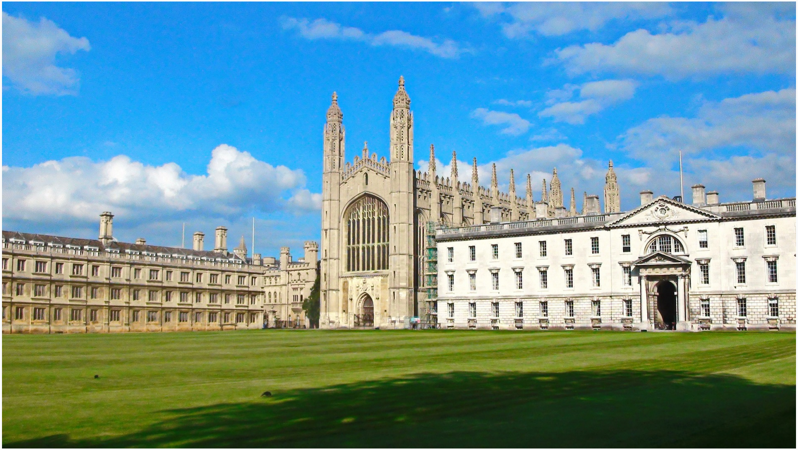
Complexity Theory

Lecture 1: Introduction and motivation

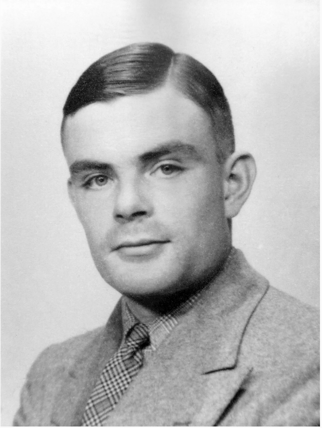
Tom Gur

http://www.cl.cam.ac.uk/teaching/2324/Complexity

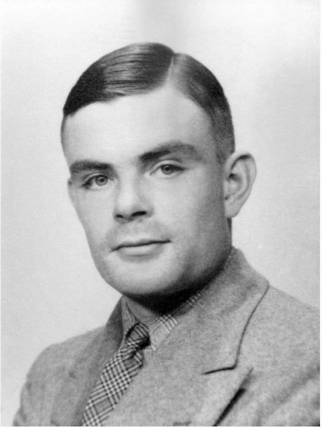
The story starts here in Cambridge...



Alan Turing and Computation Theory

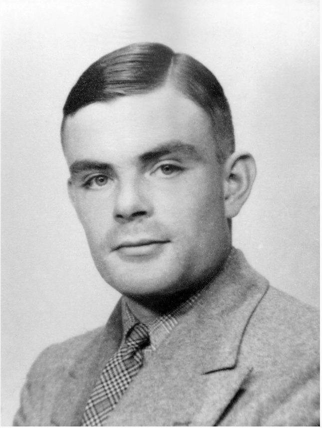


Alan Turing and Computation Theory



Infinite, or finite, that is the question.

Alan Turing and Computation Theory



Infinite, or finite, that is the question.

Or is it...

Numbers and Scale

10 - Fingers.

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

1 Trillion $= 2^{40}$ - 1T seconds = 34,842.1 years, FLOPS of PlayStation

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

1 Trillion $= 2^{40}$ - 1T seconds = 34,842.1 years, FLOPS of PlayStation

2^{50} - 35,678,377.2 years, FLOPs of a supercomputer

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

1 Trillion $= 2^{40}$ - 1T seconds = 34,842.1 years, FLOPS of PlayStation

2^{50} - 35,678,377.2 years, FLOPs of a supercomputer

2^{60} - 36 Billion years, 3 times the age of the universe

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

1 Trillion $= 2^{40}$ - 1T seconds = 34,842.1 years, FLOPS of PlayStation

2^{50} - 35,678,377.2 years, FLOPs of a supercomputer

2^{60} - 36 Billion years, 3 times the age of the universe

2^{150} - all super computers working throughout the universe's lifespan...

Numbers and Scale

10 - Fingers.

100 - a full lecture theatre

$1000 = 2^{10}$ - a rock concert

$1000000 = 2^{20}$ - 5x population of Cambridge, 1M seconds = 11.5 days

1 Billion $= 2^{30}$ - 1B seconds = 31.5 years, 1GHz = 1/billion sec, iPhone

1 Trillion $= 2^{40}$ - 1T seconds = 34,842.1 years, FLOPS of PlayStation

2^{50} - 35,678,377.2 years, FLOPs of a supercomputer

2^{60} - 36 Billion years, 3 times the age of the universe

2^{150} - all super computers working throughout the universe's lifespan...

$2^{1000000}$ complexity of an exponential-time algorithm on a small input...

What is Complexity Theory?

Complexity theory characterises tractable computation!

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

Why should you care?

Practice: learn how to avoid intractable problems.

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

Why should you care?

Practice: learn how to avoid intractable problems.

Theory: new ways of overcoming barriers (interactive proofs, sublinear algorithms, learning algorithms, etc.)

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

Why should you care?

Practice: learn how to avoid intractable problems.

Theory: new ways of overcoming barriers (interactive proofs, sublinear algorithms, learning algorithms, etc.)

Interdisciplinary: Deep connections to physics (e.g., quantum) and mathematics (e.g., P vs NP, algebraic geometry, harmonic analysis).

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

Why should you care?

Practice: learn how to avoid intractable problems.

Theory: new ways of overcoming barriers (interactive proofs, sublinear algorithms, learning algorithms, etc.)

Interdisciplinary: Deep connections to physics (e.g., quantum) and mathematics (e.g., P vs NP, algebraic geometry, harmonic analysis).

Bonus: Ways to make progress on big philosophical questions: randomness, quantum, free will, and beyond.

What is Complexity Theory?

Complexity theory characterises tractable computation!

It allows us to understand the power and limitations of algorithms.

Why should you care?

Practice: learn how to avoid intractable problems.

Theory: new ways of overcoming barriers (interactive proofs, sublinear algorithms, learning algorithms, etc.)

Interdisciplinary: Deep connections to physics (e.g., quantum) and mathematics (e.g., P vs NP, algebraic geometry, harmonic analysis).

Bonus: Ways to make progress on big philosophical questions: randomness, quantum, free will, and beyond.

So let's start!

Algorithms and Problems

Insertion Sort runs in time $O(n^2)$, while Merge Sort is an $O(n \log n)$ algorithm.

Algorithms and Problems

> *Insertion Sort runs in time $O(n^2)$, while Merge Sort is an $O(n \log n)$ algorithm.*

The first half of this statement is short for:

> *If we count the number of steps performed by the Insertion Sort algorithm on an input of size n, taking the largest such number, from among all inputs of that size, then the function of n so defined is eventually bounded by a constant multiple of n^2.*

Algorithms and Problems

> *Insertion Sort runs in time $O(n^2)$, while Merge Sort is an $O(n \log n)$ algorithm.*

The first half of this statement is short for:

> *If we count the number of steps performed by the Insertion Sort algorithm on an input of size n, taking the largest such number, from among all inputs of that size, then the function of n so defined is eventually bounded by a constant multiple of n^2.*

It makes sense to compare the two algorithms because they seek to solve the same problem.

Algorithms and Problems

> *Insertion Sort runs in time $O(n^2)$, while Merge Sort is an $O(n \log n)$ algorithm.*

The first half of this statement is short for:

> *If we count the number of steps performed by the Insertion Sort algorithm on an input of size n, taking the largest such number, from among all inputs of that size, then the function of n so defined is eventually bounded by a constant multiple of n^2.*

It makes sense to compare the two algorithms because they seek to solve the same problem.

But, what is the complexity of the sorting problem?

Review

The complexity of an algorithm (whether measuring number of steps, or amount of memory) is usually described asymptotically:

Review

The complexity of an algorithm (whether measuring number of steps, or amount of memory) is usually described asymptotically:

Definition

For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \leq cg(n)$;
- $f = \Omega(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \geq cg(n)$.
- $f = \theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Review

The complexity of an algorithm (whether measuring number of steps, or amount of memory) is usually described asymptotically:

Definition

For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \leq cg(n)$;
- $f = \Omega(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \geq cg(n)$.
- $f = \theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Usually, O is used for upper bounds and Ω for lower bounds.

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

The complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem.

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

The complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem.

To establish a *lower bound*, we need to show that no possible algorithm, including those as yet undreamed of, can do better.

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

The complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem.

To establish a *lower bound*, we need to show that no possible algorithm, including those as yet undreamed of, can do better.

In the case of sorting, we can establish a lower bound of $\Omega(n \log n)$, showing that Merge Sort is asymptotically optimal.

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

The complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem.

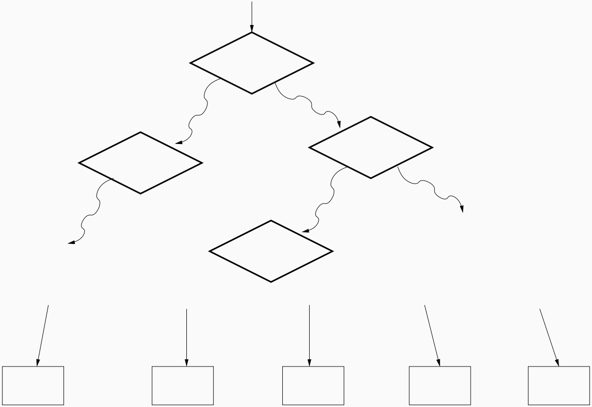
To establish a *lower bound*, we need to show that no possible algorithm, including those as yet undreamed of, can do better.

In the case of sorting, we can establish a lower bound of $\Omega(n \log n)$, showing that Merge Sort is asymptotically optimal.

Sorting is a rare example where known upper and lower bounds match.

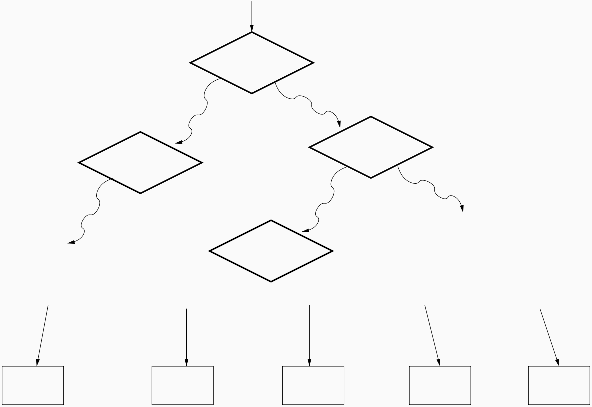
Lower Bound on Sorting

An algorithm A sorting a list of n distinct numbers $a_1, \ldots, a_n$.



Lower Bound on Sorting

An algorithm A sorting a list of n distinct numbers $a_1, \ldots, a_n$.



To work for all permutations of the input list, the tree must have at least $n!$ leaves and therefore height at least $\log_2(n!) = \theta(n \log n)$.

Travelling Salesman

Given

- V — a set of nodes.
- $c : V \times V \to \mathbb{N}$ — a cost matrix.

Travelling Salesman

Given

- V — a set of nodes.
- $c : V \times V \to \mathbb{N}$ — a cost matrix.

Find an ordering $v_1, \ldots, v_n$ of V for which the total cost:

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

is the smallest possible.

Complexity of TSP

Complexity of TSP

Obvious algorithm: Try all possible orderings of V and find the one with lowest cost.
The worst case running time is $\theta(n!)$.

Complexity of TSP

Obvious algorithm: Try all possible orderings of V and find the one with lowest cost.
The worst case running time is $\theta(n!)$.

Lower bound: An analysis like that for sorting shows a lower bound of $\Omega(n \log n)$.

Complexity of TSP

Obvious algorithm: Try all possible orderings of V and find the one with lowest cost.
The worst case running time is $\theta(n!)$.

Lower bound: An analysis like that for sorting shows a lower bound of $\Omega(n \log n)$.

Upper bound: The currently fastest known algorithm has a running time of $O(n^2 2^n)$.

Complexity of TSP

Obvious algorithm: Try all possible orderings of V and find the one with lowest cost.
The worst case running time is $\theta(n!)$.

Lower bound: An analysis like that for sorting shows a lower bound of $\Omega(n \log n)$.

Upper bound: The currently fastest known algorithm has a running time of $O(n^2 2^n)$.

Between these two is the chasm of our ignorance.

Textbooks

The main texts for the course are:

Computational Complexity.
Christos H. Papadimitriou.

Introduction to the Theory of Computation.
Michael Sipser.

Outline

A rough lecture-by-lecture guide, with relevant sections from the text by Papadimitriou (or Sipser, where marked with an S).

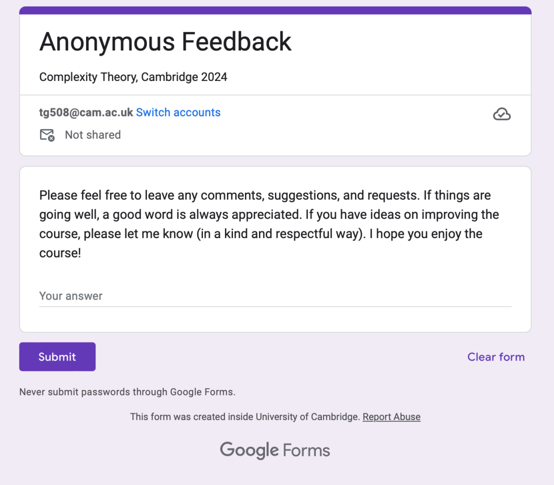
- **Algorithms and problems.** 1.1–1.3.
- **Time and space.** 2.1–2.5, 2.7.
- **Time Complexity classes.** 7.1, S7.2.
- **Nondeterminism.** 2.7, 9.1, S7.3.
- **NP-completeness.** 8.1–8.2, 9.2.
- **Graph-theoretic problems.** 9.3

Outline

- **Sets, numbers and scheduling.** 9.4
- **coNP.** 10.1–10.2.
- **Cryptographic complexity.** 12.1–12.2.
- **Space Complexity** 7.1, 7.3, S8.1.
- **Hierarchy** 7.2, S9.1.
- **Quantum Complexity** 20 [Arora-Barak]

Anonymous feedback

Let me know what works and what doesn't. Complexity theory is beautiful – let's enjoy and get the most out of it!



Anonymous Feedback

Complexity Theory, Cambridge 2024

tg508@cam.ac.uk Switch accounts

Not shared

Please feel free to leave any comments, suggestions, and requests. If things are going well, a good word is always appreciated. If you have ideas on improving the course, please let me know (in a kind and respectful way). I hope you enjoy the course!

Your answer

Never submit passwords through Google Forms.

This form was created inside University of Cambridge. Report Abuse

Questions?