Algorithms

Backtracking (part 3)
& Dynamic Programming
intro.

Recap: Canvas - updated

- HW1 mess-up: groups!

  Must sign up for a group each time.

  May look like you didn't submit. Sign up for a group!

- HW2: oral grading (round 1)

  Next Tuesday & Wednesday, your group will need to find a ½ slot to sign up for with me.

  We'll meet via zoom.

  (Review HW FAQ, & find times w/ your group that might work.)

Backtracking: the pattern

Need to make a sequence of decisions:

- Turns in a game ↳ need to choose a space
- Placing a queen → ≤ n decisions
- Is next element in the set? → 2 decisions

So: recursion! (reinforces recursion)

Need a decision

↳ recurse on all possible answers

Requires: some "state" info, so we can build up the solution (or game).

(large)

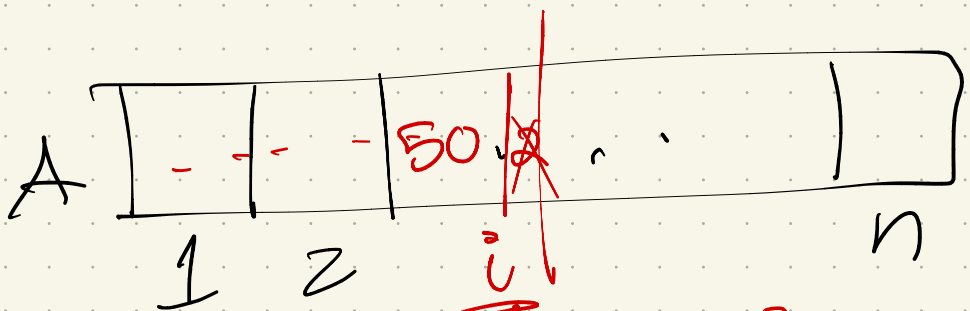
Downside: SLOW

Longest Increasing Subsequence

Why "jump to the middle"?

Need a recursion!

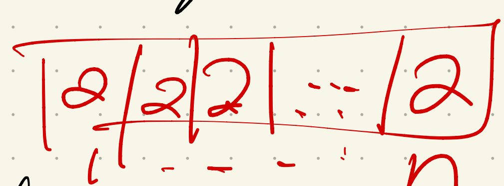
What is our decision?



Do I include A[i] or not?

- if A[i] is too small, skip
- if A[i] is "big enough", try both

Aside: How many subsequences are there?



$=2^n$

Backtracking approach:

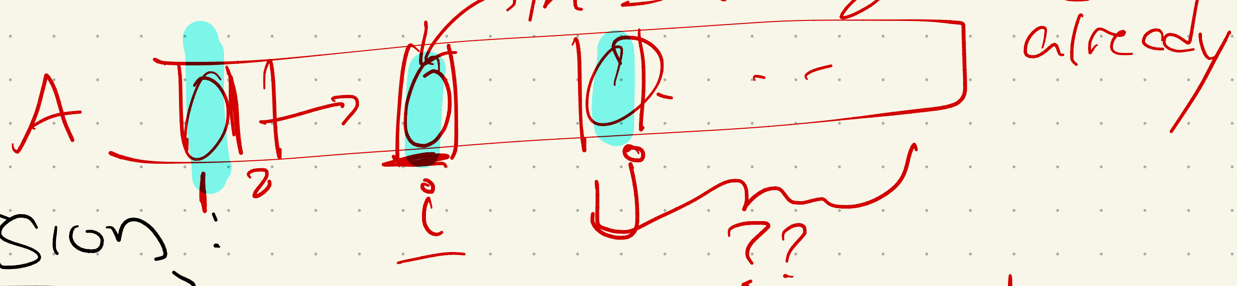
At index i: need last element included

if A[i] < last element, skip

if A[i] > last, try both ways

Result:

Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.



Recursion:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

base case

A[j] is too small

$j \longmapsto j+1$

$\Rightarrow$ base case: $j = n+1$

$\Rightarrow$ A[j] is big enough, so try both ways

Code version:

2 #5

subroutine:

(assuming A is a global)

```
LISBIGGER(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISBIGGER(i, j + 1)
  else
      skip ← LISBIGGER(i, j + 1)
      take ← LISBIGGER(j, j + 1) + 1
      return max{skip, take}
```

:)

Problem – what did we want??

input: A[1..n]

output: length of LIS

what are i + j?

Could call LIS(1, 2) – what would happen?

would include A[1] always – bad!

So:

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```

wrapper

$-\infty$, 10, 1, 2, 3, 4

should get 4

Runtime:

$$L(n) \leq \underline{2}\,L\underline{(n-1)} + O(1)$$

$$L(0) = 1$$

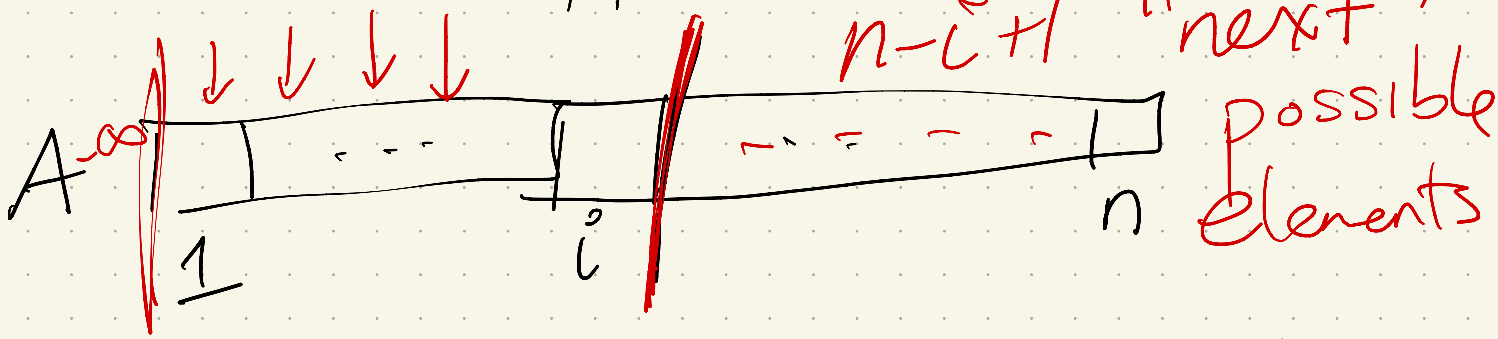
<u>Not</u> Master Thm friendly
no $\frac{n}{c}$ in recursion

Looks like Towers of Hanoi:

$$O(2^n)$$

(can solve, seen in book)

Alternative approach:



At index i, choose next element in the sequence.
(means n calls, not 2!)

```
LISFIRST(i):
    best ← 0
    for j ← i + 1 to n
        if A[j] > A[i]
            best ← max{best, LISFIRST(j)}
    return 1 + best
```

subroutine

Issue - what was our goal again??

Find LIS of A[1..n], LISFIRST(1).

Final version:

```
LIS(A[1..n]):
  best ← 0
  for i ← 1 to n
    best ← max{best, LISFIRST(i)}
  return best
```

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISFIRST(0) − 1
```

choosing first thing to include

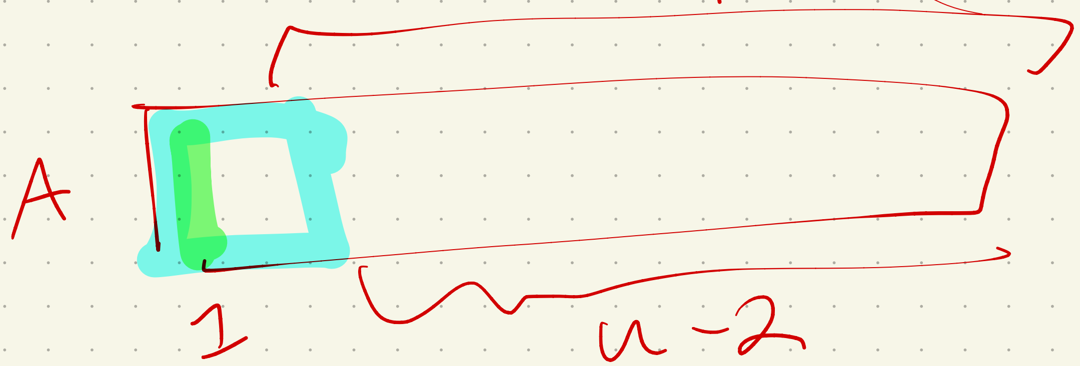
```
LISFIRST(i):
  best ← 0
  for j ← i + 1 to n
    if A[j] > A[i]
      best ← max{best, LISFIRST(j)}
  return 1 + best
```

helper

Runtime:

$$L(n) \le \sum_{i=1}^{n} L(n-i) + O(n)$$

(BAD)



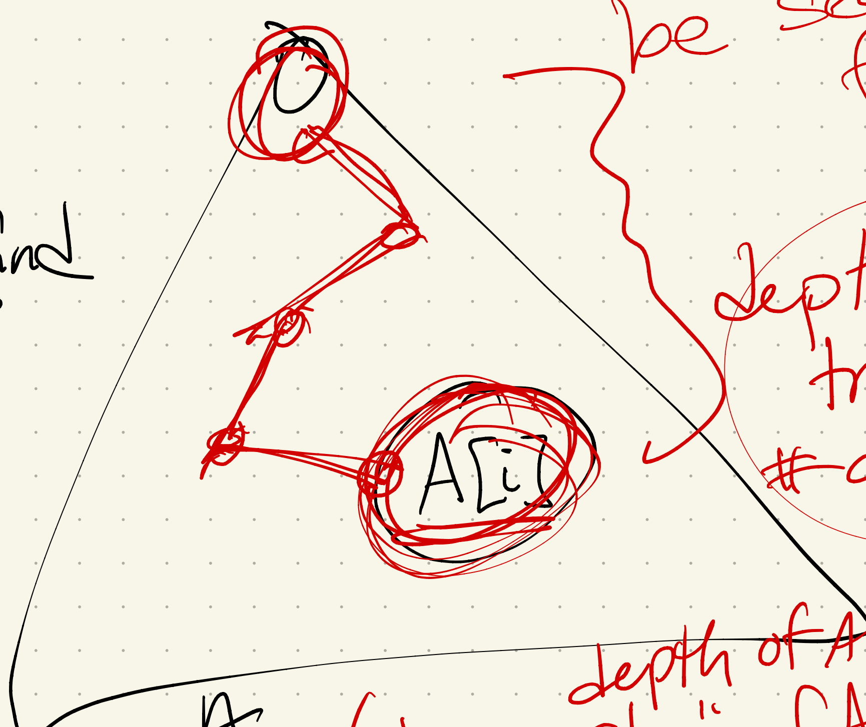
Optimal Binary Search trees:

No big questions flagged here, so hopefully made sense!
This is a huge area of study.

The idea:

- keys $A[1..n]$ go in a tree, sorted order
- access frequency for each is $f[i]$: how many times it will be searched for

Tree:



Cost to find $A[i]$?

depth in tree = # ancestors

$$\text{Cost}(T) = \sum_{i=1}^{n} \underbrace{(\#\text{ ancestors of } A[i])}_{\text{depth of } A[i]} (f[i])$$

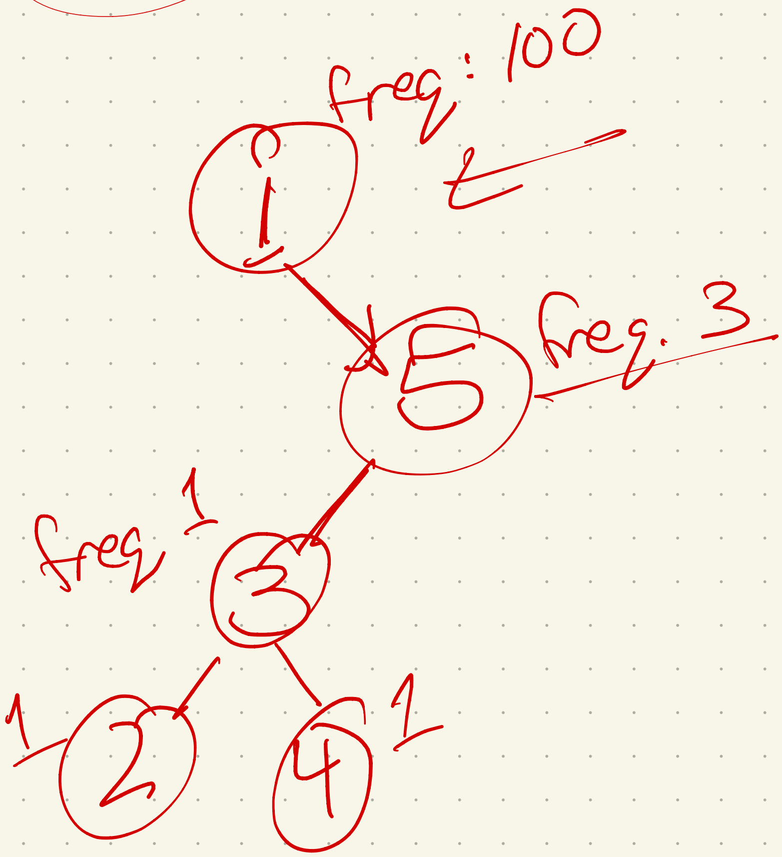
Ex!

$O(1)$ take longer

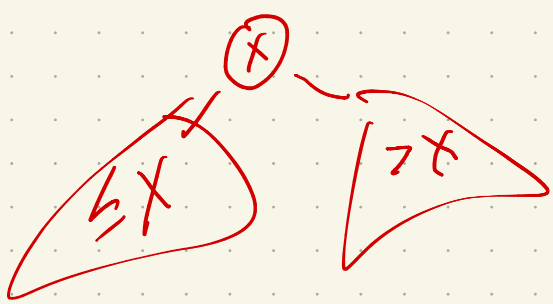
f: 100, 1, 1, 1, 3 ← freq

A: 1, 2, 3, 4, 5 ← keys

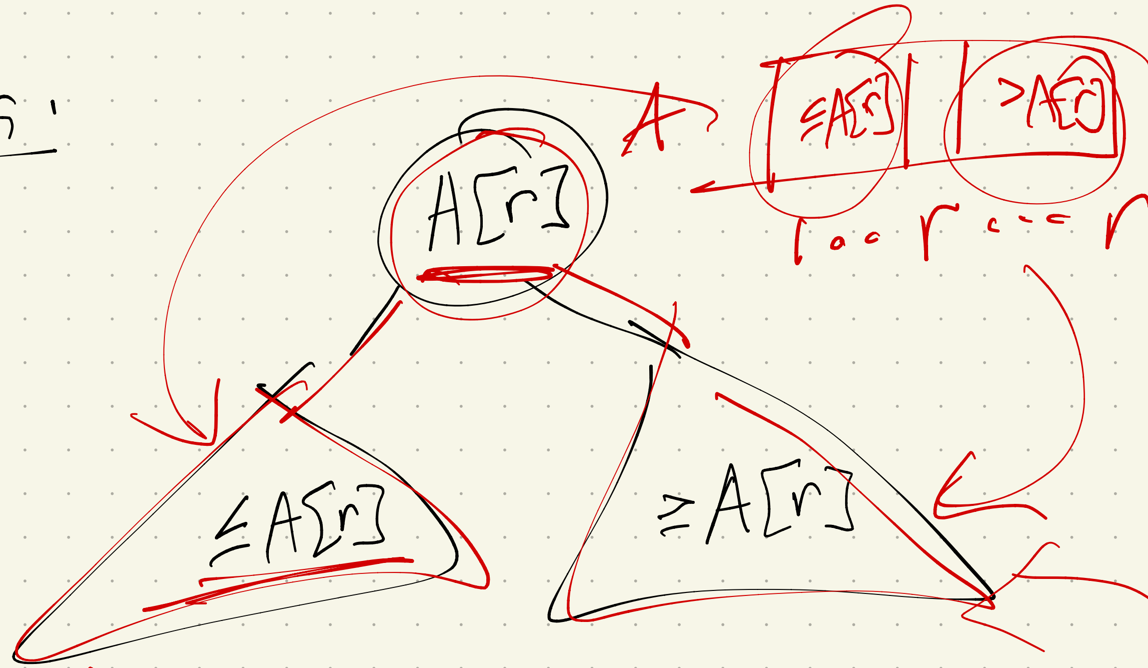
Best:



must be a BST over A



Formula:



Every node pays +1 for the root.

So:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{left}(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{right}(T)$$

$\Longrightarrow$

find best root

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

decision: choose root.

Recurrence:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \begin{Bmatrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{Bmatrix} & \text{otherwise} \end{cases}$$

for each r, try A[r] as root

$$T(n) = \sum_{r=1}^{n} \left(T(r) + T(n-r) \right)$$

$$+ \text{time to calc. cost} = f(n) = O(n)$$

awful: exponential

end of backtracking...

Dynamic Programing

- a fancy term for smarter recursion :

memoization

- Developed by Richard Bellman in mid-1950s

("programming" here actually means planning or scheduling)

Key: When recursing, if many recursive calls to overlapping subcases, remember prior results and don't do extra work!

Simple example:

Fibonacci Numbers

$F_0 = 0, \; F_1 = 1, \; F_n = F_{n-1} + F_{n-2}$

$\forall n \geq 2$

Directly get an algorithm:

```
FIB(n):
if n < 2:
    return n
else
    return FIB(n-1) + FIB(n-2)
```

Runtime:

Applying memoization :

```
MemFibo(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Better yet:

ITERFIBO(n):

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

Correctness:

Run time & space

Even better!

```
ITERFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

Run time/space: