Median

Already in quicksort we saw the need for median.
In statistics, median is used a lot as a more robust notion of mean.
Similarly, used as a way to get robust decisions from sensors.

Input: A set A of n distinct (for simplicity) numbers, and an integer $i \in \{1,..,n\}$
Output: The ith smallest element.

The cases $i=1$ (minimum) and $i=n$ (maximum) are easy to compute in $O(n)$ time.

But what the median? Let's use an idea like quicksort, except recursively call only on one side.

```
RADOMIZED SELECT (A, i)
1. Choose j randomly from {1,..,n}
2. k = PARTITION (A, j)
3. IF k=i
4.     RETURN A[k]
5. ELSIF k>i
6.     RETURN RANDOMIZED SELECT( A[1...k-1], i)
7. ELSE (k<i)
8.     RETURN RADOMIZED SELECT( A[k+1,..,n], i-k)
```

Correctness ✓ (with certainty)

Rough runtime analysis: we can expect the pivot j to be "approx-median", i.e., not in the bottom or top 30%. This happens w.p. 40%.

So the runtime is roughly:

$$T(n) \leq T(\tfrac{7n}{10}) + O(n)$$

which gives $T(n) = O(n)$.

But what if we want a deterministic algorithm? (Indeed, for quicksort, we already know how to choose pivot randomly)

Idea: replace Step 1 with another "approx-median" procedure.

First attempt: Compute recursively median of $A[1..\frac{3n}{5}]$.

This median is guaranteed to be an "approx-median", since it has $\frac{3n}{10}$ elements smaller than it, and $\frac{3n}{10}$ elements larger.

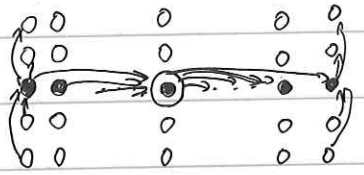
So running time is:

$$T(n) \geq T(\tfrac{3n}{5}) + T(\tfrac{7n}{10}) + O(n)$$

which gives $T(n) = \Omega(n^{1.51})$

Actual algorithm: "median-of-medians".

1. Partition A into $\frac{n}{5}$ sets of size 5 each.
2. Compute median of each set of size 5.
3. Compute median of those $\frac{n}{5}$ medians, and output that element.



The element we output is guaranteed to be bigger than $\frac{3n}{10}$ elements and also smaller than $\frac{3n}{10}$ elements. Therefore, it's an "approx-median".

The runtime is now:

$$T(n) = T(\tfrac{n}{5}) + T(\tfrac{7n}{10}) + O(n)$$

which gives $T(n) = O(n)$ as desired.

Q What happens if we replace 5 by 7? by 3?

Lower bounds on comparison queries

All the sorting/selecting algorithms we have seen so far are comparison based: they don't care about the actual numbers in the array, only their relative order.

- Comparison model: input items are "black boxes" and only allowed comparison ($<, \leq, \geq, >$)
- Time cost $\Rightarrow$ # comparisons

Take, for example, the maximum finding algorithm:

```
1. cur_max = bigger(1,2)
2. FOR i=3 to n
3.     cur_max = bigger(i, cur_max)
4. RETURN cur_max
```

Total number of comparisons: $n-1$.

Can we compute maximum with fewer comparisons? x, y, z ≥ 2 comparisons

x, y, z, w ≥ 3 comparisons

Claim Computing maximum of n elements requires $\geq n-1$ comparisons.

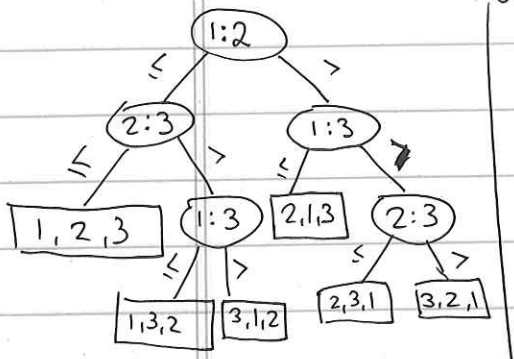
Proof There can be at most one element that has never lost a comparison. Otherwise, each of the two elements can potentially be the maximum and the algorithm has no way of telling (because they won all competitions so far).

Therefore, $n-1$ elements must lose at least one competition. But since there is one loser in each comparison, we need $n-1$ comparisons.

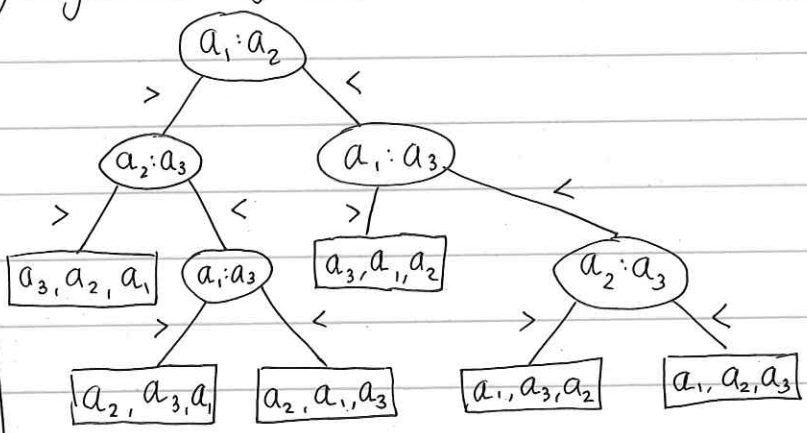
(Alternative proof: consider behavior of algorithm on $1, \ldots, n$; we claim that each of $1, \ldots, n-1$ must be compared at least once to a bigger number, i.e. "lose" a comparison. Otherwise we can replace that number with $n+1$ and the algorithm would never notice, and still output n as the biggest)

What about sorting?

We can model any comparison-based algorithm (mergesort, insertion sort, ...) as a decision tree:



Insertion sort



Notice that we have 6 leaves corresponding to all $3! = 6$ permutations of the input. In general, when sorting n elements, any correct algorithm must have at least $n!$ leaves, since all permutations are possible.

What is the number of comparisons performed on the worst case input? It is exactly the depth of the tree.

Since a depth k tree has $\leq 2^k$ leaves, we get

$$n! \leq 2^k \Rightarrow k \geq \log_2 n! = \Omega(n \log n). \qquad \text{(e.g., since } n! \geq \left(\tfrac{n}{2}\right)^{n/2}\text{)}$$

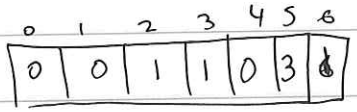
Therefore, any algorithm for sorting n elements using comparisons must use at least $\Omega(n \log n)$, and in particular run in time $\Omega(n \log n)$ in the worst case.

Corollary - MergeSort, QuickSort (with median pivot), HeapSort, are optimal up to constant.

Counting Sort

Best sorting algorithm is $O(n \cdot \sqrt{\log \log n})$ (randomized).
Conjectured: $O(n)$ possible.
Assume keys are in $\{0, .., k-1\}$.

3 5 5 2 5 6



2 3 5 5 5 6

Algorithm works, but we need to preserve the data associated with each key, not just sort the keys themselves.

```
L = array of k empty lists                    } O(k)
for j in range(n):                            }
    L[key(A[j])].append(A[j])                 } O(1)   } O(n)
output = []
for i in range(k):                            }
    output.extend(L[i])                       } O(|L[i]|+1)  } O(n+k)
```

(in CLRS you'll find a more practical algorithm that uses 3 array and doesn't need any lists like here)

Runtime is $O(n+k)$. If $k = O(n)$, this is linear time $O(n)$.
As soon as it's a little bit bigger, you're in trouble.
Ex Show stability
This was mainly a warmup, but not ultimately what we want.

```
COUNTING-SORT (A, B, k)
 For i=0 to k
     C[i]=0
 For j=1 to n
     C[A[j]] ++
 // C[i] now #elements i
 For i=1 to k
     C[i] += C[i-1]
 // C[i] now contains #elements ≤ i
 FOR j=n downto 1
     B[C[A[j]]] = A[j]
     C[A[j]] --
```

A = 2, 5, 3, 0, 2, 3, 3

C = 1, 0, 2, 3, 0, 1

C = 1, 1, 3, 6, 6, 7 (indices 0 1 2 3 4 5)

B = _ _ _ _ 3 3 _ (indices 1 2 3 4 5 6 7)

Runtime: $\Theta(k+n)$.

Beats the $\Omega(n \log n)$ lower bound since not a comparison sort (in fact, it never uses any comparison!)

Radix Sort

Assume keys are in $\{0, \dots, k^d - 1\}$. We can think of them as d-digit numbers, with digits in $\{0, \dots, k-1\}$.

We can use any stable sort (in particular counting sort) to sort:

For $i = 1$ to d do:

Stable sort based on the i^{th} digit.

The runtime is $O(d(n+k))$. So as long as d is constant and $k = O(n)$, this is $O(n)$. This way we can sort numbers in the range $\{0, \dots, n^{10}\}$ in linear time (whereas counting sort directly would be terribly slow!)

Ex Prove correctness (where you should crucially use that the sort is stable)

DYNAMIC PROGRAMMING (DP) (Bellman, 1950s) Brooklyn College,

- General, powerful alg. design technique.
- Especially good for optimization problems
- DP ≈ "careful brute force"
- DP ≈ subproblems + reuse

Fibonacci numbers:

$F_1 = F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

goal: compute F_n (assume fits in a word)

Naive recursive alg:

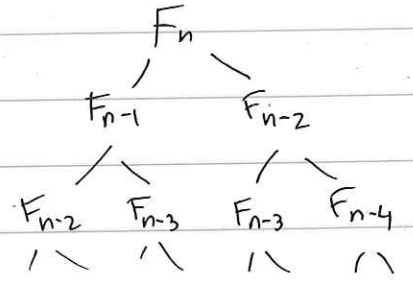
```
fib(n):
    if n≤2: f=1
    else: f = fib(n-1) + fib(n-2)
    return f
```

Correct, but exponential time:

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

$\geq F_n \approx \varphi^n \approx 1.61^n$

(alternatively, $T(n) \geq 2T(n-2)$, so $T(n) \geq 2^{n/2}$)



Memoized DP alg:

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    if n≤2: f=1
    else: f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

You can do this to any function (and Python has it built in as of version 3.2).

fib(k) only recurses the first time it's called.
memoized calls take $O(1)$.
# non-memoized calls is n:
fib(1), fib(2), .. fib(n)
Amount of work done per call is constant $O(1)$ (since recursive calls are now constant) so overall running time is linear $O(n)$.
(best algorithm uses $O(\log n)$ arithmetic operations)

DP: - memoize (remember)
& reuse solutions to subproblems

DP ≃ recursion + memoization
⇒ time = # subproblems · (time per subproblem)
↗ Not counting recursions!
No recurrences here :)!

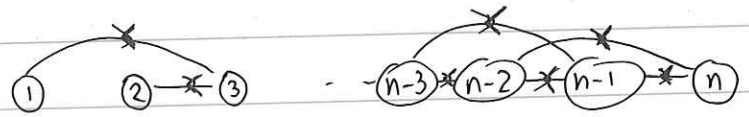
(in divide and conquer, we also had subproblems, but they were usually disjoint, and we never encountered the same subproblem more than once, so there is no gain in memoization)

Bottom-up DP algorithm:

(probably how you've seen Fibonacci computed before.
Unrolls the recursion and makes it more explicit.)

```
fib = {}
for k = 1 to n:
    [if k ≤ 2:  f = 1                          } O(1)   } O(n)
    [else:  f = fib[k-1] + fib[k-2]            }
    fib[k] = f
return fib[n]
```

Runtime obviously linear



(We can reduce space (memory) from n to constant)

Any dynamic programming alg can be converted into a bottom-up algorithm.

DP ≈ "careful brute force"

DP ≈ guess + recursion + memoization

time = #subproblems · time/subproblem (treating recursive calls as $\Theta(1)$)

5 steps for DP:

① Define subproblems (#)

② Guess part of a solution (#)

③ Recurrence (time/subprob)

④ Recurse + memoize or bottom up

time = #subproblems · time/subprob

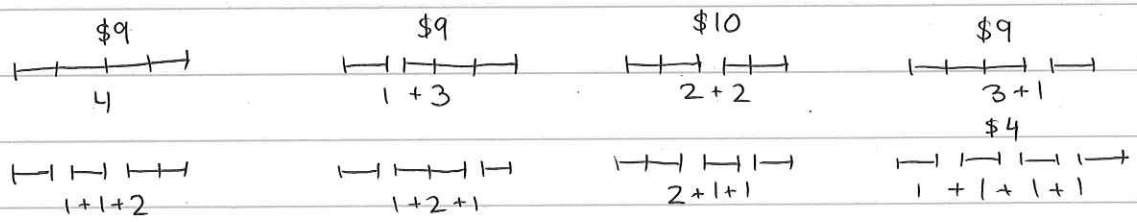
⑤ Solve original problem

Rod Cutting

What the highest revenue we can get by cutting an n feet rod and selling the pieces?

length i	1	2	3	4	5	6	7	8	9
price p_i	1	5	8	9	10	17	17	20	24

There are 2^{n-1} ways of cutting the rod, since we can either cut or not at each of the n-1 possible positions.



(in fact, enough to consider partitions up to permutation, e.g., 1+1+2 is obviously same revenue as 1+2+1; this reduces # partitions to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$ Hardy & Ramanujan '18 $= 2^{\Theta(\sqrt{n})}$, which is more than any polynomial).

length	1	2	3	4	5	6	7	8	9
revenue	1	5	8	10	13	17	18	22	25
	1=1	2=2	3=3	4=2+2	5=2+3	6=6	7=6+1 7=3+2+2	8=2+6	9=6+3

In general, we can write a recurrence that tries all possible ways:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

p_n: selling uncut; $r_1 + r_{n-1}$: cutting after 1; $r_2 + r_{n-2}$: cutting after 2

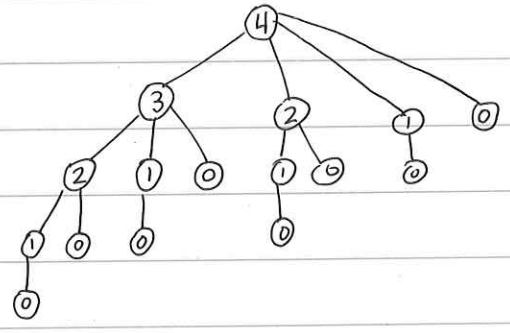
To solve the original problem of size n, we solve smaller problems of the same kind.

Alternatively, we can view a decomposition as consisting of a first piece of length i and then a remainder of length $n-i$.

$$r_n = \max(p_1 + r_{n-1},\ p_2 + r_{n-2},\ \dots,\ p_{n-1} + r_1,\ p_n + r_0)$$

(where we define $r_0 = 0$)

```
CUT ROD(n)
if n == 0:
    return 0
q = -∞                          (*)
FOR i = 1 TO n:
    q = max(q, p_i + CUTROD(n-i))
RETURN q.
```



Runtime is: $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$, giving $T(n) = 2^n$ (prove by induction!)
Not surprisingly since we are explicitly trying all possible 2^{n-1} ways to cut the rod.

We can speed it up using memoization:

```
CUTROD(n):
IF memo[n] exists:
    RETURN memo[n]
(*)
memo[n] = q
return q
```

Runtime is only $O(n^2)$ now! (Because need to compute CUTROD(1), ..., CUTROD(n) and each takes $O(n)$; recursive call are free!)

Here is a bottom-up version:

```
CUT-ROD(n)
r[0] = 0
FOR j = 1 to n
    q = -∞
    FOR i = 1 to j
        q = max(q, p_i + r[j - i])
    r[j] = q
RETURN r[n]
```

Again, running time is $\Theta(n^2)$.

This algorithm outputs the revenue. How to output the actual sequence of cuts we should do? We just need to keep track of best i in each step: "parent pointers": remember which guess was best. (automatic, no thinking required)

```
r[0] = 0
FOR j = 1 to n
    q = -∞
    FOR i = 1 to j
        IF q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
PRINT r[n]
while n > 0
    PRINT s[n]
    n = n - s[n].
```

Remark We could try all $\Theta(n^2)$ ways to guesse a piece in the middle, leading to a $\Theta(n^3)$ alg.

Remark Greedy fails here:

$p_1 = p_2 = p_3 = p_4 = 1$, $p_5 = 100$, $p_6 = 180$, $p_7 = p_8 = p_9 = p_{10}$

Then for rod length 10, greedy would start with 6 feet (since $\frac{180}{6} = \$30/\text{foot}$) but then get total value 184. Optimum is 5+5, giving \$200.

Longest Common Subsequence

Subsequences of HUMAN: UMN, HAN, UA, ...

Longest common subsequence: given two strings $x \in \Sigma^m$, $y \in \Sigma^n$, find the longest common subseq.

HYPERLINKING
DOLPHINSPEAK (PINK)
HERMOSILLO (HELLO)
MICHELANGELO
HELLENOPHOBE

THEKILLERWHALEADELE
SEA TURTLE
ERLE

This problem shows up in bioinformatics (similarity between DNA sequences) and version control (GitHub).

"optimal substructure"

Thm $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$, and Z is an LCS of X and Y, $Z = z_1 \dots z_k$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then:
   a. If $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y
   b. If $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof 1. If $z_k \neq x_m$, then we could append $x_m = y_n$ to end of Z, in contradiction.
Therefore $z_k = x_m = y_n$. Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}.
If it is not the longest, then there is another common subseq of length $\geq k$, and by appending $z_k = x_m = y_n$, we get a common subseq of X and Y of len $\geq k+1$, in contradiction

2a. Z is a common subseq of X_{m-1} and Y. If there is another common subseq of X_{m-1} & Y of length $> k$, it's also of X & Y, in contradiction.

2b. Similarly. ∎

① Subproblems: For $i = 0, \ldots, m$ and $j = 1, \ldots, n$,

Let $c[i,j]$ be length of LCS of $X_i = (x_1, \ldots x_i)$ and $Y_j = (y_1, \ldots y_j)$

② Guess: which of the last character of X and of Y is not used in the LCS? (unless both are used)

③ Recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

④

A naive recursive implementation gives an exponential time algorithm. However, since there are only $O(nm)$ subproblems, memoization or a bottom-up approach would give an $O(nm)$ time algorithm.

⑤

```
LCS (X[1..m], Y[1..n])

FOR i=1 to m:
    c[i,0] = 0
FOR j=1 to n:
    c[0,j] = 0
FOR i=1 to m:
    FOR j=1 to n:
        If Xi == Yj:
            c[i,j] = c[i-1,j-1] + 1
            b[i,j] = "↖"
        else if c[i-1,j] ≥ c[i,j-1]
            c[i,j] = c[i-1,j]
            b[i,j] = "↑"
        else
            c[i,j] = c[i,j-1]
            b[i,j] = "←"
return c[m,n]
```

Knapsack

- list of n items each of size s_i (integer) and value v_i
- knapsack of size S
- What is max total value for a subset of items of total size $\leq S$?

i	1	2	3	4
v_i	10	40	30	50
s_i	5	4	6	3

$S = 10$

① Subproblems: items $1, \ldots, i$ & remaining capacity $X \leq S$

(total of $\Theta(n \cdot S)$ subproblems)

② Guess: is item i in or not?

③ Recurrence: $DP(i, X) = \max(DP(i-1, X),\ DP(i-1, X-s_i) + v_i)$

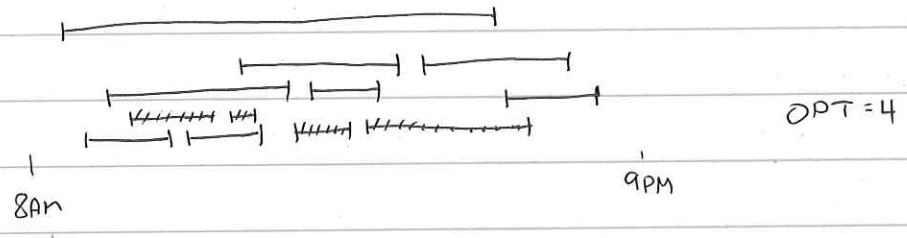
time $= \Theta(n \cdot S)$ (not really polynomial time).

input size is $\Theta(n \cdot \log S)$ "pseudopolynomial"

i \ X	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Ex Write down the algorithm and find the subset that gives optimal solution.

Activity Selection Problem / Interval Scheduling



Input: list of intervals $S = \{I_1, \ldots, I_n\}$ where for each i, $I_i = [s_i, f_i)$.

Goal: find a subset $S' \subseteq S$ of non-intersecting intervals of largest size.

First attempt: dynamic prog.

① Subproblems: for any $i < j$, the optimal solution for intervals starting after f_i and ending before s_j.

② Guess an interval used by the optimum

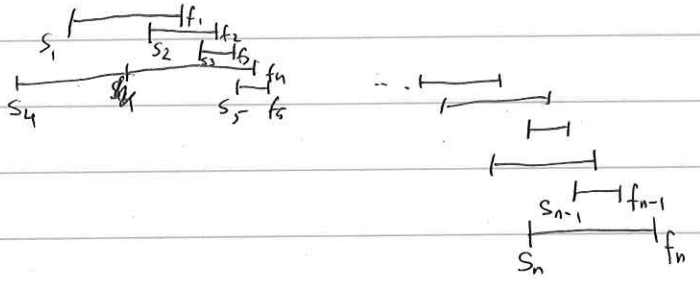
③ Recurrence:

$$DP(i,j) = \max_{k \text{ s.t. } [s_k, f_k) \subseteq [f_i, s_j)} DP(i,k) + DP(k,j) + 1$$

$O(n^2)$ subproblems. $O(n)$ time per subp $= O(n^3)$ runtime.

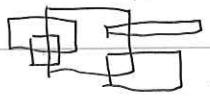
Second attempt: improved dynamic prog

Sort the activities by finish time.



X { Guess whether first interval is used or not? No! Subproblems become too complex.
For example, if we include I_1, we need to remove I_2 and I_4,
but not I_3.

Similarly for rectangle scheduling:



No way to control subproblems.

Guess whether last interval is used or not? Yes!

Subproblems: for $i = 0, \ldots, n$, optimum solution for $\{I_1, \ldots, I_i\}$.

Recurrence:

$$DP(i) = \max(DP(i-1),\ 1 + DP(k_i))$$

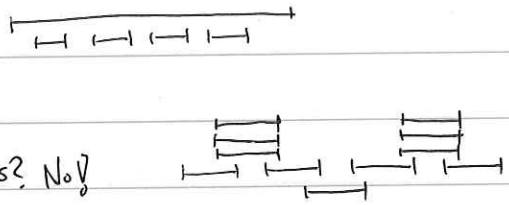
where k_i is last job to finish before s_i.

$O(n)$ subproblems $\cdot$ $O(1)$ per subproblem $= O(n)$ runtime.

(actually $O(n \log n)$ because we need to sort)

Maybe we don't need to try all possible activities?
Maybe we can immediately identify an activity that is used in an optimal solution?

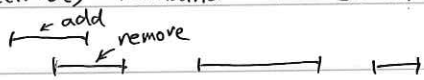
- Activity with earliest start time? NO!
- Shortest activity? No!
- Activity intersecting least number of other activities? No!



Turns out we can always take the first activity to finish!

Thm For any list of intervals S there is an optimal solution that includes the activity that finishes first.

Proof Take any optimal solution. If already includes that activity, we're done. Otherwise, add that activity. It can collide with at most one other activity, so remove that other one. The # of activities remained the same.



After sorting the input activities by finish time,
The greedy algorithm take

$$O(n) \text{ subproblems (suffixes)} \times O(1) \text{ time/sub} = O(n).$$

We can also write it directly:

```
cur_fin = -∞
For i=1 to n:
    if s_i > cur_fin:
        print i
        cur_fin = f_i
```