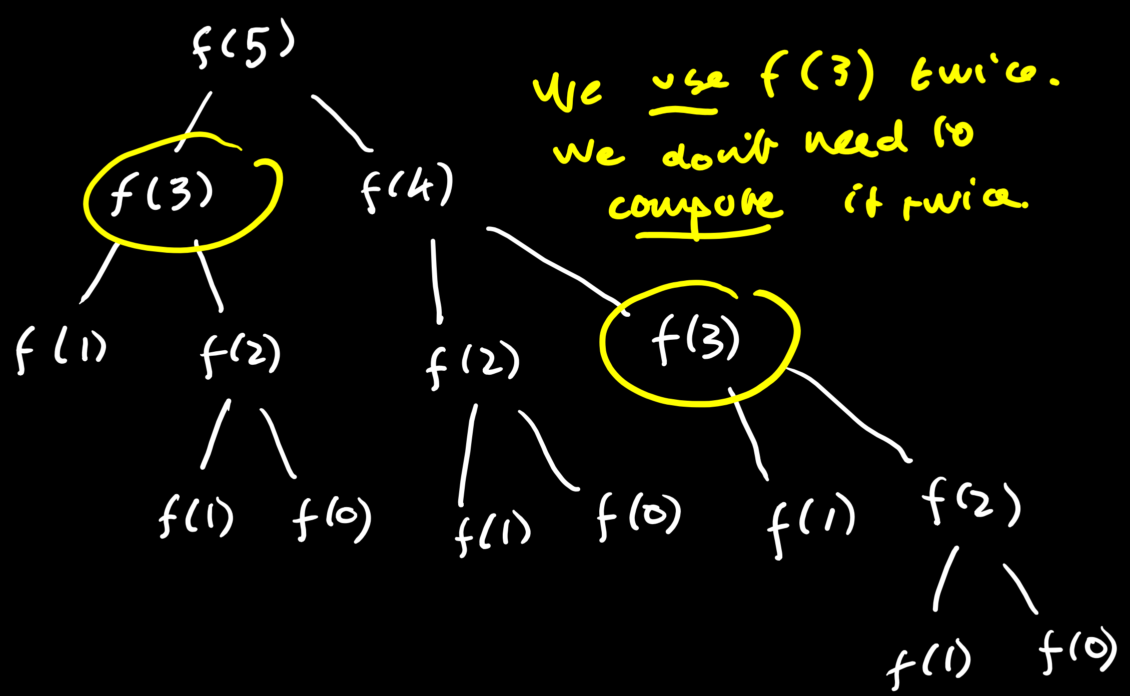
How (not) to compute the Fibonacci numbers

$$F_0 = F_1 = 1$$
$$F_n = F_{n-2} + F_{n-1} \text{ for } n \geq 2$$

```python
def f(n):
  return 1 if n<2 else f(n-2) + f(n-1)
```

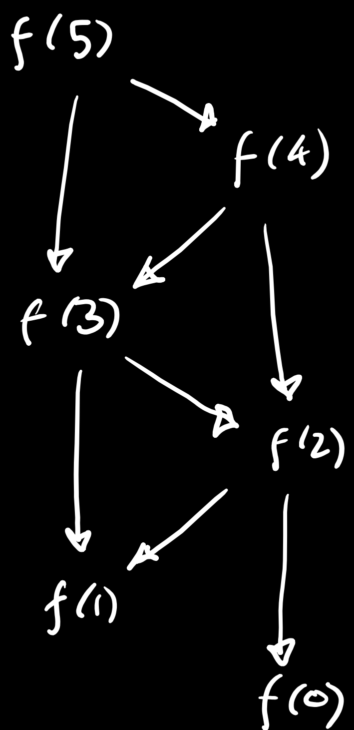
QUESTION
Why is this a daft implementation?

Tree of function calls



We use f(3) twice. We don't need to compute it twice.

Dependency graph



How ~~(not)~~ to compute the Fibonacci numbers

$$F_0 = F_1 = 1$$
$$F_n = F_{n-2} + F_{n-1} \text{ for } n \geq 2$$

We can get $\Theta(n)$ running time by leveraging duplication in the dependency graph.

```
def f(n):
  x = np.ones(n+1)
  for i in range(2,n+1):
    x[i] = x[i-2] + x[i-1]
  return x[n]
```

Brute-force bottom-up strategy
time $\Theta(n)$, space $\Theta(n)$
needs us to think about the dependency graph,

```
cache = {}
def f(n):
  if n in cache:
    return cache[n]
  else:
    res = 1 if n<2 else f(n-2)+f(n-1)
    cache[n] = res
    return res
```

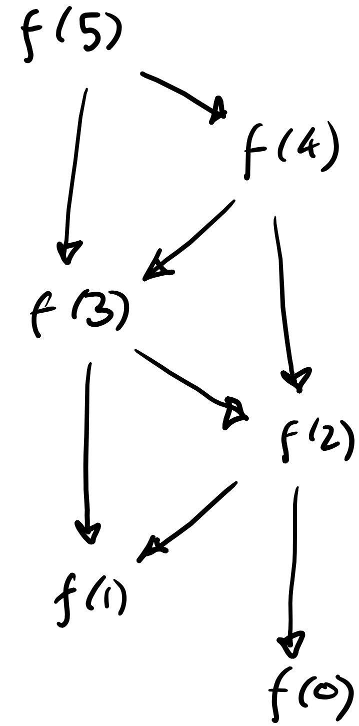
Top-down with memo-ization
time $\Theta(n)$, space $\Theta(n)$
totally generic approach, no thought required

Both these strategies are good ways to solve Dynamic Programming recursions

```
def f(n):
  x,y = 1,1
  for _ in range(2,n+1):
    x,y = y, x+y
  return y
```

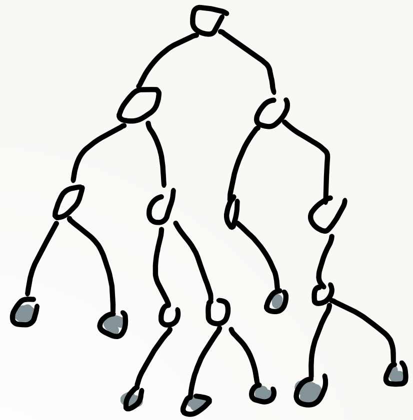
Elegant bottom-up strategy
time $\Theta(n)$, space $O(1)$.
This is a bit too special-case for our needs

Dependency graph



3.1 Practical dynamic programming

The naive recursive solution to the Bellman equation is often impractical, since the computation tree typically grows exponentially with the size of the problem.

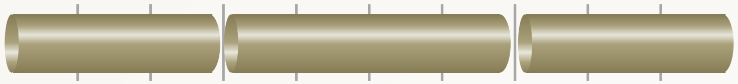


In many interesting problems, there is substantial overlap in the subproblems, permitting polynomial-time solution, using ...

- **top-down memo-ization**
  Simply implement the recursion, and cache the results
- **or bottom-up iteration**
  Start from the leaves and work up
  (but we first need to figure out the dependency graph)

Example: rod cutting

A DIY supplier has a steel rod of length $n \in \mathbb{N}$, and a machine that can cut it into smaller pieces. Different lengths can be sold for different prices; a piece of length $\ell \in \mathbb{N}$ fetches p_ℓ.



How should it be cut, to maximize profit?

Bellman equation: Let $v(n)$ be the maximum profit achievable from a rod of length n. Then

$$v(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max_{1 \le i \le n} \{p_i + v(n-i)\} & \text{if } n > 0 \end{cases}$$

ie $v(n)$ depends on $v(n-1), v(n-2), \ldots, v(0)$

Dependency graph

$v(0) \quad v(1) \quad v(2) \quad \ldots \quad v(n)$

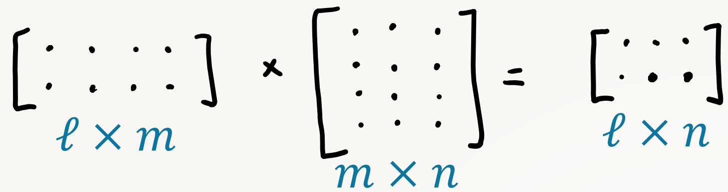
Bottom-up strategy:

1. Create an array of size $n+1$
2. Set $v(0)=0$
3. Fill in $v(1), v(2), \ldots, v(n)$ in order.

Top-down memo-ization strategy: nothing special to say here; it's a totally generic approach.

Example 3.1.1 Matrix chain multiplication

The cost of multiplying two matrices depends on their dimensions:



$\ell m n$ multiplications $+\ \ell(m-1)n$ additions
Let's take the total cost to be $\ell m n$.

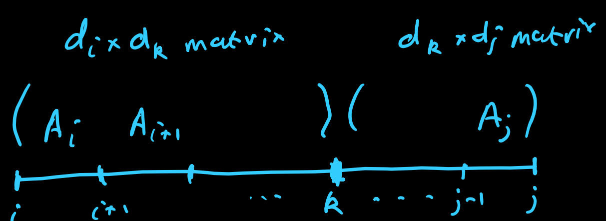
If we want to compute the product of several matrices, we have a choice about the order of multiplication (because matrix multiplication is associative). For example,

$$ABCDE = (AB)\big((CD)E\big) = A\Big(B\big((CD)E\big)\Big)$$

Find the least-cost way to compute the product $\underset{d_0 \times d_1}{A_0} \cdot \underset{d_1 \times d_2}{A_1} \cdot \cdots \cdot \underset{d_{n-1} \times d_n}{A_{n-1}}$

Bellman equation: Let $v(i,j)$ be the minimum cost for multiplying $A_i A_{i+1} \cdots A_{j-1}$, for $i < j$. Then

$$v(i,j) = \begin{cases} 0 & \text{if } j = i+1 \\ \min\limits_{i<k<j} \{d_i d_k d_j + v(i,k) + v(k,j)\} & \text{if } j > i+1 \end{cases}$$

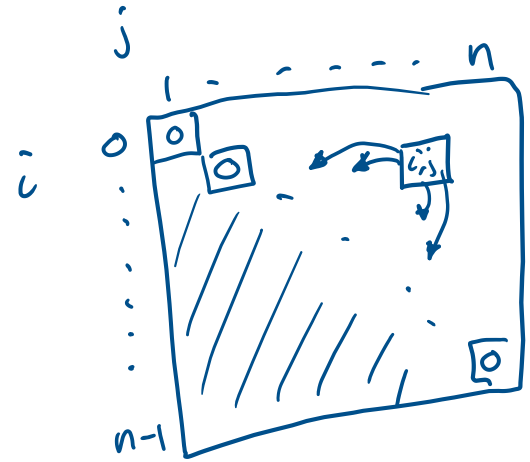


Bellman equation: Let $v(i,j)$ be the minimum cost for multiplying $A_i A_{i+1} \cdots A_{j-1}$, for $i < j$. Then

$$v(i,j) = \begin{cases} 0 & \text{if } j = i+1 \\ \min\limits_{i<k<j} \{d_i d_k d_j + v(i,k) + v(k,j)\} & \text{if } j > i+1 \end{cases}$$

ie $v(i,j)$ depends on $v(i,k)$ for k larger than i & $v(k,j)$ for k smaller than j

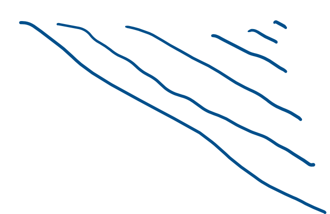
Dependency graph:



(note: ignore the $i \geq j$ part, (the part under the diagonal), since $v(i,j)$ only makes sense for $i<j$.)

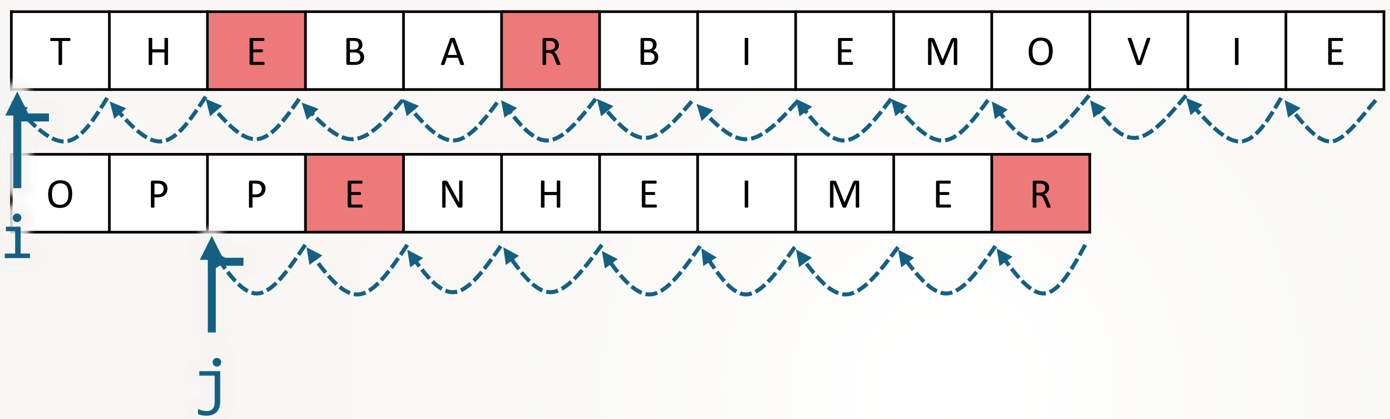
Bottom-up strategy:

1. Create a $n \times n$ matrix
2. Fill in 0 on the diagonal $(j=i+1)$
3. Fill in the $j=i+2$ diagonal
   then $j=i+3 \cdots$
   until we fill in $i=0, j=n$.



Example 3.1.2 Longest common subsequence

A *subsequence* of a string s is any string obtained by dropping zero or more characters from s. Given two strings s and t, what's the longest subsequence they have in common?



Let's frame the task as choosing a sequence from these actions:

i	decrement `i`
j	decrement `j`
m	match a character and decrement `i` & `j`

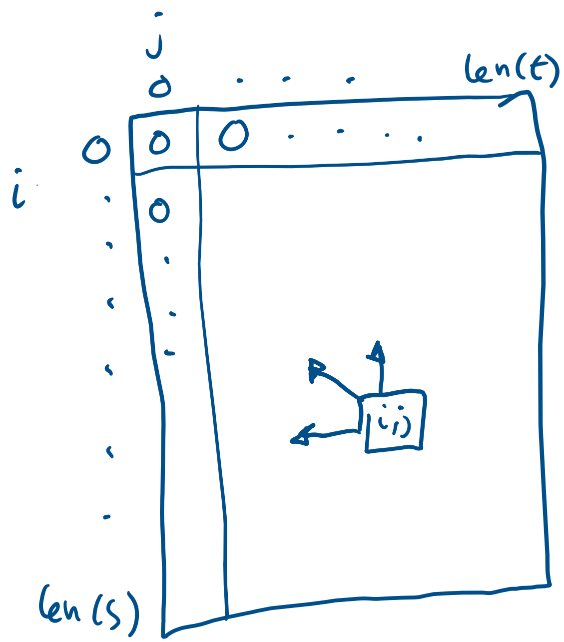
Bellman equation: Let $v_{i,j}$ be the length of the LCS between `s[0:i]` and `t[0:j]`. Then

$$v_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ v_{i-1,j} \vee v_{i,j-1} \vee \left(1 + v_{i-1,j-1}\right) & \text{if } i > 0 \text{ and } j > 0 \text{ and } \texttt{s[i-1]=t[j-1]} \\ v_{i-1,j} \vee v_{i,j-1} & \text{if } i > 0 \text{ and } j > 0 \text{ and } \texttt{s[i-1]} \neq \texttt{t[j-1]} \end{cases}$$

Bellman equation: Let $v_{i,j}$ be the length of the LCS between `s[0:i]` and `t[0:j]`. Then

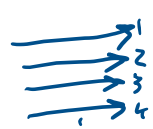
$$v_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ v_{i-1,j} \vee v_{i,j-1} \vee \left(1 + v_{i-1,j-1}\right) & \text{if } i > 0 \text{ and } j > 0 \text{ and } \texttt{s[i-1]=t[j-1]} \\ v_{i-1,j} \vee v_{i,j-1} & \text{if } i > 0 \text{ and } j > 0 \text{ and } \texttt{s[i-1]} \neq \texttt{t[j-1]} \end{cases}$$

Dependency graph

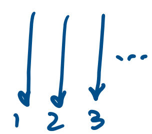


Bottom-up strategy:

1. create a (len(s)+1) × (len(t)+1) matrix
2. Fill in 0 on i=0 and on j=0
3. Fill in the rest, eg.



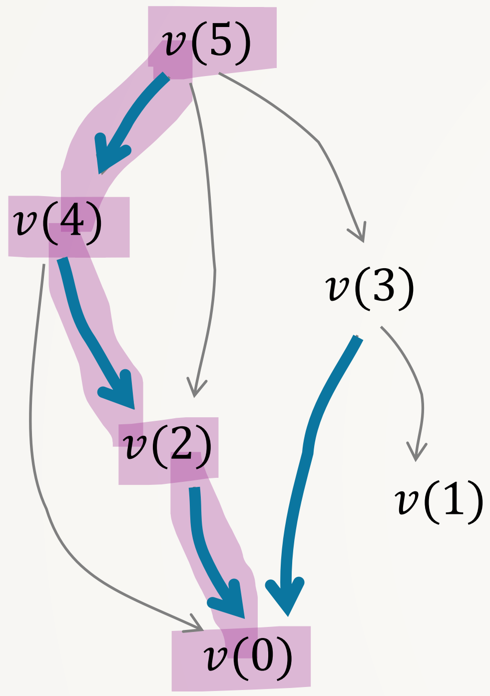
or



or

or any way that's consistent with the dependencies.

How to extract the programme



When we compute the maximum

$$v(x) = \max_a \{\text{reward}_{x,a} + v(\text{next}_{x,a})\}$$

let's also store which a achieved the maximum.

To find an optimal path, just start at the top and repeatedly pick the best action.

This works whether we're computing the values bottom-up, or top-down with memo-ization.

QUESTION
What would you do if there are two equally-good actions?

Example 3.1.2 Longest common subsequence

We produce a table of $v_{i,j}$ = length of longest common subsequence between `s[0:i]` and `t[0:j]`

At the same time, we store the optimal action at each state (i, j)

To extract the match, start at the initial state $(i, j) = $ (`len(s)`, `len(t)`), then follow the optimal actions.

A longest common substring of ALGORITHM and LOGARITHM is LGRITHM

t = "logarithm"

s = "algorithm"

i \ j		L	O	G	A	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
A	0	↑	↑	↑	↖	←	←	←	←	←
L	0	↖	←	←	↑	↑	↑	↑	↑	↑
G	0	↑	↑	↖	←	←	←	←	←	←
O	0	↑	↖	↑	↑	↑	↑	↑	↑	↑
R	0	↑	↑	↑	↑	↖	←	←	←	←
I	0	↑	↑	↑	↑	↑	↖	←	←	←
T	0	↑	↑	↑	↑	↑	↑	↖	←	←
H	0	↑	↑	↑	↑	↑	↑	↑	↖	←
M	0	↑	↑	↑	↑	↑	↑	↑	↑	↖

OPTIMAL ACTION

- ↑ decrement i
- ← decrement j
- ↖ match a character and decrement i & j

The art of dynamic programming is to formulate the problem so that we maximize overlap between subproblems.

Example. Find the least-cost way to compute the matrix product $A_0 A_1 \cdots A_{n-1}$

Recall that matrix multiplication is associative: $ABCDE = (AB)\big((CD)E\big) = A\Big(B\big((CD)E\big)\Big)$

Let's think of the problem as "repeatedly, choose a pair of adjacent matrices to multiply".

$$\begin{array}{c} d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_4 \quad d_5 \\ A\,B\,\underbrace{C\,D}\,E \end{array}$$

$$\begin{array}{c} d_0 \quad d_1 \quad d_2 \quad d_4 \quad d_5 \\ \underbrace{A\,B}\,M_1\,E \end{array}$$

$$\begin{array}{c} d_0 \quad d_2 \quad d_4 \quad d_5 \\ M_2\,\underbrace{M_1\,E} \end{array}$$

$$\begin{array}{c} d_0 \quad d_2 \quad d_5 \\ \underbrace{M_2\,M_3} \end{array}$$

$$\begin{array}{c} d_0 \quad d_5 \\ M_4 \end{array}$$

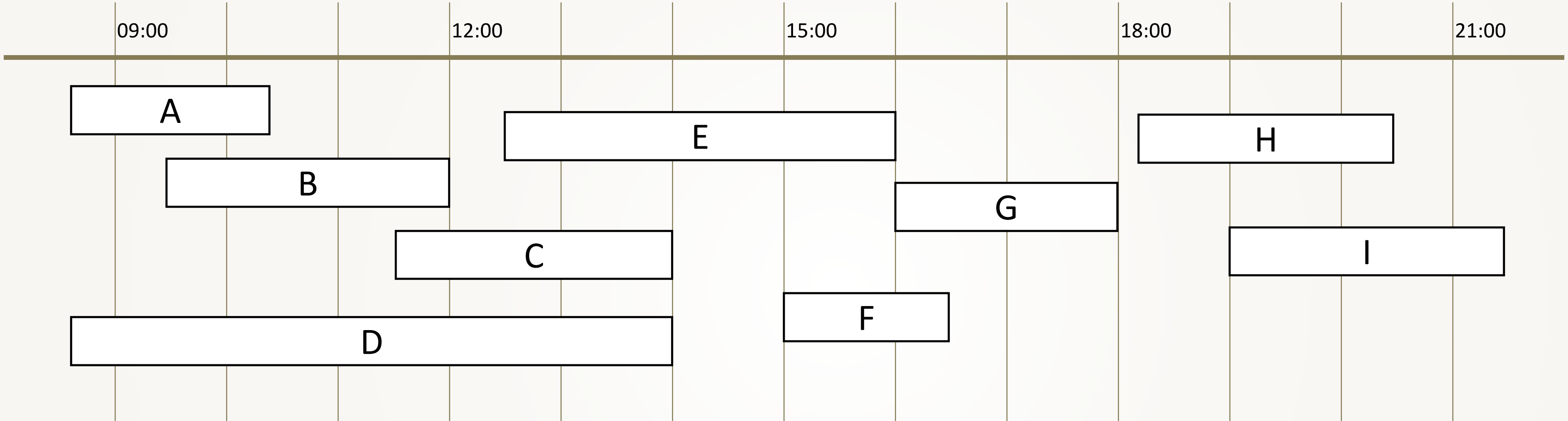
Let $v(\underline{e})$ be the minimum cost of multiplying matrices with dimension-sequence $\underline{e} = [e_0, e_1, \ldots, e_n]$. Then

$$v(\underline{e}) = \begin{cases} 0 & \text{if } n=1 \\ \min\limits_{0<k<n} \left\{ d_{k-1} d_k d_{k+1} + v\left(\underline{e} \text{ with item } k \text{ dropped}\right) \right\} \end{cases}$$

This is yucky because the dependency graph has lots of nodes (one node for every possible $\underline{e}$ for a given list of matrices).
For our other approach, #nodes is quadratic in n.

Example 3.2.1 Resource allocation

Several different university societies have all requested to book the sports hall, request k having start time $u_k \in \mathbb{R}$ and end time $v_k \in \mathbb{R}$. The hall can only fit one activity at a time. What is the maximum number of requests that can be satisfied without overlap?



$f(X)$ = max # requests that can be simultaneously satisfied from a set X.

$$f(\emptyset) = 0$$

$$f(X) = \max_{k \in X} \left\{ 1 + f\left(\{\ell \in X : v_\ell \le u_k\}\right) + f\left(\{\ell \in X : u_\ell \ge v_k\}\right) \right\}$$

events in X that finish before k starts

events in X that start after k ends

EXERCISE
Find a different formulation, not based on sets, so that the subproblems overlap better.