):

\[
f(0, 0, \ldots, 0) = 0 \\
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

- **Number of table entries is at most** \( n^{b^2} \), hence filling all entries takes \( n^{O(b^2)} \)
- If \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m \) (for the jobs with \( p' \)), then return yes, otherwise no.
- As every machine is assigned at most \( b \) jobs \( (p_i' \geq \frac{T}{b}) \) and the makespan is \( \leq T \),

\[
C_{\text{max}} \leq T + b \cdot \max_{i \in J_{\text{large}}} (p_i - p_i').
\]
Proof of Key Lemma (non-examinable)

**Use Dynamic Programming** to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \left\lceil \frac{p_i b^2}{T} \right\rceil \cdot \frac{T}{b^2}$.

  $\Rightarrow$ Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b+1, \ldots, b^2$.

- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_i \cdot j \cdot \frac{T}{b^2} \leq T$.

- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the minimum number of machines required to schedule all jobs with makespan $\leq T$:

  $f(0, 0, \ldots, 0) = 0$

  $f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2})$.

  - Number of table entries is at most $n^{b^2}$, hence filling all entries takes $n^{O(b^2)}$.
  - If $f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m$ (for the jobs with $p'$), then return yes, otherwise no.
  - As every machine is assigned at most $b$ jobs ($p'_i \geq \frac{T}{b}$) and the makespan is $\leq T$,

    $$C_{\text{max}} \leq T + b \cdot \max_{i \in J_{\text{large}}} (p_i - p'_i)$$

    $$\leq T + b \cdot \frac{T}{b^2}$$

IV. Approximation via Exact Algorithms  Bonus Material: A PTAS for Parallel Machine Scheduling (non-examinable) 25
Use **Dynamic Programming** to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T \).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left\lceil \frac{p_i b^2}{T} \right\rceil \cdot \frac{T}{b^2} \)

\[
\Rightarrow \quad \text{Every } p'_i = \alpha \cdot \frac{T}{b^2} \text{ for } \alpha = b, b + 1, \ldots, b^2
\]

- Let \( C \) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \( \sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T \).

- Let \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \) be the minimum number of machines required to schedule all jobs with makespan \( \leq T \):

\[
f(0, 0, \ldots, 0) = 0
\]

\[
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

- Number of table entries is at most \( n^{b^2} \), hence filling all entries takes \( n^{O(b^2)} \)
- If \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m \) (for the jobs with \( p' \)), then return yes, otherwise no.
- As every machine is assigned at most \( b \) jobs \( (p'_i \geq \frac{T}{b}) \) and the makespan is \( \leq T \),

\[
C_{\text{max}} \leq T + b \cdot \max_{i \in J_{\text{large}}} (p_i - p'_i)
\]

\[
\leq T + b \cdot \frac{T}{b^2} \leq (1 + \epsilon) \cdot T.
\]

\[
\square
\]
Outline

Introduction

General TSP

Metric TSP
Traveling Salesman Problem Introduction

33 city contest (1964)

Thus Flood realized that the Nearest Neighbor method is not a good estimate of the TSP but it created a decent first solution.

In 1962 a contest brought the TSP national recognition through a contest given by Proctor and Gamble. A flyer of the contest is pictured below.

The traveling salesman problem recently achieved national prominence when a soap company used it as the basis of a promotional contest. Prizes up to $10,000
532 cities (1987 [Padberg, Rinaldi])
13,509 cities (1999 [Applegate, Bixby, Chavatal, Cook])
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

---

Solution space consists of at most $n!$ possible tours. Actually the right number is $(n-1)!/2$. 

3

$1 + 2 + 3 + 4 = 9$

4

$1 + 2 + 1 + 3 = 8$

---

Metric TSP: costs satisfy triangle inequality: $\forall u, v, w \in V: c(u, w) \leq c(u, v) + c(v, w)$. 

Euclidean TSP: cities are points in the Euclidean space, costs are equal to their (rounded) Euclidean distance.
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal**: Find a hamiltonian cycle of $G$ with minimum cost.
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal**: Find a hamiltonian cycle of $G$ with minimum cost.
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given:** A complete undirected graph \( G = (V, E) \) with nonnegative integer cost \( c(u, v) \) for each edge \((u, v) \in E\)
- **Goal:** Find a hamiltonian cycle of \( G \) with minimum cost.

\[
3 + 2 + 1 + 3 = 9
\]
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal**: Find a hamiltonian cycle of $G$ with minimum cost.

Metric TSP: costs satisfy triangle inequality:

$$\forall u, v, w \in V: c(u, w) \leq c(u, v) + c(v, w).$$

Euclidean TSP: cities are points in the Euclidean space, costs are equal to their (rounded) Euclidean distance.

Special Instances

Even this version is NP hard (Ex. 35.2-2)
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given:** A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal:** Find a hamiltonian cycle of $G$ with minimum cost.

Solution space consists of at most $n!$ possible tours!
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph \( G = (V, E) \) with nonnegative integer cost \( c(u, v) \) for each edge \( (u, v) \in E \)
- **Goal**: Find a hamiltonian cycle of \( G \) with minimum cost.

Solution space consists of at most \( n! \) possible tours!

Actually the right number is \( (n - 1)!/2 \)

Special Instances

Even this version is \( NP \) hard (Ex. 35.2-2)
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph \( G = (V, E) \) with nonnegative integer cost \( c(u, v) \) for each edge \((u, v) \in E\)
- **Goal**: Find a hamiltonian cycle of \( G \) with minimum cost.

Solution space consists of at most \( n! \) possible tours!

Actually the right number is \((n - 1)!/2\)

Special Instances

Even this version is NP hard (Ex. 35.2-2)
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given:** A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal:** Find a hamiltonian cycle of $G$ with minimum cost.

Solution space consists of at most $n!$ possible tours!

Actually the right number is $(n - 1)!/2$

Special Instances

- **Metric TSP:** costs satisfy triangle inequality:

\[ \forall u, v, w \in V : \quad c(u, w) \leq c(u, v) + c(v, w). \]
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

**Formal Definition**

- **Given:** A complete undirected graph \( G = (V, E) \) with nonnegative integer cost \( c(u, v) \) for each edge \( (u, v) \in E \)
- **Goal:** Find a hamiltonian cycle of \( G \) with minimum cost.

Solution space consists of at most \( n! \) possible tours!

Actually the right number is \( (n - 1)!/2 \)

**Special Instances**

- **Metric TSP:** costs satisfy triangle inequality:

\[
\forall u, v, w \in V: \quad c(u, w) \leq c(u, v) + c(v, w).
\]

Even this version is NP hard (Ex. 35.2-2)
Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

**Formal Definition**

- **Given:** A complete undirected graph $G = (V, E)$ with nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$
- **Goal:** Find a hamiltonian cycle of $G$ with minimum cost.

Solution space consists of at most $n!$ possible tours!

Actually the right number is $(n - 1)!/2$

**Special Instances**

- **Metric TSP:** costs satisfy triangle inequality:
  \[
  \forall u, v, w \in V : \quad c(u, w) \leq c(u, v) + c(v, w).
  \]
- **Euclidean TSP:** cities are points in the Euclidean space, costs are equal to their (rounded) Euclidean distance

Even this version is NP hard (Ex. 35.2-2)
History of the TSP problem (1954)

Dantzig, Fulkerson and Johnson found an optimal tour through 42 cities.

http://www.math.uwaterloo.ca/tsp/history/img/dantzig_big.html
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable \( x(u, v) = 1 \) iff tour goes between \( u \) and \( v \))

V. Travelling Salesman Problem

Introduction
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable \( x(u, v) = 1 \) iff tour goes between \( u \) and \( v \))
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

V. Travelling Salesman Problem Introduction
The Dantzig-Fulkerson-Johnson Method

1. Create a **linear program** (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (**cutting plane**)

V. Travelling Salesman Problem Introduction

![Graph with constraints]

- $2x_1 - 9x_2 \leq -27$
- $\max \frac{1}{3}x + y$
- $4x_1 + 9x_2 \leq 36$
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

V. Travelling Salesman Problem Introduction
The Dantzig-Fulkerson-Johnson Method

1. Create a **linear program** (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (**cutting plane**)

V. Travelling Salesman Problem Introduction
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

![Graph showing linear program constraints and solution space]

- $2x_1 - 9x_2 \leq -27$
- $\max \frac{1}{3} x + y$
- $x_2 \leq 3$
- $4x_1 + 9x_2 \leq 36$

Additional constraint to cut the solution space of the LP
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$).
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane).

The graph illustrates the constraints and the feasible region of the linear program. The optimal solution is at $(2.25, 3)$, which satisfies the max $\frac{1}{3}x + y$ constraint and the additional constraint $2x_1 - 9x_2 \leq -27$.

Additional constraint to cut the solution space of the LP.
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, \nu) = 1$ iff tour goes between $u$ and $\nu$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

V. Travelling Salesman Problem Introduction
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

V. Travelling Salesman Problem

Introduction
Outline

Introduction

General TSP

Metric TSP
Hardness of Approximation

Theorem 35.3
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.
Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:


Let $G = (V, E)$ be an instance of the Hamiltonian-cycle problem. Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise.} \end{cases}$$

If $G$ has a Hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$. If $G$ does not have a Hamiltonian cycle, then any tour $T$ must use some edge $\notin E$, so

$$c(T) \geq (\rho |V| + 1) + (|V| - 1) = (\rho + 1)|V|.$$ 

Gap of $\rho + 1$ between tours which are using only edges in $G$ and those which don't.

$\rho$-Approximation of TSP in $G'$ computes Hamiltonian cycle in $G$ (if one exists).

Large weight will render this edge useless!

Can create representations of $G'$ and $c$ in time polynomial in $|V|$ and $|E|$. 

$G = (V, E)$

Reduction

$G' = (V, E')$

$1$

$1$

$1$

$\rho \cdot 4 + 1$

$1$

$\rho \cdot 4 + 1$
Hardness of Approximation

Theorem 35.3

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

Idea: Reduction from the hamiltonian-cycle problem.
Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:

Idea: Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem
**Theorem 35.3**

If \( P \neq \text{NP} \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Idea:** Reduction from the hamiltonian-cycle problem.

**Proof:**

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Theorem 35.3**

**Proof:**

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  
  $c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise.} \end{cases}$

  If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
  
  If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\notin E$.
  
  The gap of $\rho + 1$ between tours which are using only edges in $G$ and those which don't.

  A $\rho$-approximation of TSP in $G'$ computes hamiltonian cycle in $G$ (if one exists).

  Large weight will render this edge useless!

  Can create representations of $G'$ and $c$ in time polynomial in $|V|$ and $|E|$!
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise.} \end{cases}$$

If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$. If $G$ does not have a hamiltonian cycle, then any tour must use some edge $\not\in E$, so

$$c(T) \geq (\rho |V| + 1) + (|V| - 1) = (\rho + 1) |V|.$$
Hardness of Approximation

Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:


\textbf{Idea: Reduction from the hamiltonian-cycle problem.}

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

\[ c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho|V| + 1 & \text{otherwise}. \end{cases} \]

\[ G = (V, E) \]

\[ G' = (V, E') \]
Hardness of Approximation

**Theorem 35.3**

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Proof:**

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):

\[
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise.}
\end{cases}
\]

\[G = (V, E)\]

\[G' = (V, E')\]
Hardness of Approximation

**Theorem 35.3**

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Proof:**

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):

\[
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise}. 
\end{cases}
\]

\[
G = (V, E)
\]

\[
G' = (V, E')
\]

Large weight will render this edge useless!
If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Theorem 35.3**

**Proof:**

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):

\[
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise.}
\end{cases}
\]

This can create representations of \( G' \) and \( c \) in time polynomial in \( |V| \) and \( |E| \!),

\[
G = (V, E) \quad \quad G' = (V, E')
\]
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Theorem 35.3**

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  
  $$c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise.}
  \end{cases}$$

$$G = (V, E) \quad \text{Reduction} \quad G' = (V, E')$$
Hardness of Approximation

Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}
  \]
- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$. 

Reduction $G = (V, E) \rightarrow G' = (V, E')$ with weight $\rho \cdot 4 + 1$.
Hardness of Approximation

**Theorem 35.3**

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  
  $$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise.} \end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
Hardness of Approximation

**Theorem 35.3**

If P $\neq$ NP, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise}.
\end{cases}
$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.

![Diagram of the reduction from the hamiltonian-cycle problem to the TSP.](image)
Hardness of Approximation

Theorem 35.3

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

**Idea: Reduction from the hamiltonian-cycle problem.**

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a hamiltonian cycle \( H \), then \((G', c)\) contains a tour of cost \(|V|\)
Hardness of Approximation

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Proof:**

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a hamiltonian cycle \( H \), then \((G', c)\) contains a tour of cost \(|V|\).
- If \( G \) does not have a hamiltonian cycle, then any tour \( T \) must use some edge \( \not\in E \).

\[
G = (V, E) \quad \text{Reduction} \quad \rho \cdot 4 + 1 \quad G' = (V, E')
\]
Hardness of Approximation

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Theorem 35.3**

**Proof:**

*Idea: Reduction from the hamiltonian-cycle problem.*

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\notin E$, leading to a cost of $\rho \cdot 4 + 1$.
If P $\neq$ NP, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise.} \end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$. 
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\notin E$, and

$$\rho \cdot 4 + 1 \geq \rho \cdot 4 + 1 + (|V| - 1).$$
Hardness of Approximation

**Theorem 35.3**
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise.}
\end{cases}
$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\not\in E$.

V. Travelling Salesman Problem

General TSP
Theorem 35.3

If P \neq NP, then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a hamiltonian cycle \( H \), then \((G', c)\) contains a tour of cost \(|V|\).
- If \( G \) does not have a hamiltonian cycle, then any tour \( T \) must use some edge \( \notin E \).
Hardness of Approximation

Theorem 35.3
If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

- Let \( G = (V, E) \) be an instance of the Hamiltonian cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho|V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a Hamiltonian cycle \( H \), then \((G', c)\) contains a tour of cost \(|V|\).
- If \( G \) does not have a Hamiltonian cycle, then any tour \( T \) must use some edge \( \notin E \).

V. Travelling Salesman Problem

General TSP
Hardness of Approximation

**Theorem 35.3**

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:**

**Idea:** Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  
  $$c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise.}
  \end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\notin E$,
  
  $$\Rightarrow c(T) \geq (\rho |V| + 1) + (|V| - 1)$$

**Diagram:**

$G = (V, E)$ $\xrightarrow{\text{Reduction}}$ $G' = (V, E')$
Hardness of Approximation

Theorem 35.3

If P ≠ NP, then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho|V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a hamiltonian cycle \( H \), then \( (G', c) \) contains a tour of cost \( |V| \).
- If \( G \) does not have a hamiltonian cycle, then any tour \( T \) must use some edge \( \not\in E \),
  \[
  \Rightarrow c(T) \geq (\rho|V| + 1) + (|V| - 1) = (\rho + 1)|V|.
  \]

\[
\begin{array}{ll}
G = (V, E) & \rho \cdot 4 + 1 \\
G' = (V, E') & 1 \\
\end{array}
\]

Reduction

Large weight will render this edge useless!
Hardness of Approximation

Theorem 35.3

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

Proof:

Idea: Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \((u, v) \in E'\):
  \[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho|V| + 1 & \text{otherwise}.
  \end{cases}
  \]

- If \( G \) has a hamiltonian cycle \( H \), then \((G', c)\) contains a tour of cost \(|V|\).
- If \( G \) does not have a hamiltonian cycle, then any tour \( T \) must use some edge \( \notin E \),
  \[
  \Rightarrow c(T) \geq (\rho|V| + 1) + (|V| - 1) = (\rho + 1)|V|.
  \]
- Gap of \( \rho + 1 \) between tours which are using only edges in \( G \) and those which don’t

\[
\begin{align*}
G &= (V, E) \\
\text{Reduction} \\
\rho \cdot 4 + 1 \\
G' &= (V, E')
\end{align*}
\]
Hardness of Approximation

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:

Idea: Reduction from the hamiltonian-cycle problem.

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise}.
\end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\notin E$,

$$\Rightarrow \quad c(T) \geq (\rho |V| + 1) + (|V| - 1) = (\rho + 1)|V|.$$

- Gap of $\rho + 1$ between tours which are using only edges in $G$ and those which don’t.
- $\rho$-Approximation of TSP in $G'$ computes hamiltonian cycle in $G$ (if one exists)
Hardness of Approximation

Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

Proof:

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho |V| + 1 & \text{otherwise}. \end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$
- If $G$ does not have a hamiltonian cycle, then any tour $T$ must use some edge $\not\in E$,

$$\Rightarrow c(T) \geq (\rho |V| + 1) + (|V| - 1) = (\rho + 1)|V|.$$ 

- Gap of $\rho + 1$ between tours which are using only edges in $G$ and those which don’t
- $\rho$-Approximation of TSP in $G'$ computes hamiltonian cycle in $G$ (if one exists)
Proof of Theorem 35.3 from a higher perspective

instances of Hamilton

instances of TSP
Proof of Theorem 35.3 from a higher perspective

All instances with a hamiltonian cycle

instances of Hamilton

instances of TSP
Proof of Theorem 35.3 from a higher perspective

All instances with a hamiltonian cycle

All instances with cost $\leq k$

All instances with cost $> \rho \cdot k$

V. Travelling Salesman Problem

General TSP
Proof of Theorem 35.3 from a higher perspective

All instances with a hamiltonian cycle

All instances with cost $\leq k$

All instances with cost $> \rho \cdot k$

instances of Hamilton

instances of TSP
Proof of Theorem 35.3 from a higher perspective

All instances with a hamiltonian cycle

Instances of Hamilton

All instances with cost \( \leq k \)

Instances of TSP

All instances with cost \( > \rho \cdot k \)

V. Travelling Salesman Problem

General TSP
Proof of Theorem 35.3 from a higher perspective

All instances with a hamiltonian cycle

All instances with cost $\leq k$

All instances with cost $> \rho \cdot k$

instances of Hamilton

instances of TSP

V. Travelling Salesman Problem
Proof of Theorem 35.3 from a higher perspective

General Method to prove inapproximability results!

All instances with a hamiltonian cycle

All instances with cost $\leq k$

All instances with cost $> \rho \cdot k$

instances of Hamilton instances of TSP
Idea: First compute an MST, and then create a tour based on the tree.
Metric TSP (TSP Problem with the Triangle Inequality)

Idea: First compute an MST, and then create a tour based on the tree.

\textbf{APPROX-TSP-TOUR}(G, c)
1: select a vertex \( r \in G.V \) to be a “root” vertex
2: compute a minimum spanning tree \( T_{\text{min}} \) for \( G \) from root \( r \)
3: using \text{MST-PRIM}(G, c, r)
4: let \( H \) be a list of vertices, ordered according to when they are first visited
5: in a preorder walk of \( T_{\text{min}} \)
6: return the hamiltonian cycle \( H \)

Runtime is dominated by \text{MST-PRIM}, which is \( \Theta(V^2) \).

Remember: In the Metric-TSP problem, \( G \) is a complete graph.
Metric TSP (TSP Problem with the Triangle Inequality)

Idea: First compute an MST, and then create a tour based on the tree.

\[ \text{APPROX-TSP-TOUR} (G, c) \]
1: select a vertex \( r \in G.V \) to be a “root” vertex
2: compute a minimum spanning tree \( T_{\text{min}} \) for \( G \) from root \( r \)
3: using \( \text{MST-PRIM} (G, c, r) \)
4: let \( H \) be a list of vertices, ordered according to when they are first visited
5: in a preorder walk of \( T_{\text{min}} \)
6: return the hamiltonian cycle \( H \)

Runtime is dominated by \( \text{MST-PRIM} \), which is \( \Theta (V^2) \).
Metric TSP (TSP Problem with the Triangle Inequality)

Idea: First compute an MST, and then create a tour based on the tree.

\[ \text{APPROX-TSP-TOUR}(G, c) \]
1: select a vertex \( r \in G.V \) to be a “root” vertex
2: compute a minimum spanning tree \( T_{\text{min}} \) for \( G \) from root \( r \)
3: using \( \text{MST-PRIM}(G, c, r) \)
4: let \( H \) be a list of vertices, ordered according to when they are first visited
5: in a preorder walk of \( T_{\text{min}} \)
6: return the hamiltonian cycle \( H \)

Runtime is dominated by \( \text{MST-PRIM} \), which is \( \Theta(V^2) \).

Remember: In the Metric-TSP problem, \( G \) is a complete graph.
Run of APPROX-TSP-TOUR

Solution has cost \( \approx 19.704 \) - not optimal!
Better solution, yet still not optimal!

This is the optimal solution (cost \( \approx 14.715 \)).
Run of **APPROX-TSP-TOUR**

1. Compute MST $T_{\text{min}}$

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

Solution has cost \( \approx 19.704 \) - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost \( \approx 14.715 \)).

1. Compute MST \( T_{\text{min}} \) ✓

V. Travelling Salesman Problem

Metric TSP
Run of **APPROX-TSP-TOUR**

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
Run of APPROX-TSP-TOUR

V. Travelling Salesman Problem

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!

Better solution, yet still not optimal!

This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).

V. Travelling Salesman Problem Metric TSP
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of **APPROX-Tsp-TOUR**

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal! Better solution, yet still not optimal! This is the optimal solution ($\approx 14.715$).
Run of \textit{APPROX-Tsp-TOUR}

1. Compute MST $T_{\text{min}} \checkmark$
2. Perform preorder walk on MST $T_{\text{min}} \checkmark$
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!

Better solution, yet still not optimal!

This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$
2. Perform preorder walk on MST $T_{\text{min}}$
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

![Graph with vertices a, b, c, d, e, f, g, h and paths between them.]

Solution has cost $\approx 19.704$ - not optimal!

Better solution, yet still not optimal!

This is the optimal solution (cost $\approx 14.715$).
Run of **APPROX-TSP-TOUR**

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of **APPROX-TSP-TOUR**

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk ✓

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

Solution has cost $\approx 19.704$ - not optimal!

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk ✓
Run of **APPROX-TSP-TOUR**

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk ✓

Solution has cost $\approx 19.704$ - not optimal!

Better solution, yet still not optimal!

This is the optimal solution (cost $\approx 14.715$).
Run of \textbf{APPROX-TSP-TOUR}

- Better solution, yet still not optimal!

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk ✓
Run of APPROX-TSP-TOUR

1. Compute MST $T_{\text{min}}$ ✓
2. Perform preorder walk on MST $T_{\text{min}}$ ✓
3. Return list of vertices according to the preorder tree walk ✓
Run of **APPROX-TSP-TOUR**

This is the optimal solution (cost \( \approx 14.715 \)).

1. Compute MST \( T_{\text{min}} \) ✓
2. Perform preorder walk on MST \( T_{\text{min}} \) ✓
3. Return list of vertices according to the preorder tree walk ✓
Approximate Solution: Objective 921
Optimal Solution: Objective 699
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.
Proof of the Approximation Ratio

**Theorem 35.2**

\textsc{APPROX-TSP-TOUR} is a polynomial-time $2$-approximation for the traveling-salesman problem with the triangle inequality.

Proof:
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

Consider the optimal tour $H^*$ and remove an arbitrary edge $\Rightarrow$ yields a spanning tree $T$ and $c(T) \leq \text{min}$

Let $W$ be the full walk of the minimum spanning tree $T_{min}$ (including repeated visits) $\Rightarrow$ Full walk traverses every edge exactly twice, so $c(W) = 2c(T_{min}) \leq 2c(T) \leq 2c(H^*)$

Deleting duplicate vertices from $W$ yields a tour $H$ of APPROX-TSP exploiting that all edge costs are non-negative! exploiting triangle inequality!
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

Consider the optimal tour $H^*$ and remove an arbitrary edge $\implies$ yields a spanning tree $T$ and $\min c(T) \leq \text{Let } W \text{ be the full walk of the minimum spanning tree } T_{\text{min}} \text{(including repeated visits)} \implies \text{Full walk traverses every edge exactly twice, so } c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \text{ Deleting duplicate vertices from } W \text{ yields a tour } H \text{ of APPROX-TSP exploiting that all edge costs are non-negative! exploiting triangle inequality! solution } H \text{ of APPROX-TSP optimal solution } H^*
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  $\Rightarrow$ yields a spanning tree $T$ and

![Diagram of solution $H$ of APPROX-TSP and spanning tree $T$ as a subset of $H^*$]
**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  $\Rightarrow$ yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$

![Diagram of a solution $H$ of APPROX-TSP and a spanning tree $T$ as a subset of $H^*$]
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge.
- This yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$ exploiting that all edge costs are non-negative!

---

solution $H$ of APPROX-TSP

spanning tree $T$ as a subset of $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  - yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
Proof of the Approximation Ratio

**Theorem 35.2**

\textsc{Approx-Tsp-Tour} is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  \[ \implies \text{yields a spanning tree } T \text{ and } c(T_{min}) \leq c(T) \leq c(H^*) \]
- Let $W$ be the full walk of the minimum spanning tree $T_{min}$ (including repeated visits)
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  $\Rightarrow$ yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
**Theorem 35.2**

APP\-T\-SP\-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge

$\Rightarrow$ yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$

- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)

$\Rightarrow$ Full walk traverses every edge exactly twice, so

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  - yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  - Full walk traverses every edge exactly twice, so $c(W) = 2c(T_{\text{min}})$

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
**Proof of the Approximation Ratio**

Theorem 35.2

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:

- Consider the optimal tour $H^*$ and remove an arbitrary edge
- yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
- Full walk traverses every edge exactly twice, so
  \[ c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  - yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  - Full walk traverses every edge exactly twice, so
    \[ c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
**Proof of the Approximation Ratio**

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
- This yields a spanning tree $T$ and $c(T_{min}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{min}$ (including repeated visits)
- Full walk traverses every edge **exactly twice**, so
  \[ c(W) = 2c(T_{min}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

*APPROX-TSP-TOUR* is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  - yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  - Full walk traverses every edge exactly twice, so
    $$c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*)$$
- Deleting duplicate vertices from $W$ yields a tour $H$

Walk $W = (a, b, c, b, h, h, b, d, e, f, e, g, e, d, a)$

optimal solution $H^*$

\[ a \quad d \quad b \quad f \quad e \quad g \quad c \quad h \]
**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  \[ \Rightarrow \text{yields a spanning tree } T \text{ and } c(T_{\text{min}}) \leq c(T) \leq c(H^*) \]
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  \[ \Rightarrow \text{Full walk traverses every edge exactly twice, so } c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$

![Diagram](image)

Tour $H = (a, b, c, h, d, e, f, g, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  $\Rightarrow$ yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$

- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  $\Rightarrow$ Full walk traverses every edge exactly twice, so
  $$c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*)$$

- Deleting duplicate vertices from $W$ yields a tour $H$ with smaller cost:

Tour $H = (a, b, c, h, d, e, f, g, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:

- Consider the optimal tour $H^*$ and remove an arbitrary edge
  \( \Rightarrow \) yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  \( \Rightarrow \) Full walk traverses every edge exactly twice, so
  \[ c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$ with smaller cost:
  \[ c(H) \leq c(W) \]

Tour $H = (a, b, c, h, d, e, f, g, a)$

optimal solution $H^*$
Proof of the Approximation Ratio

Theorem 35.2

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  $\Rightarrow$ yields a spanning tree $T$ and $c(T_{\text{min}}) \leq c(T) \leq c(H^*)$
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  $\Rightarrow$ Full walk traverses every edge exactly twice, so
  \[ c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$ with smaller cost:
  \[ c(H) \leq c(W) \leq 2c(H^*) \]

![Diagram](https://via.placeholder.com/150)

Tour $H = (a, b, c, h, d, e, f, g, a)$

**exploiting triangle inequality!**

optimal solution $H^*$
**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour \( H^* \) and remove an arbitrary edge

\[ \Rightarrow \text{yields a spanning tree } T \text{ and } c(T_{\min}) \leq c(T) \leq c(H^*) \]

- Let \( W \) be the full walk of the minimum spanning tree \( T_{\min} \) (including repeated visits)

\[ \Rightarrow \text{Full walk traverses every edge exactly twice, so} \]

\[ c(W) = 2c(T_{\min}) \leq 2c(T) \leq 2c(H^*) \]

- Deleting duplicate vertices from \( W \) yields a tour \( H \) with smaller cost:

\[ c(H) \leq c(W) \leq 2c(H^*) \]

---

**Tour** \( H = (a, b, c, h, d, e, f, g, a) \)

**optimal solution** \( H^* \)
Proof of the Approximation Ratio

**Theorem 35.2**

Approx-TSP-Tour is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:
- Consider the optimal tour $H^*$ and remove an arbitrary edge
  \[ \Rightarrow \text{yields a spanning tree } T \text{ and } c(T_{\text{min}}) \leq c(T) \leq c(H^*) \]
- Let $W$ be the full walk of the minimum spanning tree $T_{\text{min}}$ (including repeated visits)
  \[ \Rightarrow \text{Full walk traverses every edge exactly twice, so} \]
  \[ c(W) = 2c(T_{\text{min}}) \leq 2c(T) \leq 2c(H^*) \]
- Deleting duplicate vertices from $W$ yields a tour $H$ with smaller cost:
  \[ c(H) \leq c(W) \leq 2c(H^*) \]

**Diagram:**
- Optimal solution $H^*$
- Minimum spanning tree as a subset of $H^*$
- Traveling salesman problem metric
- Proof exploiting triangle inequality!
### Theorem 35.2

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.
Christofides Algorithm

Theorem 35.2

\textbf{APPROX-TSP-TOUR} is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Can we get a better approximation ratio?
Christofides Algorithm

Theorem 35.2

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Can we get a better approximation ratio?

CHRISTOFIDES\((G, c)\)
1: select a vertex \(r \in G.V\) to be a "root" vertex
2: compute a minimum spanning tree \(T_{\text{min}}\) for \(G\) from root \(r\)
3: using MST-PRIM\((G, c, r)\)
4: compute a perfect matching \(M_{\text{min}}\) with minimum weight in the complete graph
5: over the odd-degree vertices in \(T_{\text{min}}\)
6: let \(H\) be a list of vertices, ordered according to when they are first visited
7: in a Eulearian circuit of \(T_{\text{min}} \cup M_{\text{min}}\)
8: return the hamiltonian cycle \(H\)
Christofides Algorithm

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Can we get a better approximation ratio?

**CHRISTOFIDES**$(G, c)$

1. select a vertex $r \in G.V$ to be a “root” vertex
2. compute a minimum spanning tree $T_{\text{min}}$ for $G$ from root $r$
3. using MST-PRIM$(G, c, r)$
4. compute a perfect matching $M_{\text{min}}$ with minimum weight in the complete graph over the odd-degree vertices in $T_{\text{min}}$
5. let $H$ be a list of vertices, ordered according to when they are first visited
6. in a Eulearian circuit of $T_{\text{min}} \cup M_{\text{min}}$
7. **return** the hamiltonian cycle $H$

**Theorem (Christofides’76)**

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.
Run of CHRISTOFIDES

Solution has cost $\approx 15.54$ - within 10% of the optimum!

1. Compute MST $T_{\text{min}}$
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$
4. Transform the Circuit into a Hamiltonian Cycle

All vertices in $T_{\text{min}} \cup M_{\text{min}}$ have even degree!
1. Compute MST $T_{\text{min}}$
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$

Solution has cost $\approx 15.54$ - within 10\% of the optimum!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of **CHRISTOFIDES**

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of **CHRISTOFIDES**

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$
Run of **CHRISTOFIDES**

1. Compute MST $T_{\text{min}}$

2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$
1. Compute MST $T_{min} \checkmark$

2. Add a minimum-weight perfect matching $M_{min}$ of the odd vertices in $T_{min} \checkmark$
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$

All vertices in $T_{\text{min}} \cup M_{\text{min}}$ have even degree!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓

All vertices in $T_{\text{min}} \cup M_{\text{min}}$ have even degree!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of Christofides

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!

V. Travelling Salesman Problem Metric TSP
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost ≈15.54 - within 10% of the optimum!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!

V. Travelling Salesman Problem Metric TSP
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of CHRISTOFIDES

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of **CHRISTOFIDES**

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
1. Compute MST $T_{min}$ ✓
2. Add a minimum-weight perfect matching $M_{min}$ of the odd vertices in $T_{min}$ ✓
3. Find an Eulerian Circuit in $T_{min} \cup M_{min}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle

Solution has cost $\approx 15.54$ - within 10% of the optimum!
Run of Christofides

Solution has cost $\approx 15.54$ - within 10% of the optimum!

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle ✓
Run of **CHRISTOFIDES**

Solution has cost ≈ 15.54 - within 10% of the optimum!

1. Compute MST $T_{\text{min}}$ ✓
2. Add a minimum-weight perfect matching $M_{\text{min}}$ of the odd vertices in $T_{\text{min}}$ ✓
3. Find an Eulerian Circuit in $T_{\text{min}} \cup M_{\text{min}}$ ✓
4. Transform the Circuit into a Hamiltonian Cycle ✓
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.
Proof of the Approximation Ratio

Theorem (Christofides’76)
There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):
Proof is quite similar to the previous analysis.
Proof of the Approximation Ratio

Theorem (Christofides’76)
There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio): Proof is quite similar to the previous analysis
- As before, let $H^*$ denote the optimal tour
There is a polynomial-time $3/2$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof is quite similar to the previous analysis

As before, let $H^*$ denote the optimal tour.

The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:

$$c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}}).$$

(1)
Proof of the Approximation Ratio

**Theorem (Christofides’76)**

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

**Proof (Approximation Ratio):**

- As before, let $H^*$ denote the optimal tour.
- The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:

$$c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}})$$  \hspace{1cm} (1)

Proof is quite similar to the previous analysis.

Number of odd-degree vertices is even!
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):

Proof is quite similar to the previous analysis

- As before, let $H^*$ denote the optimal tour
- The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:
  \[ c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}}) \]  
  (1)

- Let $H_{\text{odd}}^*$ be an optimal tour on the odd-degree vertices in $T_{\text{min}}$
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time $3/2$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):

- As before, let $H^*$ denote the optimal tour.
- The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:
  \[
  c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}})
  \]  
  \(1\)
- Let $H^*_{\text{odd}}$ be an optimal tour on the odd-degree vertices in $T_{\text{min}}$.
- Taking edges alternately, we obtain two matchings $M_1$ and $M_2$ such that
  \[
  c(M_1) + c(M_2) = c(H^*_{\text{odd}})
  \]
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):

- As before, let $H^*$ denote the optimal tour.
- The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:
  \[ c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}}) \]  
  \[ (1) \]
- Let $H_{\text{odd}}^*$ be an optimal tour on the odd-degree vertices in $T_{\text{min}}$.
- Taking edges alternately, we obtain two matchings $M_1$ and $M_2$ such that
  \[ c(M_1) + c(M_2) = c(H_{\text{odd}}^*) \]

Proof is quite similar to the previous analysis.

Number of odd-degree vertices is even!
Proof of the Approximation Ratio

There is a polynomial-time \( \frac{3}{2} \)-approximation algorithm for the travelling salesman problem with the triangle inequality.

**Theorem (Christofides’76)**

Proof is quite similar to the previous analysis.

**Proof (Approximation Ratio):**

- As before, let \( H^* \) denote the optimal tour.
- The **Eulerian Circuit** \( W \) uses each edge of the minimum spanning tree \( T_{\text{min}} \) and the minimum-weight matching \( M_{\text{min}} \) exactly once:
  
  \[
  c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}})
  \]
  
  \[(1)\]

- Let \( H_{\text{odd}}^* \) be an optimal tour on the odd-degree vertices in \( T_{\text{min}} \).
- Taking edges alternately, we obtain two matchings \( M_1 \) and \( M_2 \) such that
  
  \[
  c(M_1) + c(M_2) = c(H_{\text{odd}}^*)
  \]

- By shortcutting and the triangle inequality, \( \text{Number of odd-degree vertices is even!} \)

\[
(2)
\]
Proof of the Approximation Ratio

There is a polynomial-time \( \frac{3}{2} \)-approximation algorithm for the travelling salesman problem with the triangle inequality.

**Theorem (Christofides’76)**

Proof is quite similar to the previous analysis

**Proof (Approximation Ratio):**

- As before, let \( H^* \) denote the optimal tour
- The Eulerian Circuit \( W \) uses each edge of the minimum spanning tree \( T_{min} \) and the minimum-weight matching \( M_{min} \) exactly once:

\[
c(W) = c(T_{min}) + c(M_{min}) \leq c(H^*) + c(M_{min})
\]  

(1)

- Let \( H_{odd}^* \) be an optimal tour on the odd-degree vertices in \( T_{min} \)
- Taking edges alternately, we obtain two matchings \( M_1 \) and \( M_2 \) such that

\[
c(M_1) + c(M_2) = c(H_{odd}^*)
\]

- By shortcutting and the triangle inequality,

\[
c(M_{min}) \leq \frac{1}{2} c(H_{odd}^*) \leq \frac{1}{2} c(H^*).
\]  

(2)

Number of odd-degree vertices is even!
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):

- As before, let $H^*$ denote the optimal tour.
- The Eulerian Circuit $W$ uses each edge of the minimum spanning tree $T_{\text{min}}$ and the minimum-weight matching $M_{\text{min}}$ exactly once:

$$c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}}) \quad (1)$$

- Let $H_{\text{odd}}$ be an optimal tour on the odd-degree vertices in $T_{\text{min}}$.
- Taking edges alternately, we obtain two matchings $M_1$ and $M_2$ such that

$$c(M_1) + c(M_2) = c(H_{\text{odd}}^*)$$

- By shortcutting and the triangle inequality,

$$c(M_{\text{min}}) \leq \frac{1}{2} c(H_{\text{odd}}^*) \leq \frac{1}{2} c(H^*). \quad (2)$$

- Combining 1 with 2 yields

Proof is quite similar to the previous analysis.
Proof of the Approximation Ratio

Theorem (Christofides’76)

There is a polynomial-time \( \frac{3}{2} \)-approximation algorithm for the travelling salesman problem with the triangle inequality.

Proof (Approximation Ratio):

- As before, let \( H^* \) denote the optimal tour
- The Eulerian Circuit \( W \) uses each edge of the minimum spanning tree \( T_{\text{min}} \) and the minimum-weight matching \( M_{\text{min}} \) exactly once:
  \[
  c(W) = c(T_{\text{min}}) + c(M_{\text{min}}) \leq c(H^*) + c(M_{\text{min}})
  \]  
  (1)
- Let \( H_{\text{odd}}^* \) be an optimal tour on the odd-degree vertices in \( T_{\text{min}} \)
- Taking edges alternately, we obtain two matchings \( M_1 \) and \( M_2 \) such that
  \[
  c(M_1) + c(M_2) = c(H_{\text{odd}}^*)
  \]
- By shortcutting and the triangle inequality, \( \frac{1}{2} c(H_{\text{odd}}^*) \)
  \[
  c(M_{\text{min}}) \leq \frac{1}{2} c(H_{\text{odd}}^*) \leq \frac{1}{2} c(H^*).
  \]  
  (2)

- Combining 1 with 2 yields
  \[
  c(W) \leq c(H^*) + c(M_{\text{min}}) \leq c(H^*) + \frac{1}{2} c(H^*) = \frac{3}{2} c(H^*).
  \]
Concluding Remarks

Theorem (Christofides'76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

"Christos Papadimitriou told me that the traveling salesman problem is not a problem. It's an addiction."
Jon Bentley 1991
Concluding Remarks

Theorem (Christofides'76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

still the best algorithm for the metric TSP problem(!)
Concluding Remarks

Theorem (Christofides’76)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

still the best algorithm for the metric TSP problem(!)

Theorem (Arora’96, Mitchell’96)

There is a PTAS for the Euclidean TSP Problem.
Concluding Remarks

Theorem (Christofides’76)
There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

still the best algorithm for the metric TSP problem(!)

Theorem (Arora’96, Mitchell’96)
There is a PTAS for the Euclidean TSP Problem.

Both received the Gödel Award 2010

“Christos Papadimitriou told me that the traveling salesman problem is not a problem. It’s an addiction.”

Jon Bentley 1991
Concluding Remarks

Theorem (Christofides’76)
There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

still the best algorithm for the metric TSP problem(!)

Theorem (Arora’96, Mitchell’96)
There is a PTAS for the Euclidean TSP Problem.

Both received the Gödel Award 2010

“Christos Papadimitriou told me that the traveling salesman problem is not a problem. It’s an addiction.”

Jon Bentley 1991
Exercise: Prove that the approximation ratio of $\text{APPROX-}\text{TSP-TOUR}$ satisfies $\rho(n) < 2$.

Hint: Consider the effect of the shortcutting, but note that edge costs might be zero!
VI. Approx. Algorithms: Randomisation and Rounding

Thomas Sauerwald

Easter 2021
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
A randomised algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the expected cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$
A randomised algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the expected cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

Call such an algorithm randomised $\rho(n)$-approximation algorithm.
A randomised algorithm for a problem has approximation ratio \( \rho(n) \), if for any input of size \( n \), the expected cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

Call such an algorithm randomised \( \rho(n) \)-approximation algorithm.

An approximation scheme is an approximation algorithm, which given any input and \( \epsilon > 0 \), is a \((1 + \epsilon)\)-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed \( \epsilon > 0 \), the runtime is polynomial in \( n \). For example, \( O(n^2/\epsilon) \).
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both \( 1/\epsilon \) and \( n \). For example, \( O((1/\epsilon)^2 \cdot n^3) \).
Approximation Ratio

A randomised algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the expected cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

Call such an algorithm randomised $\rho(n)$-approximation algorithm.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^2/\epsilon)$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. For example, $O((1/\epsilon)^2 \cdot n^3)$. 

VI. Randomisation and Rounding
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
Given: 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

VI. Randomisation and Rounding MAX-3-CNF
MAX-3-CNF Satisfiability

- **Given**: 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal**: Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the satisfiability problem. Want to compute how “close” the formula to being satisfiable is.
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_5) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the **satisfiability** problem. Want to compute how “close” the formula to being satisfiable is.

Assume that no literal (including its negation) appears more than once in the same clause.
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: $(x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots$
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the **satisfiability** problem. Want to compute how “close” the formula to being satisfiable is.

Example:

$$(x_1 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_3} \lor \overline{x_5}) \land (x_2 \lor \overline{x_4} \lor x_5) \land (\overline{x_1} \lor x_2 \lor \overline{x_3})$$
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the *satisfiability* problem. Want to compute how “close” the formula to being satisfiable is.

Example:

\[
(x_1 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_3} \lor \overline{x_5}) \land (x_2 \lor \overline{x_4} \lor x_5) \land (\overline{x_1} \lor x_2 \lor \overline{x_3})
\]

\(x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0\) and \(x_5 = 1\) satisfies 3 (out of 4 clauses)
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the satisfiability problem. Want to compute how “close” the formula to being satisfiable is.

Example:

\[(x_1 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_3} \lor \overline{x_5}) \land (x_2 \lor \overline{x_4} \lor x_5) \land (\overline{x_1} \lor x_2 \lor \overline{x_3})\]

\[x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0 \text{ and } x_5 = 1 \text{ satisfies 3 (out of 4 clauses)}\]

**Idea:** What about assigning each variable uniformly and independently at random?
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

Proof:
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \(n\) variables \(x_1, x_2, \ldots, x_n\) and \(m\) clauses, the randomised algorithm that sets each variable independently at random is a randomised \(8/7\)-approximation algorithm.

Proof:
- For every clause \(i = 1, 2, \ldots, m\), define a random variable:
  \[Y_i = 1\{\text{clause } i \text{ is satisfied}\}\]
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1\{\text{clause } i \text{ is satisfied}\}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr [ \text{clause } i \text{ is not satisfied} ] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[\Rightarrow\]
  \[
  \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
Analysis

Theorem 35.6
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[
  \Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
  \[
  \Rightarrow \quad E[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]
Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

**Proof:**
- For every clause $i = 1, 2, \ldots, m$, define a random variable:
  $$Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}$$
- Since each literal (including its negation) appears at most once in clause $i$,
  $$\Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$$
  $$\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}$$
  $$\Rightarrow \quad E[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}$$
- Let $Y := \sum_{i=1}^{m} Y_i$ be the number of satisfied clauses. Then,
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

Proof:

- For every clause $i = 1, 2, \ldots, m$, define a random variable:
  
  $$Y_i = \mathbf{1}\{\text{clause } i \text{ is satisfied}\}$$

- Since each literal (including its negation) appears at most once in clause $i$,
  
  $$\Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$$

  $$\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}$$

  $$\Rightarrow \quad \mathbb{E}[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.$$  

- Let $Y := \sum_{i=1}^{m} Y_i$ be the number of satisfied clauses. Then,
  
  $$\mathbb{E}[Y]$$
**Theorem 35.6**

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

**Proof:**

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[
  \Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
  \[
  \Rightarrow \quad E[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]
- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[
  E[Y] = E\left[ \sum_{i=1}^{m} Y_i \right]
  \]

VI. Randomisation and Rounding
Theorem 35.6

Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

Proof:
- For every clause $i = 1, 2, \ldots, m$, define a random variable:
  \[ Y_i = 1 \{ \text{clause } i \text{ is satisfied} \} \]
- Since each literal (including its negation) appears at most once in clause $i$,
  \[ \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8} \]
  \[ \Rightarrow \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8} \]
  \[ \Rightarrow \mathbb{E}[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}. \]
- Let $Y := \sum_{i=1}^{m} Y_i$ be the number of satisfied clauses. Then,
  \[ \mathbb{E}[Y] = \mathbb{E}\left[ \sum_{i=1}^{m} Y_i \right] \]

Linearity of Expectations
Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[
  \Rightarrow \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
  \[
  \Rightarrow \mathbb{E}[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]
- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[
  \mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathbb{E}[Y_i]
  \]  
  (Linearity of Expectations)
Analysis

**Theorem 35.6**

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

**Proof:**

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[ Y_i = \mathbf{1}\{\text{clause } i \text{ is satisfied}\} \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[ \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8} \]
  \[ \Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8} \]
  \[ \Rightarrow \quad \mathbb{E}[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}. \]
- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[ \mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathbb{E}[Y_i] = \sum_{i=1}^{m} \frac{7}{8} \]
  \[
  \text{Linearity of Expectations}
  \]
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( \frac{8}{7} \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[
  \Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
  \[
  \Rightarrow \quad \mathbb{E}[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]
- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[
  \mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathbb{E}[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7}{8} \cdot m.
  \]

Linearity of Expectations

VI. Randomisation and Rounding

MAX-3-CNF
Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( 8/7 \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[ Y_i = 1 \{ \text{clause } i \text{ is satisfied} \} \]

- Since each literal (including its negation) appears at most once in clause \( i \),
  \[ \Pr [\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8} \]

  \[ \Rightarrow \quad \Pr [\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8} \]

  \[ \Rightarrow \quad \mathbb{E} [Y_i] = \Pr [Y_i = 1] \cdot 1 = \frac{7}{8}. \]

- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[ \mathbb{E} [Y] = \mathbb{E} \left[ \sum_{i=1}^{m} Y_i \right] = \sum_{i=1}^{m} \mathbb{E} [Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7}{8} \cdot m. \]

Linearity of Expectations

maximum number of satisfiable clauses is \( m \)
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised 8/7-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]
- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr[\text{clause } i \text{ is not satisfied}] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]
  \[
  \Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \frac{1}{8} = \frac{7}{8}
  \]
  \[
  \Rightarrow \quad E[Y_i] = \Pr[Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]
- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[
  E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7}{8} \cdot m. \quad \square
  \]

Linearity of Expectations

maximum number of satisfiable clauses is \( m \)
Interesting Implications

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised 8/7-approximation algorithm.
Interesting Implications

Theorem 35.6
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

Corollary
For any instance of MAX-3-CNF, there exists an assignment which satisfies at least \( \frac{7}{8} \) of all clauses.
Interesting Implications

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

Corollary

For any instance of MAX-3-CNF, there exists an assignment which satisfies at least \( \frac{7}{8} \) of all clauses.

There is \( \omega \in \Omega \) such that \( Y(\omega) \geq \mathbb{E}[Y] \)
Interesting Implications

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

Corollary

For any instance of MAX-3-CNF, there exists an assignment which satisfies at least \( \frac{7}{8} \) of all clauses.

There is \( \omega \in \Omega \) such that \( Y(\omega) \geq \mathbb{E}[Y] \)

Probabilistic Method: powerful tool to show existence of a non-obvious property.
Interesting Implications

**Theorem 35.6**

Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised $8/7$-approximation algorithm.

**Corollary**

For any instance of MAX-3-CNF, there exists an assignment which satisfies at least $\frac{7}{8}$ of all clauses.

There is $\omega \in \Omega$ such that $Y(\omega) \geq E[ Y ]$

**Probabilistic Method:** powerful tool to show existence of a non-obvious property.

**Corollary**

Any instance of MAX-3-CNF with at most 7 clauses is satisfiable.
Interesting Implications

Theorem 35.6
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

Corollary
For any instance of MAX-3-CNF, there exists an assignment which satisfies at least \( \frac{7}{8} \) of all clauses.

There is \( \omega \in \Omega \) such that \( Y(\omega) \geq \mathbb{E}[Y] \)

Probabilistic Method: powerful tool to show existence of a non-obvious property.

Corollary
Any instance of MAX-3-CNF with at most 7 clauses is satisfiable.

Follows from the previous Corollary.
Expected Approximation Ratio

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.
Expected Approximation Ratio

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

One could prove that the probability to satisfy \( \left(\frac{7}{8}\right) \cdot m \) clauses is at least \( \frac{1}{8m} \).
Expected Approximation Ratio

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( 8/7 \)-approximation algorithm.

One could prove that the probability to satisfy \((7/8) \cdot m\) clauses is at least \(1/(8m)\).

\[
E[Y] = \frac{1}{2} \cdot E[Y | x_1 = 1] + \frac{1}{2} \cdot E[Y | x_1 = 0].
\]

\( Y \) is defined as in the previous proof.
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

One could prove that the probability to satisfy \((\frac{7}{8}) \cdot m\) clauses is at least \( \frac{1}{8m} \).

\[
E[Y] = \frac{1}{2} \cdot E[Y \mid x_1 = 1] + \frac{1}{2} \cdot E[Y \mid x_1 = 0].
\]

\( Y \) is defined as in the previous proof.

One of the two conditional expectations is at least \( E[Y] \)!
Expected Approximation Ratio

**Theorem 35.6**

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( 8/7 \)-approximation algorithm.

One could prove that the probability to satisfy \( (7/8) \cdot m \) clauses is at least \( 1/(8m) \).

\[
E[Y] = \frac{1}{2} \cdot E[Y | x_1 = 1] + \frac{1}{2} \cdot E[Y | x_1 = 0].
\]

\( Y \) is defined as in the previous proof.

One of the two conditional expectations is at least \( E[Y] \!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!
**Expected Approximation Ratio**

**Theorem 35.6**

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( 8/7 \)-approximation algorithm.

One could prove that the probability to satisfy \( \left( \frac{7}{8} \right) \cdot m \) clauses is at least \( \frac{1}{8m} \).

\[
\mathbb{E}[Y] = \frac{1}{2} \cdot \mathbb{E}[Y \mid x_1 = 1] + \frac{1}{2} \cdot \mathbb{E}[Y \mid x_1 = 0].
\]

\( Y \) is defined as in the previous proof.

One of the two conditional expectations is at least \( \mathbb{E}[Y] \! \).

**GREEDY-3-CNF(\( \phi, n, m \))**

1: **for** \( j = 1, 2, \ldots, n \)
2: Compute \( \mathbb{E}[Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1] \)
3: Compute \( \mathbb{E}[Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 0] \)
4: Let \( x_j = v_j \) so that the conditional expectation is maximized
5: **return** the assignment \( v_1, v_2, \ldots, v_n \)
Analysis of GREEDY-3-CNF($\phi, n, m$)

**Theorem**

GREEDY-3-CNF($\phi, n, m$) is a polynomial-time $8/7$-approximation.
Analysis of \textsc{Greedy-3-CNF}(\(\phi, n, m\))

This algorithm is deterministic.

\textsc{Greedy-3-CNF}(\(\phi, n, m\)) is a polynomial-time 8/7-approximation.
Analysis of \textsc{Greedy-3-CNF}(\phi, n, m)

Theorem
\textsc{Greedy-3-CNF}(\phi, n, m) is a polynomial-time 8/7-approximation.

Proof:

This algorithm is deterministic.
Analysis of **GREEDY-3-CNF**($\phi, n, m$)

**Theorem**

GREEDY-3-CNF($\phi, n, m$) is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm

This algorithm is deterministic.
Analysis of $\text{GREEDY-3-CNF}(\phi, n, m)$

This algorithm is deterministic.

$\text{GREEDY-3-CNF}(\phi, n, m)$ is a polynomial-time $8/7$-approximation.

Theorem

Proof:

- **Step 1:** polynomial-time algorithm
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments

VI. Randomisation and Rounding MAX-3-CNF

$\text{GREEDY-3-CNF}(\phi, n, m)$ is a polynomial-time $8/7$-approximation.

This algorithm is deterministic.

Proof:

- **Step 1:** polynomial-time algorithm
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
Analysis of **GREEDY-3-CNF**($\phi, n, m$)

This algorithm is deterministic.

**Theorem**

**GREEDY-3-CNF**($\phi, n, m$) is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:

\[
E \left[ Y \mid x_1 = v_1, \ldots, x_{j-2} = v_{j-2}, x_j = 1 \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] \geq \ldots \geq E \left[ Y \right] = \frac{7}{8} \cdot m.
\]
Analysis of \textbf{GREEDY-3-CNF}(\(\phi, n, m\))

\textbf{Theorem}

\(\text{GREEDY-3-CNF}(\phi, n, m)\) is a polynomial-time 8/7-approximation.

\textbf{Proof:}

- \textbf{Step 1:} polynomial-time algorithm
  - In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments
  - A smarter way is to use linearity of (conditional) expectations:

  \[
  E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
  \]

This algorithm is deterministic.
Analysis of \( \text{GREEDY-3-CNF}(\phi, n, m) \)

**Theorem**

\( \text{GREEDY-3-CNF}(\phi, n, m) \) is a polynomial-time 8/7-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm
  - In iteration \( j = 1, 2, \ldots, n \), \( Y = Y(\phi) \) averages over \( 2^{n-j+1} \) assignments
  - A smarter way is to use linearity of (conditional) expectations:

\[
E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
\]

**This algorithm is deterministic.**
Analysis of GREEDY-3-CNF($\phi, n, m$)

**Theorem**

GREEDY-3-CNF($\phi, n, m$) is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm ✓
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:

\[
E \left[ Y \middle| x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \middle| x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
\]

computable in $O(1)$

This algorithm is deterministic.
Analysis of **GREEDY-3-CNF**($\phi, n, m$)

**Theorem**

GREEDY-3-CNF($\phi, n, m$) is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm ✓
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:

\[
E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
\]

- **Step 2:** satisfies at least $7/8 \cdot m$ clauses

This algorithm is deterministic.
Analysis of **GREEDY-3-CNF**($\phi, n, m$)

**Theorem**

$\text{GREEDY-3-CNF}(\phi, n, m)$ is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm ✓
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:

  $$E \left[ Y \mid x_1 = v_1, \ldots, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]$$

- **Step 2:** satisfies at least $7/8 \cdot m$ clauses
  - Due to the greedy choice in each iteration $j = 1, 2, \ldots, n$,
Analysis of **GREEDY-3-CNF**\((\phi, n, m)\)

**Theorem**

**GREEDY-3-CNF**\((\phi, n, m)\) is a polynomial-time \(8/7\)-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm √
  - In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments
  - A smarter way is to use linearity of (conditional) expectations:
    \[
    E\left[ Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E\left[ Y_i | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
    \]

- **Step 2:** satisfies at least \(7/8 \cdot m\) clauses
  - Due to the greedy choice in each iteration \(j = 1, 2, \ldots, n\),
    \[
    E\left[ Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j \right] \geq E\left[ Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1} \right]
    \]
Analysis of GREEDY-3-CNF($\phi, n, m$)

**Theorem**

GREEDY-3-CNF($\phi, n, m$) is a polynomial-time $8/7$-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm ✓
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:
    \[
    E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
    \]

- **Step 2:** satisfies at least $7/8 \cdot m$ clauses
  - Due to the greedy choice in each iteration $j = 1, 2, \ldots, n,$
    \[
    E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1} \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-2} = v_{j-2} \right]
    \]
Analysis of \texttt{GREEDY-3-CNF}(\(\phi, n, m\))

\textbf{Theorem}

\texttt{GREEDY-3-CNF}(\(\phi, n, m\)) is a polynomial-time \(8/7\)-approximation.

\textbf{Proof:}

\begin{itemize}
  \item \textbf{Step 1:} polynomial-time algorithm \(\checkmark\)
    \begin{itemize}
      \item In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments
      \item A smarter way is to use linearity of (conditional) expectations:
    \end{itemize}
    \begin{equation}
      \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1] = \sum_{i=1}^{m} \mathbb{E}[Y_i | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1]
    \end{equation}
  \item \textbf{Step 2:} satisfies at least \(7/8 \cdot m\) clauses
    \begin{itemize}
      \item Due to the greedy choice in each iteration \(j = 1, 2, \ldots, n\),
      \begin{align*}
        \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j] & \geq \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}] \\
        & \geq \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-2} = v_{j-2}] \\
        & \vdots \\
        & \geq \mathbb{E}[Y]
      \end{align*}
    \end{itemize}
\end{itemize}

This algorithm is deterministic.
Analysis of **GREEDY-3-CNF**($\phi, n, m$)

**Theorem**

**GREEDY-3-CNF**($\phi, n, m$) is a polynomial-time 8/7-approximation.

**Proof:**

- **Step 1:** polynomial-time algorithm ✓
  - In iteration $j = 1, 2, \ldots, n$, $Y = Y(\phi)$ averages over $2^{n-j+1}$ assignments
  - A smarter way is to use linearity of (conditional) expectations:
    
    \[
    E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
    \]

- **Step 2:** satisfies at least $7/8 \cdot m$ clauses
  - Due to the greedy choice in each iteration $j = 1, 2, \ldots, n$,
    
    \[
    E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1} \right]
    \]
    
    \[
    \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-2} = v_{j-2} \right] 
    \]
    
    \[
    \vdots 
    \]
    
    \[
    \geq E \left[ Y \right] = \frac{7}{8} \cdot m.
    \]
Analysis of \textit{GREEDY-3-CNF}(\(\phi, n, m\))

\textbf{Theorem}

\textit{GREEDY-3-CNF}(\(\phi, n, m\)) is a polynomial-time 8/7-approximation.

\textbf{Proof:}

\textbf{Step 1:} polynomial-time algorithm \(\checkmark\)
- In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments
- A smarter way is to use linearity of (conditional) expectations:

\[
E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right] = \sum_{i=1}^{m} E \left[ Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1 \right]
\]

\textbf{Step 2:} satisfies at least \(7/8 \cdot m\) clauses \(\checkmark\)
- Due to the greedy choice in each iteration \(j = 1, 2, \ldots, n\),

\[
E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1} \right] \geq E \left[ Y \mid x_1 = v_1, \ldots, x_{j-2} = v_{j-2} \right] \cdots \geq E \left[ Y \right] = \frac{7}{8} \cdot m.
\]
Analysis of \textsc{Greedy-3-CNF}(\phi, n, m)

This algorithm is deterministic.

\textsc{Greedy-3-CNF}(\phi, n, m) is a polynomial-time 8/7-approximation.

Proof:

- **Step 1**: polynomial-time algorithm \(\checkmark\)
  - In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments.
  - A smarter way is to use linearity of (conditional) expectations:

\[
E[Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1] = \sum_{i=1}^{m} E[Y_i \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1]
\]

- **Step 2**: satisfies at least \(7/8 \cdot m\) clauses \(\checkmark\)
  - Due to the greedy choice in each iteration \(j = 1, 2, \ldots, n\),

\[
E[Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v_j] \geq E[Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}]
\]

\[
\geq E[Y \mid x_1 = v_1, \ldots, x_{j-2} = v_{j-2}]
\]

\[
\vdots
\]

\[
\geq E[Y] = \frac{7}{8} \cdot m. \quad \square
\]
Run of \textsc{Greedy-3-CNF}(\varphi, n, m)

\[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor \overline{x}_4) \land (x_1 \lor x_2 \lor \overline{x}_4) \land (x_1 \lor x_2 \lor x_4) \land (x_1 \lor \overline{x}_2 \lor \overline{x}_5) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x}_3 \lor \overline{x}_4)\]
Run of **GREEDY-3-CNF**($\varphi, n, m$)

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor \overline{x}_4) \land (x_1 \lor x_2 \lor x_4) \land (x_1 \lor \overline{x}_2 \lor x_3) \land (x_1 \lor x_2 \lor \overline{x}_4) \land (x_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x}_3 \lor \overline{x}_4)$$
Run of GREEDY-3-CNF($\varphi, n, m$)

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_1 \lor x_2 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_1 \lor x_2 \lor x_4) \land (\overline{x_1} \lor \overline{x_3} \lor \overline{x_4}) \land (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x_3} \lor \overline{x_4})$$

![Diagram of the run of GREEDY-3-CNF($\varphi, n, m$)]
Run of $\text{GREEDY-3-CNF}(\varphi, n, m)$

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x_3} \lor \overline{x_4})$$
Run of GREEDY-3-CNF$(\varphi, n, m)$

\[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_4) \land (x_1 \lor x_2 \lor x_4) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_3 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)\]

VI. Randomisation and Rounding
Run of **GREEDY-3-CNF**($\varphi, n, m$)

$$1 \land 1 \land 1 \land (x_3 \lor x_4) \land 1 \land (\overline{x_2} \lor \overline{x_3}) \land (x_2 \lor x_3) \land (\overline{x_2} \lor x_3) \land 1 \land (x_2 \lor \overline{x_3} \lor \overline{x_4})$$

VI. Randomisation and Rounding MAX-3-CNF
Run of GREEDY-3-CNF ($\varphi, n, m$)

$$1 \land 1 \land 1 \land (\overline{x}_3 \lor x_4) \land 1 \land (\overline{x}_2 \lor x_3) \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_3) \land 1 \land (x_2 \lor \overline{x}_3 \lor \overline{x}_4)$$
Run of **GREEDY-3-CNF** $(\varphi, n, m)$

\[
1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land (\overline{x_2} \lor \overline{x_3}) \land (x_2 \lor x_3) \land (\overline{x_2} \lor x_3) \land 1 \land (x_2 \lor \overline{x_3} \lor \overline{x_4})
\]
Run of GREEDY-3-CNF($\varphi, n, m$)

$$1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land (\overline{x_2} \lor x_3) \land (x_2 \lor x_3) \land (\overline{x_2} \lor x_3) \land 1 \land (x_2 \lor x_3 \lor \overline{x_4})$$
Run of **GREEDY-3-CNF**($\varphi, n, m$)

$$1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land 1 \land (x_3) \land 1 \land 1 \land (\overline{x_3} \lor \overline{x_4})$$
Run of GREEDY-3-CNF($\varphi, n, m$)

\[ 1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land 1 \land (x_3) \land 1 \land 1 \land (\overline{x_3} \lor \overline{x_4}) \]

VI. Randomisation and Rounding MAX-3-CNF
Run of \textsc{Greedy-3-CNF}(\(\varphi, n, m\))

\[
1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land 1 \land (x_3) \land 1 \land 1 \land (\overline{x_3} \lor \overline{x_4})
\]
Run of **GREEDY-3-CNF** \((\varphi, n, m)\)

\[ 1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land 1 \land (x_3) \land 1 \land 1 \land (x_3 \lor \overline{x_4}) \]

\[ x_1 = 0 \]

\[ x_1 = 1 \]

\[ x_2 = 0 \]

\[ x_2 = 1 \]

\[ x_3 = 0 \]

\[ x_3 = 1 \]

\[ x_4 = 0 \]

\[ x_4 = 1 \]

\[ x_5 = 0 \]

\[ x_5 = 1 \]

\[ x_6 = 0 \]

\[ x_6 = 1 \]

\[ x_7 = 0 \]

\[ x_7 = 1 \]

\[ x_8 = 0 \]

\[ x_8 = 1 \]

\[ x_9 = 0 \]

\[ x_9 = 1 \]

\[ x_{10} = 0 \]

\[ x_{10} = 1 \]

\[ x_{11} = 0 \]

\[ x_{11} = 1 \]
Run of GREEDY-3-CNF(φ, n, m)

1 ∧ 1 ∧ 1 ∧ 1 ∧ 1 ∧ 1 ∧ 0 ∧ 1 ∧ 1 ∧ 1

x₁ = 0

x₂ = 0

x₃ = 0

x₁ = 1

x₂ = 1

x₃ = 1

x₂ = 0

x₃ = 0

x₁ = 0

x₂ = 1

x₃ = 1

x₁ = 1

x₂ = 1

x₃ = 1

x₂ = 0

x₃ = 0

x₂ = 1

x₃ = 1

Returned solution satisfies 9 out of 10 clauses, but the formula is satisfiable.
Run of **GREEDY-3-CNF**($\phi, n, m$)

$$1 \land 1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1$$

$\phi$, $n$, $m$
Run of GREEDY-3-CNF($\varphi, n, m$)

$1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1$

V. Randomisation and Rounding MAX-3-CNF

 reused
Run of GREEDY-3-CNF($\varphi, n, m$)

1 ∧ 1 ∧ 1 ∧ 1 ∧ 1 ∧ 1 ∧ 0 ∧ 1 ∧ 1 ∧ 1

VI. Randomisation and Rounding MAX-3-CNF
Run of **GREEDY-3-CNF**\((\varphi, n, m)\)

\[ 1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1 \]

Randomisation and Rounding MAX-3-CNF

\[ x_1 = 0 \quad 8.75 \]

\[ x_1 = 1 \]

\[ x_2 = 0 \]

\[ x_2 = 1 \]

\[ x_3 = 0 \]

\[ x_3 = 1 \]
Run of **GREEDY-3-CNF** $(\varphi, n, m)$

$$1 \land 1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1$$

### Randomisation and Rounding MAX-3-CNF

The solution process involves making decisions at each step based on the values of $x_1, x_2, x_3$. The tree structure visualizes the decision-making process, with each node representing a possible assignment of variables, and the branches indicating the next steps based on the satisfaction of clauses.

- **Base Case (leaf nodes)**: Each leaf node represents a complete assignment of variables, and the number indicates the satisfaction level of the corresponding clauses.
- **Decision Points**: Each internal node represents a decision point where the value of a variable is chosen. The decision is guided by the goal of maximizing the number of satisfied clauses.

The run of the algorithm results in satisfying 9 out of 10 clauses, indicating a high level of satisfaction with the given constraints.
Run of GREEDY-3-CNF($\varphi, n, m$)

\[ 1 \land 1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1 \]

Returned solution satisfies 9 out of 10 clauses, but the formula is satisfiable.
Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( 8/7 \)-approximation algorithm.
Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

**Theorem** 

**Theorem**

$\text{GREEDY-3-CNF}(\phi, n, m)$ is a polynomial-time $8/7$-approximation.

**Theorem 35.6**

Essentially there is nothing smarter than just guessing!
Theorem 35.6

Given an instance of MAX-3-CNF with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomised algorithm that sets each variable independently at random is a randomised $8/7$-approximation algorithm.

Theorem

$\text{GREEDY-3-CNF}(\phi, n, m)$ is a polynomial-time $8/7$-approximation.

Theorem (Hastad’97)

For any $\epsilon > 0$, there is no polynomial time $8/7 - \epsilon$ approximation algorithm of MAX3-CNF unless P=NP.
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( 8/7 \)-approximation algorithm.

**Theorem 3.5.6**
\[
\text{GREEDY-3-CNF}(\phi, n, m) \text{ is a polynomial-time } 8/7\text{-approximation.}
\]

**Theorem**
For any \( \epsilon > 0 \), there is no polynomial time \( 8/7 - \epsilon \) approximation algorithm of MAX3-CNF unless P=NP.

**Theorem (Hastad’97)**
Essentially there is nothing smarter than just guessing!
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
The **Weighted Vertex-Cover Problem**

- **Given**: Undirected, vertex-weighted graph \( G = (V, E) \)
- **Goal**: Find a minimum-weight subset \( V' \subseteq V \) such that if \((u, v) \in E(G)\), then \( u \in V' \) or \( v \in V' \).

**Vertex Cover Problem**

**Applications:**
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Weight of a vertex could be salary of a person
- Perform all tasks with the minimal amount of resources
The Weighted Vertex-Cover Problem

Vertex Cover Problem

- **Given:** Undirected, vertex-weighted graph $G = (V, E)$
- **Goal:** Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$. 

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Weight of a vertex could be salary of a person
- Perform all tasks with the minimal amount of resources
The **Weighted Vertex-Cover Problem**

**Vertex Cover Problem**

- **Given:** Undirected, vertex-weighted graph $G = (V, E)$
- **Goal:** Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

**Applications:**
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Weight of a vertex could be salary of a person
- Perform all tasks with the minimal amount of resources
The Weighted Vertex-Cover Problem

- **Given:** Undirected, vertex-weighted graph $G = (V, E)$
- **Goal:** Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is (still) an NP-hard problem.
The **Weighted Vertex-Cover Problem**

Given: Undirected, vertex-weighted graph $G = (V, E)$

Goal: Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is (still) an NP-hard problem.

Applications:

Every edge forms a task, and every vertex represents a person/machine which can execute that task. Perform all tasks with the minimal amount of resources.
The **Weighted Vertex-Cover Problem**

**Given:** Undirected, vertex-weighted graph $G = (V, E)$

**Goal:** Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is (still) an NP-hard problem.

**Applications:**
- Every edge forms a task, and every vertex represents a person/machine which can execute that task.

![Graph Example]
The **Weighted Vertex-Cover Problem**

Given: Undirected, vertex-weighted graph $G = (V, E)$

Goal: Find a minimum-weight subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is (still) an NP-hard problem.

**Applications:**

- Every edge forms a **task**, and every vertex represents a person/machine which can execute that task
- **Weight** of a vertex could be salary of a person
The **Weighted Vertex-Cover Problem**

**Given:** Undirected, vertex-weighted graph \( G = (V, E) \)

**Goal:** Find a minimum-weight subset \( V' \subseteq V \) such that if \( (u, v) \in E(G) \), then \( u \in V' \) or \( v \in V' \).

This is (still) an NP-hard problem.

**Applications:**
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Weight of a vertex could be salary of a person
- Perform all tasks with the minimal amount of resources
The Greedy Approach from (Unweighted) Vertex Cover

**APPROX-VERTEX-COVER** \((G)\)

1.  \(C = \emptyset\)
2.  \(E' = G.E\)
3.  \textbf{while} \(E' \neq \emptyset\)
4.      let \((u, v)\) be an arbitrary edge of \(E'\)
5.      \(C = C \cup \{u, v\}\)
6.      remove from \(E'\) every edge incident on either \(u\) or \(v\)
7.  \textbf{return} \(C\)
The Greedy Approach from (Unweighted) Vertex Cover

**Algorithm: APPROX-VERTEX-COVER**

1. $C = \emptyset$
2. $E' = G.E$
3. while $E' \neq \emptyset$
4. let $(u, v)$ be an arbitrary edge of $E'$
5. $C = C \cup \{u, v\}$
6. remove from $E'$ every edge incident on either $u$ or $v$
7. return $C$

**Diagram:**

- The graph consists of vertices labeled a, b, c, d, e, with weights 1, 1, 1, 1, 1 respectively.
- Vertex a is connected to b, c, d, and e.
- The total weight of the computed solution is 100.
- The optimal solution has a weight of 4.
The Greedy Approach from (Unweighted) Vertex Cover

\textbf{APPROX-VERTEX-COVER}(G)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \textbf{while} \( E' \neq \emptyset \)
4. \textbf{let} \((u, v)\) \textbf{be an arbitrary edge of} \( E' \)
5. \( C = C \cup \{u, v\} \)
6. \textbf{return} \( C \)

\begin{figure}
\centering
\includegraphics[width=0.7\textwidth]{figure}
\caption{The operation of \textsc{Approx-Vertex-Cover}.
(a) The input graph \( G \), which has 7 vertices and 8 edges.
(b) The edge \((b, c)\), shown heavy, is the first edge chosen by \textsc{Approx-Vertex-Cover}.
Vertices \( b \) and \( c \), shown lightly shaded, are added to the set \( C \) containing the vertex cover being created.
Edges \((a, b)\), \((c, e)\), and \((c, d)\), shown dashed, are removed since the vertex \( y \) are now covered by some vertex in \( C \).
(c) Edge \((e, f)\) is chosen; vertices \( e \) and \( f \) are added to \( C \).
(d) Edge \((d, g)\) is chosen; vertices \( d \) and \( g \) are added to \( C \).
(e) The set \( C \), which is the vertex cover produced by \textsc{Approx-Vertex-Cover}, contains the vertices \( b; c; d; e; f; g \).
(f) The optimal vertex cover for this problem contains only three vertices: \( b \), \( d \), and \( e \).}
\end{figure}
The Greedy Approach from (Unweighted) Vertex Cover

**APPROX-VERTEX-COVER** *(G)*

1. \( C = \emptyset \)
2. \( E' = G . E \)
3. while \( E' \neq \emptyset \)
4. \( \text{let} \ (u, v) \text{ be an arbitrary edge of } E' \)
5. \( C = C \cup \{u, v\} \)
6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)

Optimal solution has weight 4

Computed solution has weight 101

VI. Randomisation and Rounding

Weighted Vertex Cover
Invoking an (Integer) Linear Program

Idea: Round the solution of an associated linear program.
Invoking an (Integer) Linear Program

Idea: Round the solution of an associated linear program.

0-1 Integer Program

minimize \( \sum_{v \in V} w(v)x(v) \)
subject to \( x(u) + x(v) \geq 1 \) for each \( (u, v) \in E \)
\( x(v) \in \{0, 1\} \) for each \( v \in V \)
Invoking an (Integer) Linear Program

**Idea:** Round the solution of an associated linear program.

0-1 Integer Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v)x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in \{0, 1\} \quad \text{for each } v \in V
\end{align*}
\]

Linear Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v)x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in [0, 1] \quad \text{for each } v \in V
\end{align*}
\]
Invoking an (Integer) Linear Program

Idea: Round the solution of an associated linear program.

0-1 Integer Program

minimize \( \sum_{v \in V} w(v)x(v) \)

subject to \( x(u) + x(v) \geq 1 \) for each \((u, v) \in E\)

\( x(v) \in \{0, 1\} \) for each \( v \in V \)

Linear Program

minimize \( \sum_{v \in V} w(v)x(v) \)

subject to \( x(u) + x(v) \geq 1 \) for each \((u, v) \in E\)

\( x(v) \in [0, 1] \) for each \( v \in V \)

Optimum is a lower bound on the optimal weight of a minimum weight-cover.

Rounding Rule: if \( x(v) \geq 1/2 \) then round up, otherwise round down.
Invoking an (Integer) Linear Program

Idea: Round the solution of an associated linear program.

0-1 Integer Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v) x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in \{0, 1\} \quad \text{for each } v \in V
\end{align*}
\]

Linear Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v) x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in [0, 1] \quad \text{for each } v \in V
\end{align*}
\]

Rounding Rule: if \( x(v) \geq 1/2 \) then round up, otherwise round down.

optimum is a lower bound on the optimal weight of a minimum weight-cover.
The Algorithm

\textsc{Approx-Min-Weight-VC}(G, w)
1 \hspace{1em} C = \emptyset
2 \hspace{1em} \text{compute } \bar{x}, \text{ an optimal solution to the linear program}
3 \hspace{1em} \text{for each } v \in V
4 \hspace{2em} \text{if } \bar{x}(v) \geq 1/2
5 \hspace{3em} C = C \cup \{v\}
6 \hspace{1em} \text{return } C
The Algorithm

**APPROX-MIN-WEIGHT-VC**($G, w$)

1. $C = \emptyset$
2. compute $\bar{x}$, an optimal solution to the linear program
3. **for** each $v \in V$
4. 
5. **if** $\bar{x}(v) \geq 1/2$
6. 
7. **return** $C$

**Theorem 35.7**

**APPROX-MIN-WEIGHT-VC** is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.
The Algorithm

\begin{algorithm}
\caption{APPROX-MIN-WEIGHT-VC \((G, w)\)}
\begin{algorithmic}[1]
\State \( C = \emptyset \)
\State compute \( \tilde{x} \), an optimal solution to the linear program
\For {each \( v \in V \)}
\If {\( \tilde{x}(v) \geq 1/2 \)}
\State \( C = C \cup \{v\} \)
\EndIf
\EndFor
\State return \( C \)
\end{algorithmic}
\end{algorithm}

**Theorem 35.7**

APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

is polynomial-time because we can solve the linear program in polynomial time
Example of APPROX-MIN-WEIGHT-VC

\[ \overline{x}(a) = \overline{x}(b) = \overline{x}(e) = \frac{1}{2}, \overline{x}(d) = 1, \overline{x}(c) = 0 \]

Fractional solution of LP with weight = 5.5

VI. Randomisation and Rounding Weighted Vertex Cover
Example of **APPROX-MIN-WEIGHT-VC**

\[
\overline{x}(a) = \overline{x}(b) = \overline{x}(e) = \frac{1}{2}, \quad \overline{x}(d) = 1, \quad \overline{x}(c) = 0 \\
\]

\[
x(a) = x(b) = x(e) = 1, \quad x(d) = 1, \quad x(c) = 0 \\
\]

fractional solution of LP with weight = 5.5

rounded solution of LP with weight = 10
Example of APPROX-MIN-WEIGHT-VC

\[
\bar{x}(a) = \bar{x}(b) = \bar{x}(e) = \frac{1}{2}, \bar{x}(d) = 1, \bar{x}(c) = 0
\]

\[
x(a) = x(b) = x(e) = 1, x(d) = 1, x(c) = 0
\]

fractional solution of LP with weight = 5.5
rounded solution of LP with weight = 10
optimal solution with weight = 6
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem. Let $z^*$ be the value of an optimal solution to the linear program, so $z^* \leq w(C^*)$.

Step 1: The computed set $C$ covers all vertices:
Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1 \Rightarrow$ at least one of $x(u)$ and $x(v)$ is at least $1/2 \Rightarrow C$ covers edge $(u, v)$.

Step 2: The computed set $C$ satisfies $w(C) \leq 2z^*$:

$$w(C^*) \geq z^* = \sum_{v \in V} w(v) x(v) \geq \sum_{v \in V} x(v) \geq 1/2 w(v) \cdot 1/2 = 1/2 w(C).$$
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.

Let $z^*$ be the value of an optimal solution to the linear program, so $z^* \leq w(C^*)$.

**Step 1:**
The computed set $C$ covers all vertices:

Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$.

$\Rightarrow$ at least one of $x(u)$ and $x(v)$ is at least $1/2$.

$\Rightarrow C$ covers edge $(u, v)$.

**Step 2:**
The computed set $C$ satisfies $w(C) \leq 2z^*$:

$w(C^*) \geq z^* = \sum_{v \in V} w(v) x(v) \geq \sum_{v \in V} x(v) \geq 1/2 w(v) \cdot 1/2 = 1/4 w(C)$.
Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so

![Diagram](image_url)
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the **minimum-weight vertex cover problem**
- Let $z^*$ be the value of an optimal solution to the linear program, so

$$z^* \leq w(C^*)$$

!![Graph of a vertex cover problem with weights and vertices labeled a, b, c, d, e. The graph is shown before and after rounding.]

- **Step 1:** The computed set $C$ covers all vertices.
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$ $\Rightarrow$ at least one of $x(u)$ and $x(v)$ is at least $1/2$ $\Rightarrow$ $C$ covers edge $(u, v)$.

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:
  - $w(C^*) \geq z^* = \sum_{v \in V} w(v) x(v) \geq \sum_{v \in V}: x(v) \geq 1/2 w(v) \cdot 1/2 = 1/2 w(C^*)$. 

Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

- **Step 1:** The computed set $C$ covers all vertices:

  \[ \text{Rounding} \]

\[ \begin{array}{ccc}
  \text{a} & \text{b} & \text{c} \\
  4 & 3 & 3 \\
  \text{e} & 2 & 1 \\
  \text{d} & & 1 \\
  \hline
  \text{c} & \text{d} & \text{e} \\
  3 & 1 & 2 \\
\end{array} \]
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

**Step 1:** The computed set $C$ covers all vertices:
- Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$

![Diagram showing vertex cover algorithm](image-url)
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

- **Step 1**: The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$
  - $\Rightarrow$ at least one of $\overline{x}(u)$ and $\overline{x}(v)$ is at least $1/2$

VI. Randomisation and Rounding

Weighted Vertex Cover
Proof (Approximation Ratio is 2 and Correctness):

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so

\[ z^* \leq w(C^*) \]

**Step 1:** The computed set $C$ covers all vertices:
- Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$
  \[ \Rightarrow \text{ at least one of } x(u) \text{ and } x(v) \text{ is at least } 1/2 \Rightarrow C \text{ covers edge } (u, v) \]
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so

\[ z^* \leq w(C^*) \]

- **Step 1:** The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$ $\Rightarrow$ at least one of $\overline{x}(u)$ and $\overline{x}(v)$ is at least $1/2$ $\Rightarrow$ $C$ covers edge $(u, v)$.

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$.
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

- **Step 1:** The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$ \[ \Rightarrow \] at least one of $\overline{x}(u)$ and $\overline{x}(v)$ is at least $1/2$ \[ \Rightarrow \] $C$ covers edge $(u, v)$

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
4 & 3 & 3 & \text{Rounding} \\
\text{e} & 2 & & \\
\text{c} & 3 & 1 & \\
\end{array}
\]

VI. Randomisation and Rounding

Weighted Vertex Cover
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

- **Step 1:** The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$ \(\Rightarrow\) at least one of $x(u)$ and $x(v)$ is at least $1/2 \Rightarrow C$ covers edge $(u, v)$

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:
  \[ w(C^*) \geq z^* \]
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

**Step 1:** The computed set $C$ covers all vertices:
- Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$.
  \[ \Rightarrow \ \text{at least one of } x(u) \text{ and } x(v) \text{ is at least } 1/2 \Rightarrow C \text{ covers edge } (u, v) \]

**Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:

\[ w(C^*) \geq z^* = \sum_{v \in V} w(v)\bar{x}(v) \]

Rounding
### Approximation Ratio

**Proof (Approximation Ratio is 2 and Correctness):**

- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem.
- Let $z^*$ be the value of an optimal solution to the linear program, so

\[
z^* \leq w(C^*)
\]

**Step 1:** The computed set $C$ covers all vertices:

- Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1 \Rightarrow$ at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2 \Rightarrow C$ covers edge $(u, v)$

**Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:

\[
w(C^*) \geq z^* = \sum_{v \in V} w(v)\bar{x}(v) \geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2}
\]
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so
  \[ z^* \leq w(C^*) \]

- **Step 1:** The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$\n    $\Rightarrow$ at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$ $\Rightarrow$ $C$ covers edge $(u, v)$

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:
  \[
  w(C^*) \geq z^* = \sum_{v \in V} w(v) \bar{x}(v) \geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \frac{1}{2} w(C).
  \]
Proof (Approximation Ratio is 2 and Correctness):

- Let \( C^* \) be an optimal solution to the minimum-weight vertex cover problem.
- Let \( z^* \) be the value of an optimal solution to the linear program, so

\[ z^* \leq w(C^*) \]

**Step 1:** The computed set \( C \) covers all vertices:
- Consider any edge \((u, v) \in E\) which imposes the constraint \( x(u) + x(v) \geq 1 \)
  \[ \Rightarrow \text{ at least one of } x(u) \text{ and } x(v) \text{ is at least } \frac{1}{2} \Rightarrow C \text{ covers edge } (u, v) \]

**Step 2:** The computed set \( C \) satisfies \( w(C) \leq 2z^* \):

\[
\begin{align*}
w(C^*) & \geq z^* = \sum_{v \in V} w(v)\bar{x}(v) \\
& \geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \frac{1}{2} w(C).
\end{align*}
\]
Approximation Ratio

Proof (Approximation Ratio is 2 and Correctness):
- Let $C^*$ be an optimal solution to the minimum-weight vertex cover problem
- Let $z^*$ be the value of an optimal solution to the linear program, so

$$z^* \leq w(C^*)$$

- **Step 1:** The computed set $C$ covers all vertices:
  - Consider any edge $(u, v) \in E$ which imposes the constraint $x(u) + x(v) \geq 1$  
    $\Rightarrow$ at least one of $x(u)$ and $x(v)$ is at least $1/2$  
    $\Rightarrow$ $C$ covers edge $(u, v)$

- **Step 2:** The computed set $C$ satisfies $w(C) \leq 2z^*$:

$$w(C^*) \geq z^* = \sum_{v \in V} w(v)\bar{x}(v) \geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \frac{1}{2}w(C). \square$$
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
The **Weighted Set-Covering Problem**

Set Cover Problem

- **Given:** set $X$ and a family of subsets $\mathcal{F}$, and a cost function $c : \mathcal{F} \to \mathbb{R}^+$
- **Goal:** Find a minimum-cost subset $C \subseteq \mathcal{F}$ such that
  \[ X = \bigcup_{S \in C} S. \]
The **Weighted Set-Covering Problem**

Given: set $X$ and a family of subsets $F$, and a cost function $c : F \rightarrow \mathbb{R}^+$

Goal: Find a minimum-cost subset $C \subseteq F$

s.t. $X = \bigcup_{S \in C} S$

Set Cover Problem

- **Given**: set $X$ and a family of subsets $F$, and a cost function $c : F \rightarrow \mathbb{R}^+$
- **Goal**: Find a minimum-cost subset $C \subseteq F$

Sum over the costs of all sets in $C$
The **Weighted Set-Covering Problem**

**Set Cover Problem**

- **Given**: set $X$ and a family of subsets $\mathcal{F}$, and a cost function $c : \mathcal{F} \to \mathbb{R}^+$
- **Goal**: Find a minimum-cost subset $\mathcal{C} \subseteq \mathcal{F}$

\[
\text{Sum over the costs of all sets in } \mathcal{C}
\]

\[
X = \bigcup_{S \in \mathcal{C}} S.
\]
The Weighted Set-Covering Problem

Given: set $X$ and a family of subsets $\mathcal{F}$, and a cost function $c : \mathcal{F} \rightarrow \mathbb{R}^+$

Goal: Find a minimum-cost subset $C \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in C} S$

Remarks:
- generalisation of the weighted vertex-cover problem
- models resource allocation problems
Exercise: Try to formulate the integer program and linear program of the weighted SET-COVER problem (solution on next slide!)
Setting up an Integer Program

0-1 Integer Program

minimize \[ \sum_{S \in \mathcal{F}} c(S)y(S) \]

subject to \[ \sum_{S \in \mathcal{F} : x \in S} y(S) \geq 1 \quad \text{for each } x \in X \]
\[ y(S) \in \{0, 1\} \quad \text{for each } S \in \mathcal{F} \]
## Setting up an Integer Program

### 0-1 Integer Program

Minimize: \[ \sum_{S \in \mathcal{F}} c(S) y(S) \]

Subject to:
\[ \sum_{S \in \mathcal{F}: x \in S} y(S) \geq 1 \quad \text{for each } x \in X \]
\[ y(S) \in \{0, 1\} \quad \text{for each } S \in \mathcal{F} \]

### Linear Program

Minimize: \[ \sum_{S \in \mathcal{F}} c(S) y(S) \]

Subject to:
\[ \sum_{S \in \mathcal{F}: x \in S} y(S) \geq 1 \quad \text{for each } x \in X \]
\[ y(S) \in [0, 1] \quad \text{for each } S \in \mathcal{F} \]
Back to the Example

The strategy employed for Vertex-Cover would take all 6 sets!

Even worse: If all y's were below 1/2, we would not even return a valid cover!
Back to the Example

Cost equals 8.5.

The strategy employed for Vertex-Cover would take all 6 sets!

Even worse: If all $y$'s were below $1/2$, we would not even return a valid cover!
The strategy employed for Vertex-Cover would take all 6 sets!

Even worse: If all $y$'s were below $\frac{1}{2}$, we would not even return a valid cover!

Cost equals 8.5
Back to the Example

Cost equals 8.5

The strategy employed for Vertex-Cover would take all 6 sets!
Back to the Example

The strategy employed for Vertex-Cover would take all 6 sets!

Even worse: If all $y$’s were below $1/2$, we would not even return a valid cover!

Cost equals 8.5
Randomised Rounding

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$y(.)$:</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Idea: Interpret the $y$-values as probabilities for picking the respective set.

The expected cost satisfies

$$E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$$

The probability that an element $x \in X$ is covered satisfies

$$\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}.$$
Randomised Rounding

Idea: Interpret the $y$-values as probabilities for picking the respective set.
Randomised Rounding

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$y(\cdot)$</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Idea: Interpret the $y$-values as probabilities for picking the respective set.

Randomised Rounding

- Let $C \subseteq \mathcal{F}$ be a random set with each set $S$ being included independently with probability $y(S)$.
- More precisely, if $y$ denotes the optimal solution of the LP, then we compute an integral solution $\bar{y}$ by:

$$
\bar{y}(S) = \begin{cases} 
1 & \text{with probability } y(S) \\
0 & \text{otherwise.} 
\end{cases} 
$$

for all $S \in \mathcal{F}$.
Randomised Rounding

<table>
<thead>
<tr>
<th></th>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
<th>S₄</th>
<th>S₅</th>
<th>S₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>y(.)</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Idea: Interpret the $y$-values as probabilities for picking the respective set.

- Let $C \subseteq \mathcal{F}$ be a random set with each set $S$ being included independently with probability $y(S)$.
- More precisely, if $y$ denotes the optimal solution of the LP, then we compute an integral solution $\tilde{y}$ by:

$$\tilde{y}(S) = \begin{cases} 1 & \text{with probability } y(S) \\ 0 & \text{otherwise.} \end{cases}$$

- Therefore, $\mathbb{E}[\tilde{y}(S)] = y(S)$. 

Randomised Rounding

The expected cost satisfies $\mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.

The probability that an element $x \in X$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$. 

Lemma VI. Randomisation and Rounding Weighted Set Cover 23
Randomised Rounding

\[
\begin{array}{ccccccc}
S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\
c : & 2 & 3 & 3 & 5 & 1 & 2 \\
y(.) : & 1/2 & 1/2 & 1/2 & 1/2 & 1 & 1/2 \\
\end{array}
\]

Idea: Interpret the \( y \)-values as probabilities for picking the respective set.

Lemma
Randomised Rounding

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$y(.)$</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Idea: Interpret the $y$-values as probabilities for picking the respective set.

Lemma

- The expected cost satisfies

$$
E \left[ c(C) \right] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)
$$
### Idea
Interpret the $y$-values as probabilities for picking the respective set.

### Lemma
- The expected cost satisfies
  \[
  E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)
  \]
- The probability that an element $x \in X$ is covered satisfies
  \[
  \Pr \left[ x \in \bigcup_{S \in C} S \right] \geq 1 - \frac{1}{e}.
  \]
Proof of Lemma

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[ c(C) ] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[ x \in \bigcup_{S \in C} S ] \geq 1 - \frac{1}{e}$.
Proof of Lemma

Let \( C \subseteq F \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( \mathbb{E}[c(C)] = \sum_{S \in F} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C \)
Proof of Lemma

Lemma

Let \( C \subseteq \mathcal{F} \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C \)

\[
E[c(C)]
\]
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $\mathbb{E} [ c(C) ] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr [ x \in \bigcup_{S \in C} S ] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

  $$\mathbb{E} [ c(C) ] = \mathbb{E} \left[ \sum_{S \in C} c(S) \right]$$
Proof of Lemma

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1:** The expected cost of the random set $C$

  \[
  E[c(C)] = E \left[ \sum_{S \in C} c(S) \right] = E \left[ \sum_{S \in F} 1_{S \in C} \cdot c(S) \right]
  \]
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[ c(C) ] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[ x \in \bigcup_{S \in C} S ] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1:** The expected cost of the random set $C$

  \[
  E[ c(C) ] = E \left[ \sum_{S \in C} c(S) \right] = E \left[ \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in \mathcal{F}} \Pr[ S \in C ] \cdot c(S)
  \]
Proof of Lemma

Lemma

Let \( C \subseteq F \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( \mathbb{E}[c(C)] = \sum_{S \in F} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C \)

\[
\mathbb{E}[c(C)] = \mathbb{E}\left[ \sum_{S \in C} c(S) \right] = \mathbb{E}\left[ \sum_{S \in F} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in F} \Pr[S \in C] \cdot c(S) = \sum_{S \in F} y(S) \cdot c(S).
\]
Proof of Lemma

Let \( C \subseteq \mathcal{F} \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( \mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \mathbb{P}[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C \)

  \[
  \mathbb{E}[c(C)] = \mathbb{E}
  \left[
  \sum_{S \in C} c(S)
  \right]
  = \mathbb{E}
  \left[
  \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S)
  \right]
  = \sum_{S \in \mathcal{F}} \mathbb{P}[S \in C] \cdot c(S)
  = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
  \]
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$
  
  $$E[c(C)] = E\left[ \sum_{S \in C} c(S) \right] = E\left[ \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in \mathcal{F}} \Pr[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).$$

- **Step 2**: The probability for an element to be (not) covered
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $\mathbb{E} [ c(C) ] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr \left[ x \in \bigcup_{S \in C} S \right] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

  $$
  \mathbb{E} [ c(C) ] = \mathbb{E} \left[ \sum_{S \in \mathcal{C}} c(S) \right] = \mathbb{E} \left[ \sum_{S \in \mathcal{F}} 1_{S \subseteq \mathcal{C}} \cdot c(S) \right] \\
  = \sum_{S \in \mathcal{F}} \Pr [ S \subseteq \mathcal{C} ] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
  $$

- **Step 2**: The probability for an element to be (not) covered

  $$
  \Pr [ x \notin \bigcup_{S \in C} S ]
  $$
Proof of Lemma

Lemma

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \cup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1:** The expected cost of the random set $C$

  $$E[c(C)] = E\left[\sum_{S \in C} c(S)\right] = E\left[\sum_{S \in F} 1_{S \in C} \cdot c(S)\right] = \sum_{S \in F} \Pr[S \in C] \cdot c(S) = \sum_{S \in F} y(S) \cdot c(S).$$

- **Step 2:** The probability for an element to be (not) covered

  $$\Pr[x \not\in \cup_{S \in C} S] = \prod_{S \in F: x \in S} \Pr[S \not\in C]$$
Proof of Lemma

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

  $$E[c(C)] = E \left[ \sum_{S \in C} c(S) \right] = E \left[ \sum_{S \in F} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in F} \Pr[S \in C] \cdot c(S) = \sum_{S \in F} y(S) \cdot c(S).$$

- **Step 2**: The probability for an element to be (not) covered

  $$\Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in F: x \in S} \Pr[S \notin C] = \prod_{S \in F: x \in S} (1 - y(S))$$
Proof of Lemma

**Lemma**

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

\[
E[c(C)] = E\left[\sum_{S \in C} c(S)\right] = E\left[\sum_{S \in F} 1_{S \in C} \cdot c(S)\right] = \sum_{S \in F} \Pr[S \in C] \cdot c(S) = \sum_{S \in F} y(S) \cdot c(S).
\]

- **Step 2**: The probability for an element to be (not) covered

\[
\Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in F: \ x \in S} \Pr[S \notin C] = \prod_{S \in F: \ x \in S} (1 - y(S)).
\]

$1 + x \leq e^x$ for any $x \in \mathbb{R}$
Proof of Lemma

**Lemma**

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $\mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\mathbb{P}[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

**Proof:**

- **Step 1:** The expected cost of the random set $C$

$$
\mathbb{E}[c(C)] = \mathbb{E}\left[ \sum_{S \in C} c(S) \right] = \mathbb{E}\left[ \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in \mathcal{F}} \mathbb{P}[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
$$

- **Step 2:** The probability for an element to be (not) covered

$$
\mathbb{P}[x \notin \bigcup_{S \in C} S] = \prod_{S \in \mathcal{F}, x \in S} \mathbb{P}[S \notin C] = \prod_{S \in \mathcal{F}, x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F}, x \in S} e^{-y(S)} \leq 1 + x \leq e^x \text{ for any } x \in \mathbb{R}
$$
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

$$E[c(C)] = E \left[ \sum_{S \in C} c(S) \right] = E \left[ \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in \mathcal{F}} \Pr[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).$$

- **Step 2**: The probability for an element to be (not) covered

$$\Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in \mathcal{F}: x \in S} \Pr[S \notin C] = \prod_{S \in \mathcal{F}: x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F}: x \in S} e^{-y(S)} = e^{-\sum_{S \in \mathcal{F}: x \in S} y(S)}$$

$1 + x \leq e^x$ for any $x \in \mathbb{R}$.
Proof of Lemma

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

  \[
  E[c(C)] = E \left[ \sum_{S \in C} c(S) \right] = E \left[ \sum_{S \in F} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in F} \Pr[S \in C] \cdot c(S) = \sum_{S \in F} y(S) \cdot c(S).
  \]

- **Step 2**: The probability for an element to be (not) covered

  \[
  \Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in F: x \in S} \Pr[S \notin C] = \prod_{S \in F: x \in S} (1 - y(S)) \leq \prod_{S \in F: x \in S} e^{-y(S)} \leq e^{-\sum_{S \in F: x \in S} y(S)}
  \]

  $1 + x \leq e^x$ for any $x \in \mathbb{R}$

  $y$ solves the LP!
Proof of Lemma

Lemma

Let \( C \subseteq \mathcal{F} \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( \mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C \)

  \[
  \mathbb{E}[c(C)] = \mathbb{E}\left[ \sum_{S \in C} c(S) \right] = \mathbb{E}\left[ \sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S) \right] = \sum_{S \in \mathcal{F}} \Pr[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
  \]

- **Step 2**: The probability for an element to be (not) covered

  \[
  \Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in \mathcal{F} : x \in S} \Pr[S \notin C] = \prod_{S \in \mathcal{F} : x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F} : x \in S} e^{-y(S)} = e^{-\sum_{S \in \mathcal{F} : x \in S} y(S)} \leq e^{-1}.
  \]

\( y \) solves the LP!
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$

  
  \[
  E[c(C)] = E\left[\sum_{S \in C} c(S)\right] = E\left[\sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S)\right] = \sum_{S \in \mathcal{F}} \Pr[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
  \]

- **Step 2**: The probability for an element to be (not) covered

  \[
  \Pr[x \notin \bigcup_{S \in \mathcal{C}} S] = \prod_{S \in \mathcal{F}: x \in S} \Pr[S \notin C] = \prod_{S \in \mathcal{F}: x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F}: x \in S} e^{-y(S)} = e^{-\sum_{S \in \mathcal{F}: x \in S} y(S)} \leq e^{-1}
  \]

  $1 + x \leq e^x$ for any $x \in \mathbb{R}$

  $y$ solves the LP!
Proof of Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $\mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Proof:

- **Step 1**: The expected cost of the random set $C$ holds
  \[
  \mathbb{E}[c(C)] = \mathbb{E}\left[\sum_{S \in C} c(S)\right] = \mathbb{E}\left[\sum_{S \in \mathcal{F}} 1_{S \in C} \cdot c(S)\right] = \sum_{S \in \mathcal{F}} \Pr[S \in C] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
  \]

- **Step 2**: The probability for an element to be (not) covered
  \[
  \Pr[x \notin \bigcup_{S \in C} S] = \prod_{S \in \mathcal{F} : x \in S} \Pr[S \notin C] = \prod_{S \in \mathcal{F} : x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F} : x \in S} e^{-y(S)} = e^{-\sum_{S \in \mathcal{F} : x \in S} y(S)} \leq e^{-1}.
  \]

$y$ solves the LP!
The Final Step

Let \( C \subseteq \mathcal{F} \) be a **random subset** with each set \( S \) being included independently with probability \( y(S) \).

- The **expected cost** satisfies
  \[
  \mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S).
  \]
- The **probability** that \( x \) is **covered** satisfies
  \[
  \Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}.
  \]
The Final Step

**Lemma**

Let \( C \subseteq F \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( \mathbb{E}[c(C)] = \sum_{S \in F} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[ x \in \bigcup_{S \in C} S ] \geq 1 - \frac{1}{e} \).

**Problem:** Need to make sure that every element is covered!
The Final Step

**Lemma**

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

**Problem:** Need to make sure that every element is covered!

**Idea:** Amplify this probability by taking the union of $\Omega(\log n)$ random sets $C$. 

VI. Randomisation and Rounding Weighted Set Cover
The Final Step

Let $C \subseteq F$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in F} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

**Problem:** Need to make sure that every element is covered!

**Idea:** Amplify this probability by taking the union of $\Omega(\log n)$ random sets $C$.

**Algorithm**

```
WEIGHTED SET COVER-LP($X, F, c$)
1: compute $y$, an optimal solution to the linear program
2: $C = \emptyset$
3: repeat $2 \ln n$ times
4: for each $S \in F$
5: let $C = C \cup \{S\}$ with probability $y(S)$
6: return $C$
```
The Final Step

Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $E[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

Problem: Need to make sure that every element is covered!

Idea: Amplify this probability by taking the union of $\Omega(\log n)$ random sets $C$.

**WEIGHTED SET COVER-LP($X$, $\mathcal{F}$, $c$)**

1: compute $y$, an optimal solution to the linear program
2: $C = \emptyset$
3: repeat $2 \ln n$ times
4: for each $S \in \mathcal{F}$
5: let $C = C \cup \{S\}$ with probability $y(S)$
6: return $C$

clearly runs in polynomial-time!
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.
With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.

The expected approximation ratio is $2 \ln(n)$.

Proof:
Theorem
- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:
- **Step 1**: The probability that $C$ is a cover
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1**: The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
Analysis of Weighted Set Cover-LP

--- Theorem ---

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

**Proof:**

- **Step 1:** The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    
    $$\Pr [ x \not\in \bigcup_{S \in C} S ] \le \left( \frac{1}{e} \right)^{2 \ln n}$$
Analysis of Weighted Set Cover - LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

**Step 1**: The probability that $C$ is a cover

- By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

$$
\Pr [x \notin \bigcup_{S \in C} S] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.
$$
Analysis of Weighted Set Cover - LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1**: The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    $$\Pr \left[ x \notin \bigcup_{S \in C} S \right] \leq \left( \frac{1}{e} \right)^{2\ln n} = \frac{1}{n^2}.$$  
  - This implies for the event that all elements are covered:
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1**: The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[
    \Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.
    \]
  - This implies for the event that all elements are covered:
    \[
    \Pr[X = \bigcup_{S \in C} S] =
    \]
Analysis of Weighted Set Cover -LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

**Step 1:** The probability that $C$ is a cover

- By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

$$\Pr [ x \notin \bigcup_{S \in C} S ] \leq \left( \frac{1}{e} \right)^{2\ln n} = \frac{1}{n^2}.$$  

- This implies for the event that all elements are covered:

$$\Pr [ X = \bigcup_{S \in C} S ] = 1 - \Pr \left[ \bigcup_{x \in X} \{ x \notin \bigcup_{S \in C} S \} \right].$$
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

**Step 1:** The probability that $C$ is a cover

- By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

$$\Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.$$  

- This implies for the event that all elements are covered:

$$\Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[\bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\}\right].$$

**Pr [A ∪ B] ≤ Pr [A] + Pr [B]**
Analysis of **Weighted Set Cover** -LP

**Theorem**

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

**Proof:**

**Step 1**: The probability that $C$ is a cover

- By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

  $$\Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.$$  

- This implies for the event that all elements are covered:

  $$\Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[\bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\}\right]$$

**VI. Randomisation and Rounding**

Weighted Set Cover

26
Analysis of WEIGHTED SET COVER -LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1:** The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[ \Pr[x \notin \bigcup_{S \in C} S] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}. \]
  - This implies for the event that all elements are covered:
    \[ \Pr[X = \bigcup_{S \in C} S] = 1 - \Pr \left[ \bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\} \right]. \]

\[
\Pr[A \cup B] \leq \Pr[A] + \Pr[B] \geq 1 - \sum_{x \in X} \Pr[x \notin \bigcup_{S \in C} S] \geq 1 - n \cdot \frac{1}{n^2}.
\]
With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.

The expected approximation ratio is $2 \ln(n)$.

Proof:

**Step 1:** The probability that $C$ is a cover

By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

$$\Pr\left[ x \notin \bigcup_{S \in C} S \right] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.$$ 

This implies for the event that all elements are covered:

$$\Pr\left[ X = \bigcup_{S \in C} S \right] = 1 - \Pr\left[ \bigcup_{x \in X} \{ x \notin \bigcup_{S \in C} S \} \right] \geq 1 - \sum_{x \in X} \Pr\left[ x \notin \bigcup_{S \in C} S \right] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.$$
### Analysis of Weighted Set Cover-LP

**Theorem**
- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

**Proof:**
- **Step 1**: The probability that $C$ is a cover $\checkmark$
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    
    $$
    \Pr [ x \notin \bigcup_{S \in C} S ] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.
    $$
  - This implies for the event that all elements are covered:
    $$
    \Pr [ X = \bigcup_{S \in C} S ] = 1 - \Pr \left[ \bigcup_{x \in X} \{ x \notin \bigcup_{S \in C} S \} \right] \geq 1 - \sum_{x \in X} \Pr [ x \notin \bigcup_{S \in C} S ] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
    $$

**VI. Randomisation and Rounding**

Weighted Set Cover
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1:** The probability that $C$ is a cover $✓$
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[
    \Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2 \ln n} = \frac{1}{n^2}.
    \]
  - This implies for the event that all elements are covered:
    \[
    \Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[\bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\}\right].
    \]

- **Step 2:** The expected approximation ratio
  \[
  \Pr[A \cup B] \leq \Pr[A] + \Pr[B] \geq 1 - \sum_{x \in X} \Pr[x \notin \bigcup_{S \in C} S] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
  \]
Analysis of Weighted Set Cover -LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

**Step 1:** The probability that $C$ is a cover

- By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that

\[
\Pr \left[ x \notin \bigcup_{S \in C} S \right] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.
\]

- This implies for the event that all elements are covered:

\[
\Pr \left[ X = \bigcup_{S \in C} S \right] = 1 - \Pr \left[ \bigcup_{x \in X} \{ x \notin \bigcup_{S \in C} S \} \right] 
\]

\[
\Pr [ A \cup B ] \leq \Pr [ A ] + \Pr [ B ] \geq 1 - \sum_{x \in X} \Pr [ x \notin \bigcup_{S \in C} S ] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
\]

**Step 2:** The expected approximation ratio

- By previous lemma, the expected cost of one iteration is $\sum_{S \in \mathcal{F}} c(S) \cdot y(S)$. 

VI. Randomisation and Rounding

Weighted Set Cover
Analysis of Weighted Set Cover-LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1**: The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[
    \Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.
    \]
  - This implies for the event that all elements are covered:
    \[
    \Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[\bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\}\right] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
    \]

- **Step 2**: The expected approximation ratio
  - By previous lemma, the expected cost of one iteration is $\sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
  - Linearity $\Rightarrow E[c(C)] \leq 2 \ln(n) \cdot \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$
Analysis of Weighted Set Cover - LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1:** The probability that $C$ is a cover $\checkmark$
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[
    \Pr[x \notin \bigcup_{S \in C} S] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.
    \]
    - This implies for the event that all elements are covered:
      \[
      \Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[ \bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\} \right] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
      \]

- **Step 2:** The expected approximation ratio
  - By previous lemma, the expected cost of one iteration is $\sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
  - Linearity $\Rightarrow E[c(C)] \leq 2 \ln(n) \cdot \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \leq 2 \ln(n) \cdot c(C^*)$.
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1:** The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
  $$\Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.$$  
  - This implies for the event that all elements are covered:
    $$\Pr[X = \bigcup_{S \in C} S] = 1 - \Pr\left[\bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\}\right] \geq 1 - \frac{1}{n^2}.$$  

- **Step 2:** The expected approximation ratio
  - By previous lemma, the expected cost of one iteration is $\sum_{S \in F} c(S) \cdot y(S)$.
  - Linearity $\Rightarrow E[c(C)] \leq 2 \ln(n) \cdot \sum_{S \in F} c(S) \cdot y(S) \leq 2 \ln(n) \cdot c(C^*)$
**Analysis of Weighted Set Cover-LP**

**Theorem**
- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

**Proof:**
- **Step 1:** The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    $$\Pr[x \notin \bigcup_{S \in C} S] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}.$$
  - This implies for the event that all elements are covered:
    $$\Pr[X = \bigcup_{S \in C} S] = 1 - \Pr \left[ \bigcup_{x \in X} \{x \notin \bigcup_{S \in C} S\} \right]$$

- **Step 2:** The expected approximation ratio
  - By previous lemma, the expected cost of one iteration is $\sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
  - Linearity $\Rightarrow E[c(C)] \leq 2 \ln(n) \cdot \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \leq 2 \ln(n) \cdot c(C^*)$
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

By Markov’s inequality, $\Pr \left[ c(C) \leq 4 \ln(n) \cdot c(C^*) \right] \geq 1/2$. 
Analysis of Weighted Set Cover - LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

By Markov's inequality, $\Pr[c(C) \leq 4 \ln(n) \cdot c(C^*)] \geq 1/2$.

Hence with probability at least $1 - \frac{1}{n} - \frac{1}{2} > \frac{1}{3}$, solution is within a factor of $4 \ln(n)$ of the optimum.
With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.

The expected approximation ratio is $2 \ln(n)$.

By Markov’s inequality, $\Pr[ c(C) \leq 4 \ln(n) \cdot c(C^*) ] \geq 1/2$.

Hence with probability at least $1 - \frac{1}{n} - \frac{1}{2} > \frac{1}{3}$, solution is within a factor of $4 \ln(n)$ of the optimum.

Probability could be further increased by repeating
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

By Markov’s inequality, $\Pr[ c(C) \leq 4 \ln(n) \cdot c(C^*) ] \geq 1/2$.

Hence with probability at least $1 - \frac{1}{n} - \frac{1}{2} > \frac{1}{3}$, solution is within a factor of $4 \ln(n)$ of the optimum.

Probability could be further increased by repeating.

Typical Approach for Designing Approximation Algorithms based on LPs
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
Recall:

**MAX-3-CNF Satisfiability**

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

**MAX-CNFSatisfiability (MAX-SAT)**
MAX-CNF

Recall:

MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

MAX-CNF Satisfiability (MAX-SAT)

- **Given:** CNF formula, e.g.: \((x_1 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_4 \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.
Recall:

MAX-3-CNF Satisfiability

- **Given**: 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal**: Find an assignment of the variables that satisfies as many clauses as possible.

MAX-CNF Satisfiability (MAX-SAT)

- **Given**: CNF formula, e.g.: \((x_1 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_4 \lor \overline{x_5}) \land \cdots\)
- **Goal**: Find an assignment of the variables that satisfies as many clauses as possible.

Why study this generalised problem?
Recall:

**MAX-3-CNF Satisfiability**
- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

**MAX-CNF Satisfiability (MAX-SAT)**
- **Given:** CNF formula, e.g.: \((x_1 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_4 \lor \overline{x_5}) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Why study this generalised problem?
- Allowing arbitrary clause lengths makes the problem more interesting (we will see that simply guessing is not the best!)
- A nice concluding example where we can practice previously learned approaches
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!

For any clause $i$ which has length $\ell$, $\Pr[\text{clause $i$ is satisfied}] = 1 - 2^{-\ell} = \alpha \ell$.

In particular, the guessing algorithm is a randomised 2-approximation.

Analysis

Proof:

First statement as in the proof of Theorem 35.6. For clause $i$ not to be satisfied, all $\ell$ occurring variables must be set to a specific value.

As before, let $Y := \sum_{i=1}^{m} Y_i$ be the number of satisfied clauses. Then,

$$E[Y] = E[m \sum_{i=1}^{m} Y_i] \geq m \sum_{i=1}^{m} E[Y_i] = \frac{1}{2} m.$$
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!

For any clause $i$ which has length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] = 1 - 2^{-\ell} := \alpha_\ell.$$ 

In particular, the guessing algorithm is a randomised 2-approximation.
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!

Analysis

For any clause $i$ which has length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] = 1 - 2^{-\ell} := \alpha_\ell.$$  

In particular, the guessing algorithm is a randomised 2-approximation.

Proof:
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!

Analysis

For any clause \( i \) which has length \( \ell \),

\[
\Pr \left[ \text{clause } i \text{ is satisfied} \right] = 1 - 2^{-\ell} := \alpha_\ell.
\]

In particular, the guessing algorithm is a randomised 2-approximation.

Proof:

- First statement as in the proof of Theorem 35.6. For clause \( i \) not to be satisfied, all \( \ell \) occurring variables must be set to a specific value.
Approach 1: Guessing the Assignment

Assign each variable true or false uniformly and independently at random.

Recall: This was the successful approach to solve MAX-3-CNF!

For any clause $i$ which has length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] = 1 - 2^{-\ell} := \alpha_\ell.$$ 

In particular, the guessing algorithm is a randomised 2-approximation.

Analysis

For any clause $i$ which has length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] = 1 - 2^{-\ell} := \alpha_\ell.$$ 

In particular, the guessing algorithm is a randomised 2-approximation.

Proof:

- First statement as in the proof of Theorem 35.6. For clause $i$ not to be satisfied, all $\ell$ occurring variables must be set to a specific value.
- As before, let $Y := \sum_{i=1}^{m} Y_i$ be the number of satisfied clauses. Then,

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] \geq \sum_{i=1}^{m} \frac{1}{2} = \frac{1}{2} \cdot m.$$ 

$\square$
First solve a linear program and use fractional values for a **biased** coin flip.
Approach 2: Guessing with a “Hunch” (Randomised Rounding)

First solve a linear program and use fractional values for a biased coin flip.

The same as randomised rounding!
Approach 2: Guessing with a “Hunch” (Randomised Rounding)

First solve a linear program and use fractional values for a biased coin flip.

The same as randomised rounding!

0-1 Integer Program

maximize \[ \sum_{i=1}^{m} z_i \]
subject to \[ \sum_{j \in C_i^+} y_j + \sum_{j \in C_i^-} (1 - y_j) \geq z_i \quad \text{for each } i = 1, 2, \ldots, m \]
\[ z_i \in \{0, 1\} \quad \text{for each } i = 1, 2, \ldots, m \]
\[ y_j \in \{0, 1\} \quad \text{for each } j = 1, 2, \ldots, n \]
Approach 2: Guessing with a “Hunch” (Randomised Rounding)

First solve a linear program and use fractional values for a biased coin flip.

The same as randomised rounding!

0-1 Integer Program

maximize \( \sum_{i=1}^{m} z_i \)

subject to \( \sum_{j \in C_i^+} y_j + \sum_{j \in C_i^-} (1 - y_j) \geq z_i \) for each \( i = 1, 2, \ldots, m \)

\( z_i \in \{0, 1\} \) for each \( i = 1, 2, \ldots, m \)

\( y_j \in \{0, 1\} \) for each \( j = 1, 2, \ldots, n \)

\( C_i^+ \) is the index set of the un-negated variables of clause \( i \).
**Approach 2: Guessing with a “Hunch” (Randomised Rounding)**

First solve a linear program and use fractional values for a **biased** coin flip.

The same as **randomised rounding**!

**0-1 Integer Program**

maximize \[ \sum_{i=1}^{m} z_i \]

subject to \[ \sum_{j \in C_i^+} y_j + \sum_{j \in C_i^-} (1 - y_j) \geq z_i \text{ for each } i = 1, 2, \ldots, m \]

\( z_i \in \{0, 1\} \text{ for each } i = 1, 2, \ldots, m \)

\( y_j \in \{0, 1\} \text{ for each } j = 1, 2, \ldots, n \)

\( C_i^+ \) is the index set of the un-negated variables of clause \( i \).
Approach 2: Guessing with a “Hunch” (Randomised Rounding)

First solve a linear program and use fractional values for a **biased** coin flip.

The same as **randomised rounding**!

0-1 Integer Program

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{m} z_i \\
\text{subject to} & \quad \sum_{j \in C_i^+} y_j + \sum_{j \in C_i^-} (1 - y_j) \geq z_i \quad \text{for each } i = 1, 2, \ldots, m \\
& \quad z_i \in \{0, 1\} \quad \text{for each } i = 1, 2, \ldots, m \\
& \quad y_j \in \{0, 1\} \quad \text{for each } j = 1, 2, \ldots, n
\end{align*}
\]

- \(C_i^+\) is the index set of the un-negated variables of clause \(i\).

In the **corresponding LP** each \(\in \{0, 1\}\) is replaced by \(\in [0, 1]\)

- Let \((y^*, z^*)\) be the optimal solution of the LP
- Obtain an integer solution \(y\) through randomised rounding of \(y^*\)
For any clause $i$ of length $\ell$, 
\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.
\]
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$ 

Proof of Lemma (1/2):
Analysis of Randomised Rounding

Lemma

For any clause \( i \) of length \( \ell \),

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.
\]

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause \( i \) appear non-negated (otherwise replace every occurrence of \( x_j \) by \( \overline{x}_j \) in the whole formula)
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr \left[ \text{clause } i \text{ is satisfied} \right] \geq \left( 1 - \left( 1 - \frac{1}{\ell} \right)^\ell \right) \cdot z_i^*.$$

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated (otherwise replace every occurrence of $x_j$ by $\overline{x_j}$ in the whole formula)

- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$
Analysis of Randomised Rounding

Lemma

For any clause \( i \) of length \( \ell \),

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z^*_i.
\]

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause \( i \) appear non-negated (otherwise replace every occurrence of \( x_j \) by \( \overline{x_j} \) in the whole formula)
- Further, by relabelling assume \( C_i = (x_1 \lor \cdots \lor x_\ell) \)

\[
\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] =
\]
Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.$$

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated (otherwise replace every occurrence of $x_j$ by $\overline{x_j}$ in the whole formula)
- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$

$$\Rightarrow \Pr[\text{clause } i \text{ is satisfied}] = 1 - \prod_{j=1}^\ell \Pr[\ y_j \text{ is false }]$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$  

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated (otherwise replace every occurrence of $x_j$ by $\overline{x_j}$ in the whole formula).
- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$

$$\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \prod_{j=1}^{\ell} \Pr[ y_j \text{ is false } ] = 1 - \prod_{j=1}^{\ell} \left(1 - y_j^*\right)$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$, 

$$
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.
$$

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated (otherwise replace every occurrence of $x_j$ by $\overline{x_j}$ in the whole formula)
- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$

$$
\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \prod_{j=1}^{\ell} \Pr[\ \overline{y_j} \text{ is false }] = 1 - \prod_{j=1}^{\ell} (1 - y_j^*)
$$

Arithmetic vs. geometric mean:

$$
\frac{a_1 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 \times \cdots \times a_k}.
$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left( 1 - \left( 1 - \frac{1}{\ell} \right)^{\ell} \right) \cdot z_i^*.$$ 

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated 
  (otherwise replace every occurrence of $x_j$ by $\overline{x}_j$ in the whole formula)
- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$

$$\Rightarrow \quad \Pr[\text{clause } i \text{ is satisfied}] = 1 - \prod_{j=1}^{\ell} \Pr[\text{ }y_j \text{ is false }] = 1 - \prod_{j=1}^{\ell} (1 - y_j^*)$$

Arithmetic vs. geometric mean:

$$\frac{a_1 + \ldots + a_k}{k} \geq \sqrt[k]{a_1 \times \ldots \times a_k}.$$
Analysis of Randomised Rounding

Lemma

For any clause \( i \) of length \( \ell \),

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.
\]

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause \( i \) appear non-negated (otherwise replace every occurrence of \( x_j \) by \( \overline{x_j} \) in the whole formula)
- Further, by relabelling assume \( C_i = (x_1 \lor \cdots \lor x_\ell) \)

\[
\Rightarrow \Pr[\text{clause } i \text{ is satisfied}] = 1 - \prod_{j=1}^{\ell} \Pr[\text{ } y_j \text{ is false }] = 1 - \prod_{j=1}^{\ell} \left(1 - y_j^*\right)
\]

Arithmetic vs. geometric mean:

\[
\frac{a_1 + \ldots + a_k}{k} \geq \sqrt[k]{a_1 \times \ldots \times a_k}.
\]

\[
\geq 1 - \left(\frac{\sum_{j=1}^{\ell} (1 - y_j^*)}{\ell}\right)^\ell
\]

\[
= 1 - \left(1 - \frac{\sum_{j=1}^{\ell} y_j^*}{\ell}\right)^\ell
\]
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr \left[ \text{clause } i \text{ is satisfied} \right] \geq \left( 1 - \left( 1 - \frac{1}{\ell} \right)^\ell \right) \cdot z_i^*.$$

Proof of Lemma (1/2):

- Assume w.l.o.g. all literals in clause $i$ appear non-negated (otherwise replace every occurrence of $x_j$ by $\overline{x_j}$ in the whole formula).
- Further, by relabelling assume $C_i = (x_1 \lor \cdots \lor x_\ell)$

$$\Rightarrow \Pr \left[ \text{clause } i \text{ is satisfied} \right] = 1 - \prod_{j=1}^{\ell} \Pr \left[ y_j \text{ is false} \right] = 1 - \prod_{j=1}^{\ell} \left( 1 - y_j^* \right)$$

Arithmetic vs. geometric mean:

$$\frac{a_1 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 \times \cdots \times a_k}.$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.$$

Proof of Lemma (2/2):

- So far we have shown:

$$\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^\ell$$
Lemma

For any clause $i$ of length $\ell$,

$$
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z^*_i.
$$

Proof of Lemma (2/2):

- So far we have shown:

$$
\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z^*_i}{\ell}\right)^{\ell}
$$

- For any $\ell \geq 1$, define $g(z) := 1 - \left(1 - \frac{z}{\ell}\right)^{\ell}$. 
Analysis of Randomised Rounding

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$

Proof of Lemma (2/2):
- So far we have shown:

$$\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^{\ell}.$$

- For any $\ell \geq 1$, define $g(z) := 1 - (1 - \frac{z}{\ell})^{\ell}$. This is a concave function with $g(0) = 0$ and $g(1) = 1 - \left(1 - \frac{1}{\ell}\right)^{\ell} =: \beta_{\ell}$. 
Analysis of Randomised Rounding

Lemma
For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.$$ 

Proof of Lemma (2/2):

- So far we have shown:

$$\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^\ell$$

- For any $\ell \geq 1$, define $g(z) := 1 - (1 - \frac{z}{\ell})^\ell$. This is a concave function with $g(0) = 0$ and $g(1) = 1 - \left(1 - \frac{1}{\ell}\right)^\ell =: \beta_\ell$. 

$$g(z)$$

$1 - (1 - \frac{1}{3})^3$

$0$ $1$ $z$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.$$

Proof of Lemma (2/2):

- So far we have shown:

$$\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^\ell$$

- For any $\ell \geq 1$, define $g(z) := 1 - (1 - \frac{z}{\ell})^\ell$. This is a concave function with $g(0) = 0$ and $g(1) = 1 - \left(1 - \frac{1}{\ell}\right)^\ell =: \beta_\ell$.

$$\Rightarrow g(z) \geq \beta_\ell \cdot z \quad \text{for any } z \in [0, 1] \quad 1 - (1 - \frac{1}{3})^3$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$

Proof of Lemma (2/2):

- So far we have shown:

  $$\Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left(1 - \frac{z_i^*}{\ell}\right)^{\ell}.$$

- For any $\ell \geq 1$, define $g(z) := 1 - \left(1 - \frac{z}{\ell}\right)^{\ell}$. This is a concave function with $g(0) = 0$ and $g(1) = 1 - \left(1 - \frac{1}{\ell}\right)^{\ell} =: \beta_\ell$. Therefore, $g(z) \geq \beta_\ell \cdot z$ for any $z \in [0, 1]$.  

  $$\Rightarrow g(z) \geq \beta_\ell \cdot z \quad \text{for any } z \in [0, 1].$$

- Therefore, $\Pr[\text{clause } i \text{ is satisfied}] \geq \beta_\ell \cdot z_i^*$. 

VI. Randomisation and Rounding MAX-CNF 31
Analysis of Randomised Rounding

**Lemma**

For any clause $i$ of length $\ell$,

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left( 1 - \left( 1 - \frac{1}{\ell} \right)^{\ell} \right) \cdot z_i^*.
\]

**Proof of Lemma (2/2):**

- So far we have shown:
  \[
  \Pr[\text{clause } i \text{ is satisfied}] \geq 1 - \left( 1 - \frac{z_i^*}{\ell} \right)^{\ell}
  \]

- For any $\ell \geq 1$, define $g(z) := 1 - \left( 1 - \frac{z}{\ell} \right)^{\ell}$. This is a concave function with $g(0) = 0$ and $g(1) = 1 - \left( 1 - \frac{1}{\ell} \right)^{\ell} =: \beta_\ell$.

  \[
  \Rightarrow \quad g(z) \geq \beta_\ell \cdot z \quad \text{for any } z \in [0, 1]
  \]

- Therefore, $\Pr[\text{clause } i \text{ is satisfied}] \geq \beta_\ell \cdot z_i^*$. \qed
Analysis of Randomised Rounding

**Lemma**
For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.$$

**Theorem**
Randomised Rounding yields a $1/(1 - 1/e) \approx 1.5820$ randomised approximation algorithm for MAX-CNF.
Analysis of Randomised Rounding

Lemma
For any clause $i$ of length $\ell$,

$$\Pr \left[ \text{clause } i \text{ is satisfied} \right] \geq \left( 1 - \left( 1 - \frac{1}{\ell} \right) \right) \cdot z_i^*.$$  

Theorem
Randomised Rounding yields a $1/(1 - 1/e) \approx 1.5820$ randomised approximation algorithm for MAX-CNF.

Proof of Theorem:
Analysis of Randomised Rounding

**Lemma**

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$

**Theorem**

Randomised Rounding yields a $1/(1 - 1/e) \approx 1.5820$ randomised approximation algorithm for MAX-CNF.

**Proof of Theorem:**

- For any clause $i = 1, 2, \ldots, m$, let $\ell_i$ be the corresponding length.
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$ 

Theorem

Randomised Rounding yields a $1/(1 - 1/e) \approx 1.5820$ randomised approximation algorithm for MAX-CNF.

Proof of Theorem:

- For any clause $i = 1, 2, \ldots, m$, let $\ell_i$ be the corresponding length.
- Then the expected number of satisfied clauses is:

$$\mathbb{E}[Y] = \sum_{i=1}^{m} \mathbb{E}[Y_i] \geq \ldots$$
Analysis of Randomised Rounding

Lemma

For any clause $i$ of length $\ell$,

$$\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.$$ 

Theorem

Randomised Rounding yields a $1/(1 - 1/e) \approx 1.5820$ randomised approximation algorithm for MAX-CNF.

Proof of Theorem:

- For any clause $i = 1, 2, \ldots, m$, let $\ell_i$ be the corresponding length.
- Then the expected number of satisfied clauses is:

$$E[\mathcal{Y}] = \sum_{i=1}^{m} E[\mathcal{Y}_i] \geq \sum_{i=1}^{m} \left(1 - \left(1 - \frac{1}{\ell_i}\right)^{\ell_i}\right) \cdot z_i^*$$

By Lemma
Analysis of Randomised Rounding

Lemma

For any clause \( i \) of length \( \ell \),

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right) \cdot z_i^*.
\]

Theorem

Randomised Rounding yields a \( 1/(1 - 1/e) \approx 1.5820 \) randomised approximation algorithm for MAX-CNF.

Proof of Theorem:

- For any clause \( i = 1, 2, \ldots, m \), let \( \ell_i \) be the corresponding length.
- Then the expected number of satisfied clauses is:

\[
E[Y] = \sum_{i=1}^{m} E[Y_i] \geq \sum_{i=1}^{m} \left(1 - \left(1 - \frac{1}{\ell_i}\right)^{\ell_i}\right) \cdot z_i^* \geq \sum_{i=1}^{m} \left(1 - \frac{1}{e}\right) \cdot z_i^ *
\]

By Lemma

Since \((1 - 1/x)^x \leq 1/e\)
Analysis of Randomised Rounding

Lemma

For any clause \( i \) of length \( \ell \),

\[
\Pr[\text{clause } i \text{ is satisfied}] \geq \left(1 - \left(1 - \frac{1}{\ell}\right)^\ell\right) \cdot z_i^*.
\]

Theorem

Randomised Rounding yields a \( 1/(1 - 1/e) \approx 1.5820 \) randomised approximation algorithm for MAX-CNF.

Proof of Theorem:

- For any clause \( i = 1, 2, \ldots, m \), let \( \ell_i \) be the corresponding length.
- Then the expected number of satisfied clauses is:

\[
E[Y] = \sum_{i=1}^{m} E[Y_i] \geq \sum_{i=1}^{m} \left(1 - \left(1 - \frac{1}{\ell_i}\right)^{\ell_i}\right) \cdot z_i^* \geq \sum_{i=1}^{m} \left(1 - \frac{1}{e}\right) \cdot z_i^* \geq \left(1 - \frac{1}{e}\right) \cdot \text{OPT}
\]

By Lemma  
Since \( (1 - 1/x)^x \leq 1/e \)  
LP solution at least as good as optimum
### Summary

- Approach 1 (Guessing) achieves better guarantee on longer clauses.
- Approach 2 (Rounding) achieves better guarantee on shorter clauses.
Approach 3: Hybrid Algorithm

Summary
- Approach 1 (Guessing) achieves better guarantee on longer clauses
- Approach 2 (Rounding) achieves better guarantee on shorter clauses

Idea: Consider a hybrid algorithm which interpolates between the two approaches

Algorithm sets each variable $x_i$ to TRUE with prob. $\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot y^*_i$. Note, however, that variables are not independently assigned!
Approach 3: Hybrid Algorithm

Summary

- Approach 1 (Guessing) achieves better guarantee on longer clauses
- Approach 2 (Rounding) achieves better guarantee on shorter clauses

Idea:
Consider a hybrid algorithm which interpolates between the two approaches

HYBRID-MAX-CNF(\varphi, n, m)
1: Let \( b \in \{0, 1\} \) be the flip of a fair coin
2: If \( b = 0 \) then perform random guessing
3: If \( b = 1 \) then perform randomised rounding
4: return the computed solution
Approach 3: Hybrid Algorithm

Summary
- Approach 1 (Guessing) achieves better guarantee on longer clauses
- Approach 2 (Rounding) achieves better guarantee on shorter clauses

Idea: Consider a hybrid algorithm which interpolates between the two approaches

\[
\text{HYBRID-MAX-CNF}(\varphi, n, m)
\]
1. Let \( b \in \{0, 1\} \) be the flip of a fair coin
2. If \( b = 0 \) then perform random guessing
3. If \( b = 1 \) then perform randomised rounding
4. return the computed solution

Algorithm sets each variable \( x_i \) to TRUE with prob. \( \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot y_i^* \).
Note, however, that variables are not independently assigned!
Theorem

\textbf{HYBRID-MAX-CNF}(\varphi, n, m) is a randomised 4/3-approx. algorithm.
HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z^*\).

For any clause \(i\) of length \(\ell\):

- Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha \ell\).
- Algorithm 2 satisfies it with probability \(\beta \ell \cdot z^*\).

HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \alpha \ell \cdot z^* + \frac{1}{2} \beta \ell \cdot z^*\).

Note that \(\alpha \ell + \beta \ell^2 = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\alpha \ell + \beta \ell^2 \geq 3/4\) (see figure)

\(\Rightarrow\) HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with prob. at least \(3/4 \cdot z^*\).
Analysis of Hybrid Algorithm

Theorem

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z_i^*\).
Analysis of Hybrid Algorithm

Theorem

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z^*_i\).
- For any clause \(i\) of length \(\ell\):

\[\text{Algorithm 1 satisfies it with probability } 1 - 2^{-\ell} = \alpha \ell \geq \alpha \ell \cdot z^*_i.\]

\[\text{Algorithm 2 satisfies it with probability } \beta \ell \cdot z^*_i.\]

HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha \ell \cdot z^*_i + \frac{1}{2} \cdot \beta \ell \cdot z^*_i.\)

Note \(\alpha \ell + \beta \ell^2 = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\alpha \ell + \beta \ell^2 \geq 3/4\) (see figure).
HYBRID-MAX-CNF(ϕ, n, m) is a randomised 4/3-approx. algorithm.

Proof:
- It suffices to prove that clause \( i \) is satisfied with probability at least \( 3/4 \cdot z_i^* \).
- For any clause \( i \) of length \( \ell \):
  - Algorithm 1 satisfies it with probability \( 1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^* \).
Analysis of Hybrid Algorithm

Theorem

HYBRID-MAX-CNF(ϕ, n, m) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \( i \) is satisfied with probability at least \( 3/4 \cdot z_i^* \).
- For any clause \( i \) of length \( \ell \):
  - Algorithm 1 satisfies it with probability \( 1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^* \).
  - Algorithm 2 satisfies it with probability \( \beta_\ell \cdot z_i^* \).
Analysis of Hybrid Algorithm

**Theorem**

*HYBRID-MAX-CNF*(ϕ, n, m) is a randomised 4/3-approx. algorithm.

**Proof:**

- It suffices to prove that clause $i$ is satisfied with probability at least $3/4 \cdot z_i^*$.
- For any clause $i$ of length $\ell$:
  - Algorithm 1 satisfies it with probability $1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*$.
  - Algorithm 2 satisfies it with probability $\beta_\ell \cdot z_i^*$.
  - *HYBRID-MAX-CNF*(ϕ, n, m) satisfies it with probability $\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*$.

Note $\alpha_\ell + \beta_\ell \geq 3/4$ for $\ell \in \{1, 2\}$, and for $\ell \geq 3$, $\alpha_\ell + \beta_\ell \geq 3/4$ (see figure).
Analysis of Hybrid Algorithm

**Theorem**

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

**Proof:**

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z^*_i\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z^*_i\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z^*_i\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z^*_i + \frac{1}{2} \cdot \beta_\ell \cdot z^*_i\).
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\),
Analysis of Hybrid Algorithm

Theorem

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z^*_i\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z^*_i\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z^*_i\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z^*_i + \frac{1}{2} \cdot \beta_\ell \cdot z^*_i\).
  
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4\) (see figure)
HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z_i^*\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z_i^*\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*\).
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4\) (see figure)
Analysis of Hybrid Algorithm

Theorem

HYBRID-MAX-CNF(ϕ, n, m) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause $i$ is satisfied with probability at least $3/4 \cdot z_i^*$
- For any clause $i$ of length $\ell$:
  - Algorithm 1 satisfies it with probability $1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*$.
  - Algorithm 2 satisfies it with probability $\beta_\ell \cdot z_i^*$.
  - HYBRID-MAX-CNF(ϕ, n, m) satisfies it with probability $\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*$.
- Note $\frac{\alpha_\ell + \beta_\ell}{2} = 3/4$ for $\ell \in \{1, 2\}$, and for $\ell \geq 3$, $\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4$ (see figure)
**Analysis of Hybrid Algorithm**

**Theorem**

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

**Proof:**

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z_i^*\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z_i^*\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*\).
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4\) (see figure)
Theorem

HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z_i^*\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z_i^*\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*\).
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4\) (see figure)
HYBRID-MAX-CNF(\(\varphi, n, m\)) is a randomised 4/3-approx. algorithm.

Proof:

- It suffices to prove that clause \(i\) is satisfied with probability at least \(3/4 \cdot z_i^*\).
- For any clause \(i\) of length \(\ell\):
  - Algorithm 1 satisfies it with probability \(1 - 2^{-\ell} = \alpha_\ell \geq \alpha_\ell \cdot z_i^*\).
  - Algorithm 2 satisfies it with probability \(\beta_\ell \cdot z_i^*\).
  - HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with probability \(\frac{1}{2} \cdot \alpha_\ell \cdot z_i^* + \frac{1}{2} \cdot \beta_\ell \cdot z_i^*\).
- Note \(\frac{\alpha_\ell + \beta_\ell}{2} = 3/4\) for \(\ell \in \{1, 2\}\), and for \(\ell \geq 3\), \(\frac{\alpha_\ell + \beta_\ell}{2} \geq 3/4\) (see figure)
- \(\Rightarrow\) HYBRID-MAX-CNF(\(\varphi, n, m\)) satisfies it with prob. at least \(3/4 \cdot z_i^*\)

![Graph showing the probability of clause satisfaction over \(\ell\)]
Since $\alpha^2 = \beta^2 = 3/4$, we cannot achieve a better approximation ratio than $4/3$ by combining Algorithm 1 & 2 in a different way.

The $4/3$-approximation algorithm can be easily derandomised:

- Idea: use the conditional expectation trick for both Algorithm 1 & 2 and output the better solution.

The $4/3$-approximation algorithm applies unchanged to a weighted version of MAX-CNF, where each clause has a non-negative weight.

Even MAX-2-CNF (every clause has length 2) is NP-hard!
Exercise (easy): Consider any minimisation problem, where $x$ is the optimal cost of the LP relaxation, $y$ is the optimal cost of the IP and $z$ is the solution obtained by rounding up the LP solution. Which of the following statements are true?

1. $x \leq y \leq z$,
2. $y \leq x \leq z$,
3. $y \leq z \leq x$. 
Exercise (trickier): Consider a version of the SET-COVER problem, where each element \( x \in X \) has to be covered by at least two subsets. Design and analyse an efficient approximation algorithm. 

Hint: You may use the result that if \( X_1, X_2, \ldots, X_n \) are independent Bernoulli random variables with \( X := \sum_{i=1}^{n} X_i, \mathbb{E}[X] \geq 2 \), then

\[
\Pr[X \geq 2] \geq \frac{1}{4} \cdot (1 - e^{-1}).
\]
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover

MAX-CNF

Conclusion
Spectrum of Approximations

KNAPSACK
SUBSET-SUM
SCHEDULING,
EUCLIDEAN-
TSP
VERTEX-COVER,
MAX-3-CNF , MAX-CUT
METRIC-TSP
SET-COVER
MAX-CLIQUE
FPTAS
PTAS APX log-APX poly-APX

VI. Randomisation and Rounding Conclusion 38
Spectrum of Approximations

- KNAPSACK
- SUBSET-SUM
- SCHEDULING,
- EUCLIDEAN-
  TSP
- VERTEX-COVER,
- MAX-3-CNF , MAX-CUT
- METRIC-TSP
- SET-COVER
- MAX-CLIQUE

- FPTAS
- PTAS
- APX
- log-APX
- poly-APX

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

KNAPSACK
SUBSET-SUM
SCHEDULING,
EUCLIDEAN-
TSP
VERTEX-COVER,
MAX-3-CNF , MAX-CUT
METRIC-TSP
SET-COVER
MAX-CLIQUE
FPTAS PTAS
APX log-APX poly-APX

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

- KNAPSACK
- SUBSET-SUM
- SCHEDULING
- EUCLIDEAN-TSP
- VERTEX-COVER
- MAX-3-CNF, MAX-CUT
- METRIC-TSP
- SET-COVER
- MAX-CLIQUE

- FPTAS
- PTAS
- APX
- log-APX
- poly-APX

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

KNAPSACK, SUBSET-SUM, SCHEDULING, EUCLIDEAN-TSP, VERTEX-COVER, MAX-3-CNF, MAX-CUT, METRIC-TSP, SET-COVER, MAX-CLIQUE

FPTAS, PTAS, APX, log-APX, poly-APX

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

- KNAPSACK
- SUBSET-SUM
- SCHEDULING
- EUCLIDEAN-TSP
- VERTEX-COVER, MAX-3-CNF, MAX-CUT
- METRIC-TSP
- SET-COVER
- MAX-CLIQUE
- FPTAS
- PTAS
- APX
- log-APX
- poly-APX

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

- KNAPSACK
- SUBSET-SUM
- SCHEDULING,
- EUCLIDEAN-
  TSP
- VERTEX-COVER,
  MAX-3-CNF, MAX-CUT
  METRIC-TSP
- SCHEDULING,
  EUCLIDEAN-
  TSP
- KNAPSACK
  SUBSET-SUM

FPTAS PTAS APX log-APX
Spectrum of Approximations

VI. Randomisation and Rounding

Conclusion
Spectrum of Approximations

KNAPSACK
SUBSET-SUM
SCHEDULING,
EUCLIDEAN-TSP
VERTEX-COVER,
MAX-3-CNF, MAX-CUT
METRIC-TSP
SET-COVER
VERTEX-COVER,
MAX-3-CNF, MAX-CUT
METRIC-TSP
SCHEDULING,
EUCLIDEAN-TSP
KNAPSACK
SUBSET-SUM
FPTAS PTAS APX log-APX poly-APX
Topics Covered

I. Sorting and Counting Networks
   - 0/1-Sorting Principle, Bitonic Sorting, Batcher’s Sorting Network
     Bonus Material: A Glimpse at the AKS network
   - Balancing Networks, Counting Network Construction, Counting vs. Sorting

II. Linear Programming
   - Geometry of Linear Programs, Applications of Linear Programming
   - Simplex Algorithm, Finding a Feasible Initial Solution
   - Fundamental Theorem of Linear Programming

III. Approximation Algorithms: Covering Problems
   - Intro to Approximation Algorithms, Definition of PTAS and FPTAS
   - (Unweighted) Vertex-Cover: 2-approx. based on Greedy
   - (Unweighted) Set-Cover: $O(\log n)$-approx. based on Greedy

IV. Approximation Algorithms via Exact Algorithms
   - Subset-Sum: FPTAS based on Trimming and Dynamic Programming
   - Scheduling: 2-approx. based on Simple Greedy, 4/3-approx. using LPT
     Bonus Material: A PTAS for Machine Scheduling based on Rounding and Dynamic Programming

V. The Travelling Salesman Problem
   - Inapproximability of the General TSP problem
   - Metric TSP: 2-approx. based on MST, 3/2-approx. based on MST + matching

VI. Approximation Algorithms: Rounding and Randomisation
   - MAX3-CNF: 8/7-approx. based on Guessing, Derandomisation with Greedy
   - (Weighted) Vertex-Cover: 2-approx. based on Deterministic Rounding
   - (Weighted) Set-Cover: $O(\log n)$-approx. based on Randomised Rounding
   - MAX-CNF: 4/3-approx. based on Guessing + Randomised Rounding
Thank you and Best Wishes for the Exam!