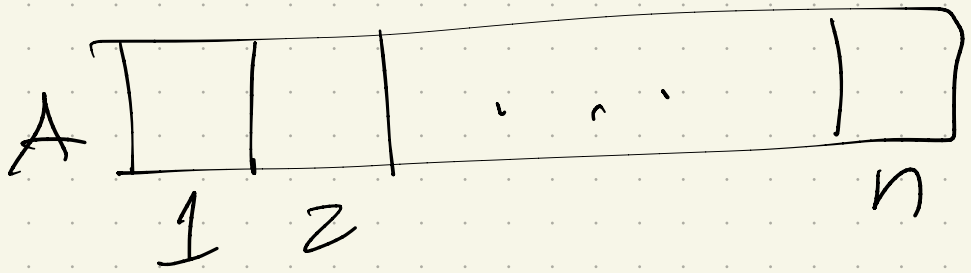
Algorithms — Spring '25

Backtracking:
LIS
Optimal BSTs

Longest Increasing Subsequence

Why "jump to the middle"?
Need a recursion!

First: how many subsequences?



Backtracking approach:

At index i:

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:

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

Code version:

```
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}
```

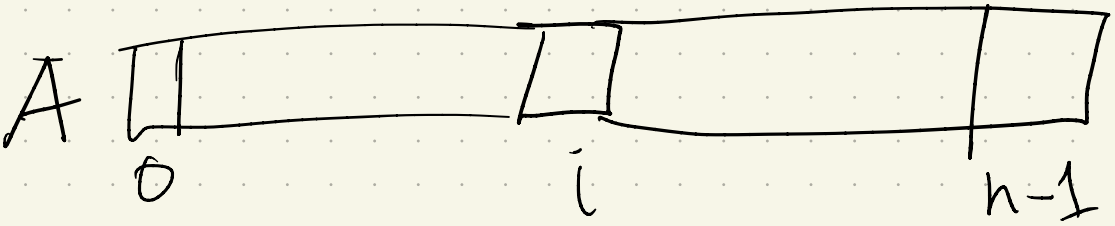
Problem – what did we want??

So:

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

Runtime:

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
```

Issue - what was our goal again??

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
```

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

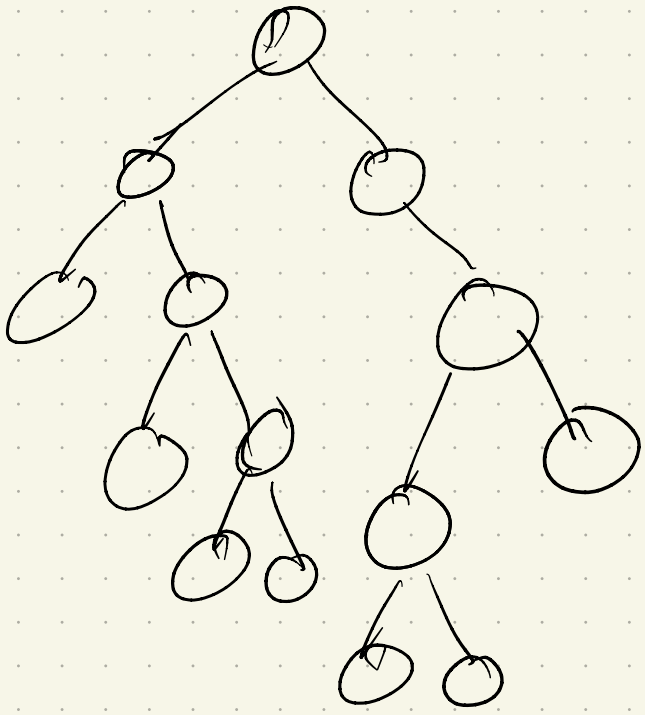
Runtime:

Optimal Binary Search trees:

The idea:

- keys A[1..n] go in a tree, sorted order
- access frequency for each is f[i]

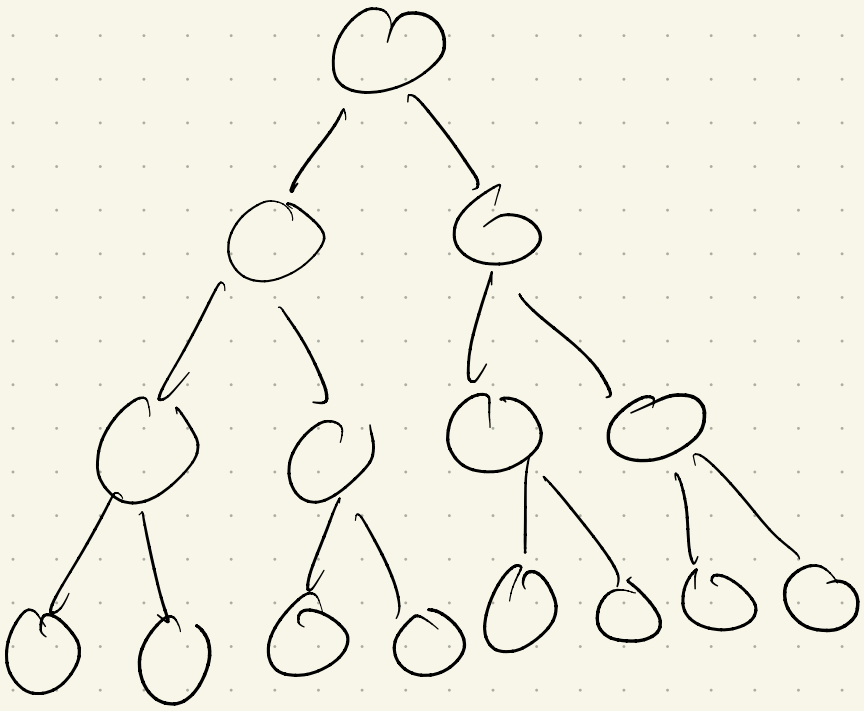
Tree:



Cost to find A[i]?

$$Cost(T) = \sum_{i=1}^{n}$$

Compare to balanced BST:



Example:

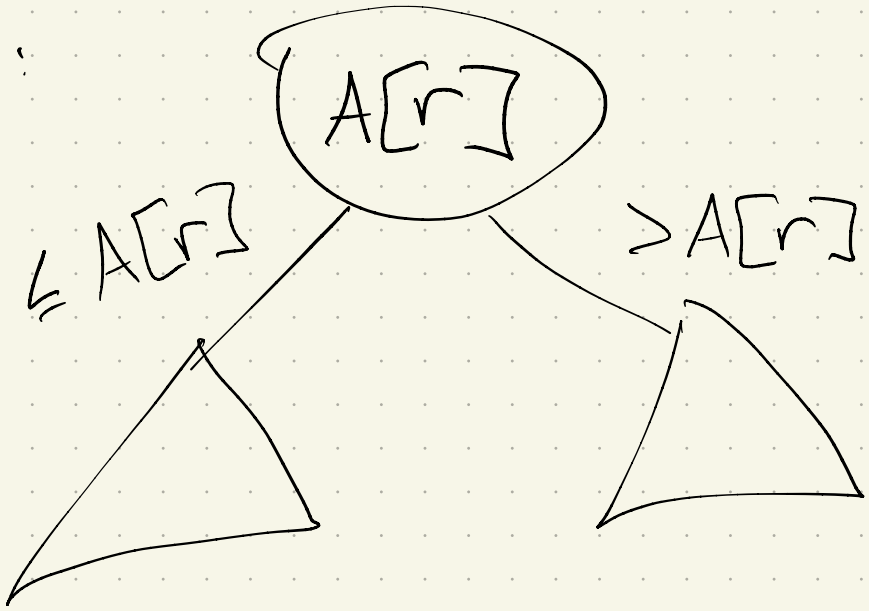
f: 100, 1, 1, 2, 8

A: 1, 2, 3, 4, 5 ← assume sorted

Many BSTs: which is best?

Constructor methods we've studied in data structures:

Formula:



Every node pays +1 for the root, because search path must compare to it.

So:

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

⇒

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

Recurrence:

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

$T(n) =$

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] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← 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: