Algorithms — Spring '25

Backtracking:

LIS

Optimal BSTs

Recap

- Readings posted for next 3 class days

- HW2 posted
  - ↳ Note on runtimes:

    Do want some recurrence/justification

    But don't need to solve if "obviously" exponential

- Reminder: do have slack space

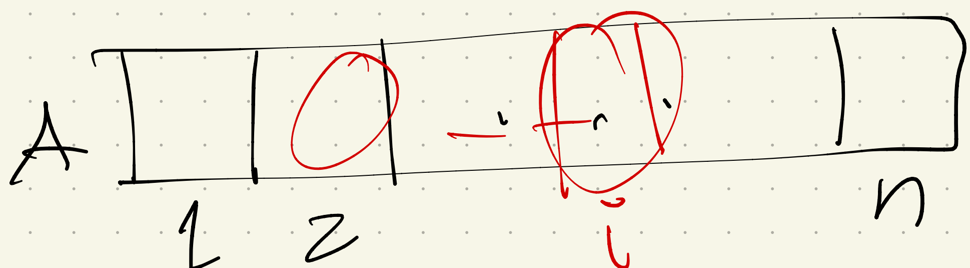
Longest Increasing Subsequence

List of #s. Want longest subseq. which is increasing.

Why "jump to the middle"?

Need a recursion!

First: how many subsequences?



Each element could be in or out: 2^n

Backtracking approach:

At index i: 2 options

include i, recurse from

skip i → $(i+1)$ on

Why not greedy?

$-\infty$

0

$\boxed{\underline{5}}$, 1, 2, 11, 3, 88, −5, 63, 65, 4, 5, 6

LIS(0, 1)

→ LIS(0, 2) → LIS(0,3)

→ LIS(2,3) + 1

LIS(1, 2) + 1

→ LIS[3] is smaller

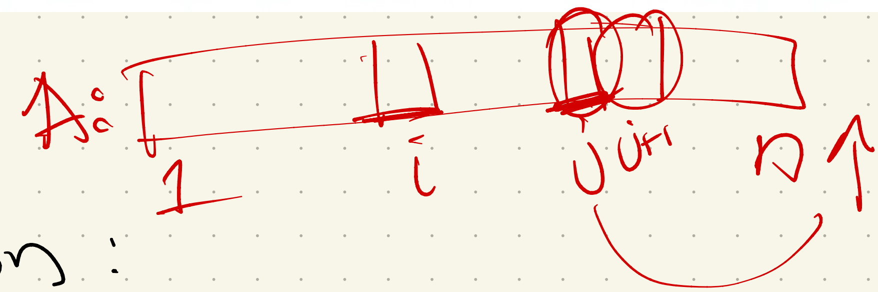
→ LIS(1, 3)

Result:

last element chosen

next one I'm considering

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:

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

take j

or skip j:

A[j] is too small → must skip

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}
```

A[j] is too big → skip

Problem – what did we want??

We wanted Longest Inc Subsequence of A.

So:

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

Runtime:

$$T(n) \leq 2T(n-1) + \underset{O(1)}{6}$$

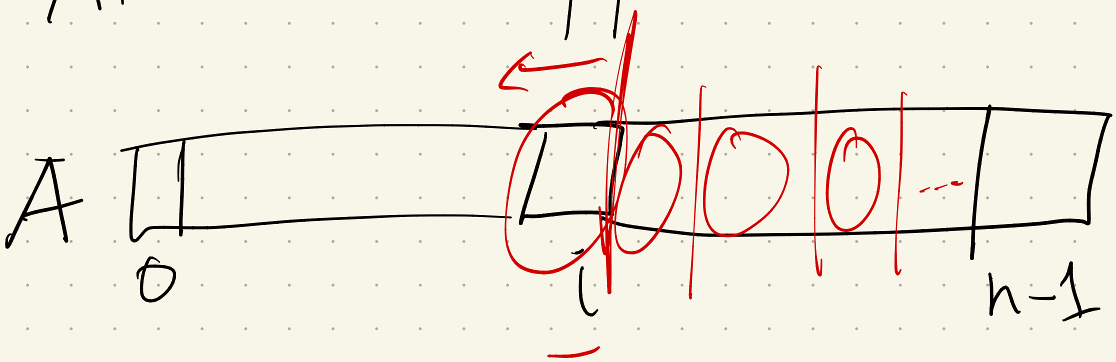
↳ Hanoi-like

$$\Rightarrow T(n) = O(2^n)$$

$$\cancel{T(n) = \sum_{i=1}^{n} T(i) + O(n)}$$

~~exponential~~

Alternative approach:



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

LIS(0) →

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

check if can include A[j]

Issue - what was our goal again??

top level; input was A (no i)

Final version:

choose 1st element

```
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:

$$T(k) = \left[\sum_{i=2}^{k} T(k-i) \right] + O(1)$$

$$= \left(\sum_{i=0}^{k-2} T(i) \right) + O(1)$$

exponential

Optimal Binary Search trees:

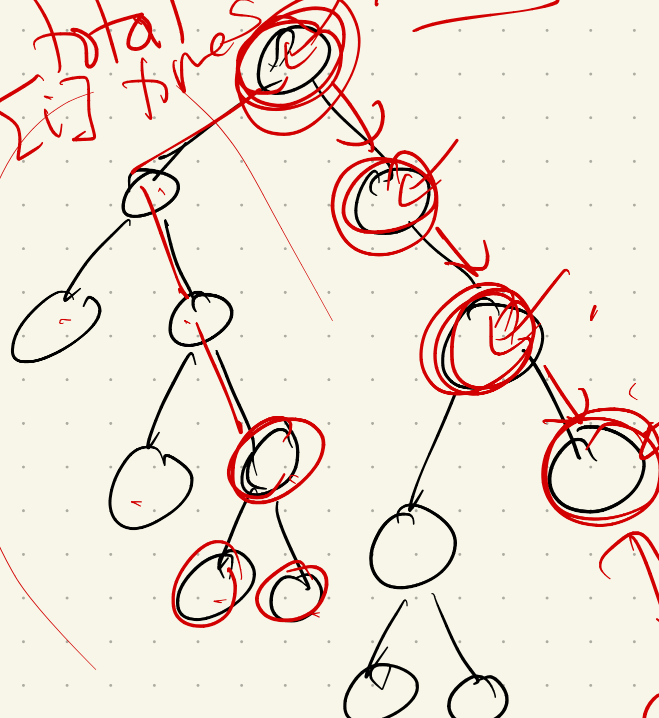
The idea:

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

↳ search for $A[i]$ a total of $f[i]$ times

Tree:

A[8]



A[i]

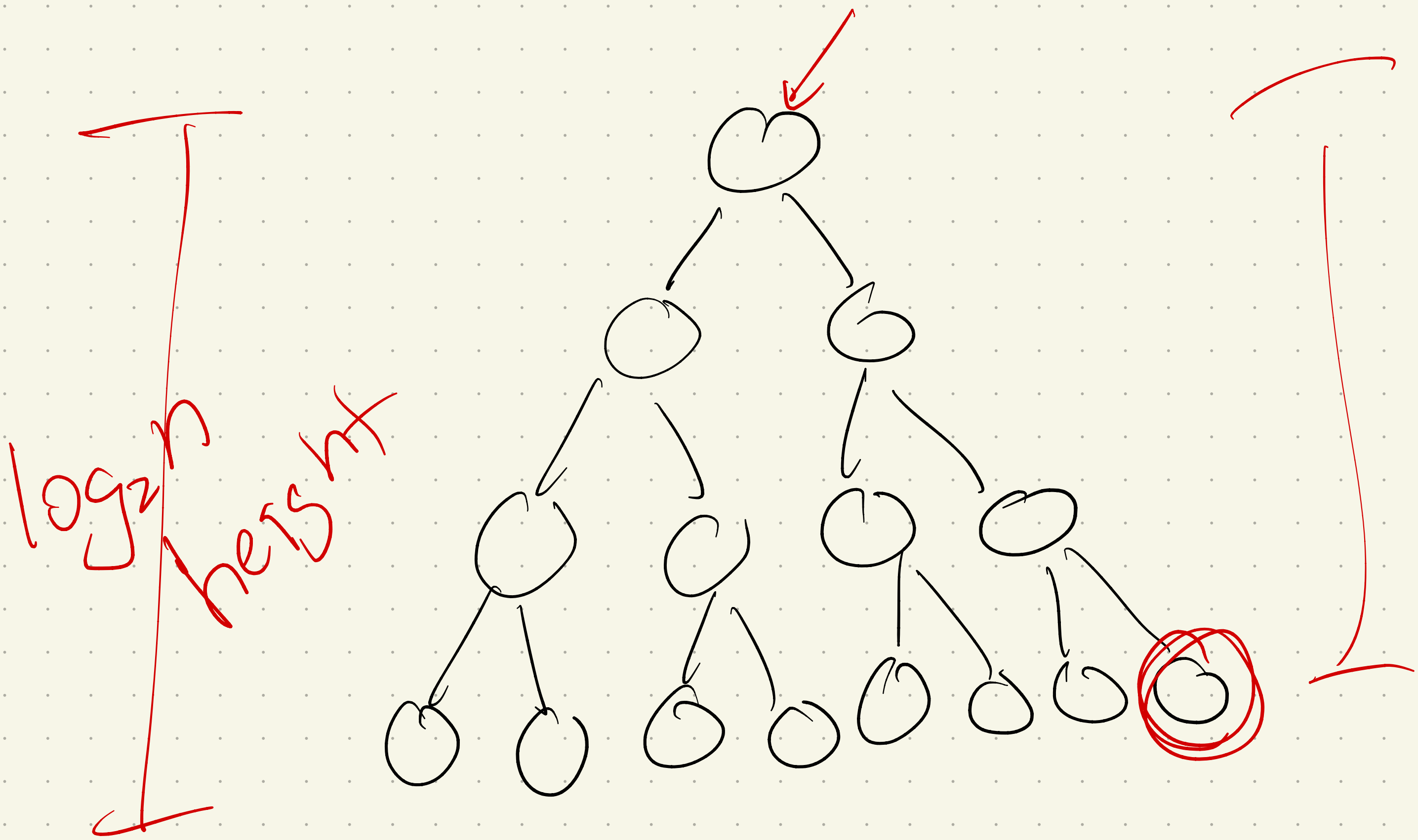
costs 4 = depth(i)

Cost to find $A[i]$?

$= \text{depth}(A[i])$

$$\text{Cost}(T) = \sum_{i=1}^{n} f[i] \cdot \text{depth}(i)$$

Compare to balanced BST:



worst case time

$O(\log_2 n)$

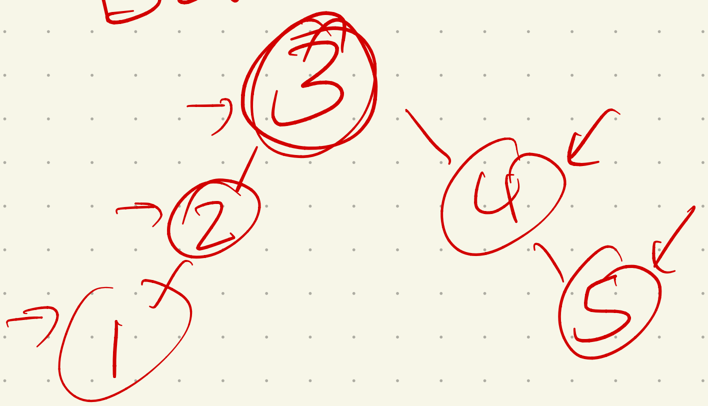
Example:

f: 100, 1, 1, 2, 8

A: 1, 2, 3, 4, 5 ← assume sorted

Many BSTs: which is best?

Balanced:



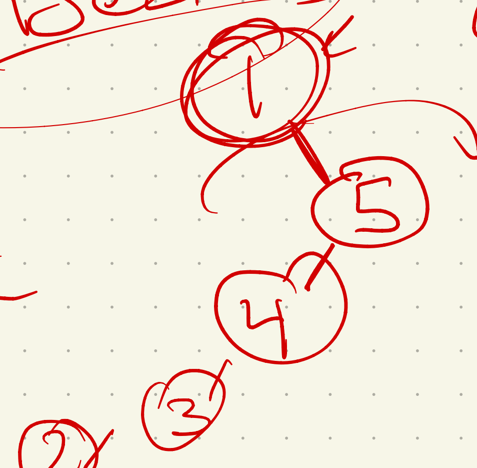
cost: $\sum_{i=1}^{5} f[i] \cdot depth(i)$

$= 100 \cdot 3 + 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 2 + 5 \cdot 3$

Construction methods we've studied in data structures:

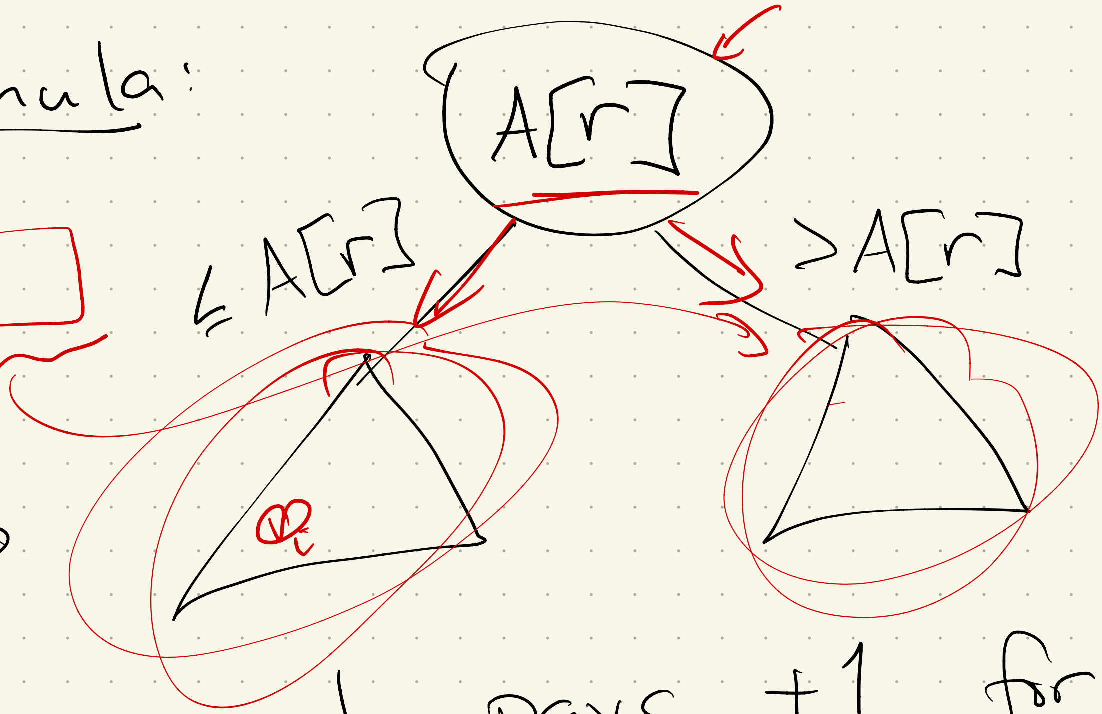
↳ AVL
treap
Red-Black

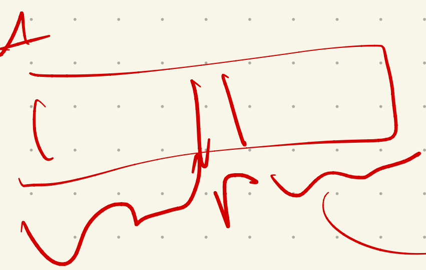
Best



cost =
$100 \cdot 1$
$+ 8 \cdot 2$
$+ 2 \cdot 3$
$+ 1 \cdot 4 + 1 \cdot 5$

Formula:





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

So: Cost(T, f) = Best cost tree w/ freq counts in f

left

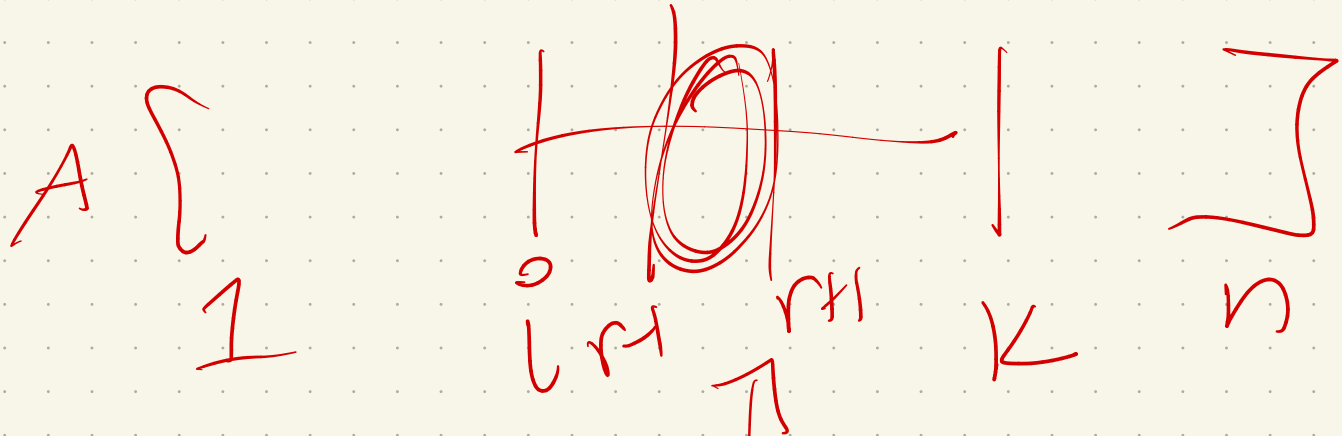
$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$

pays to compare at root

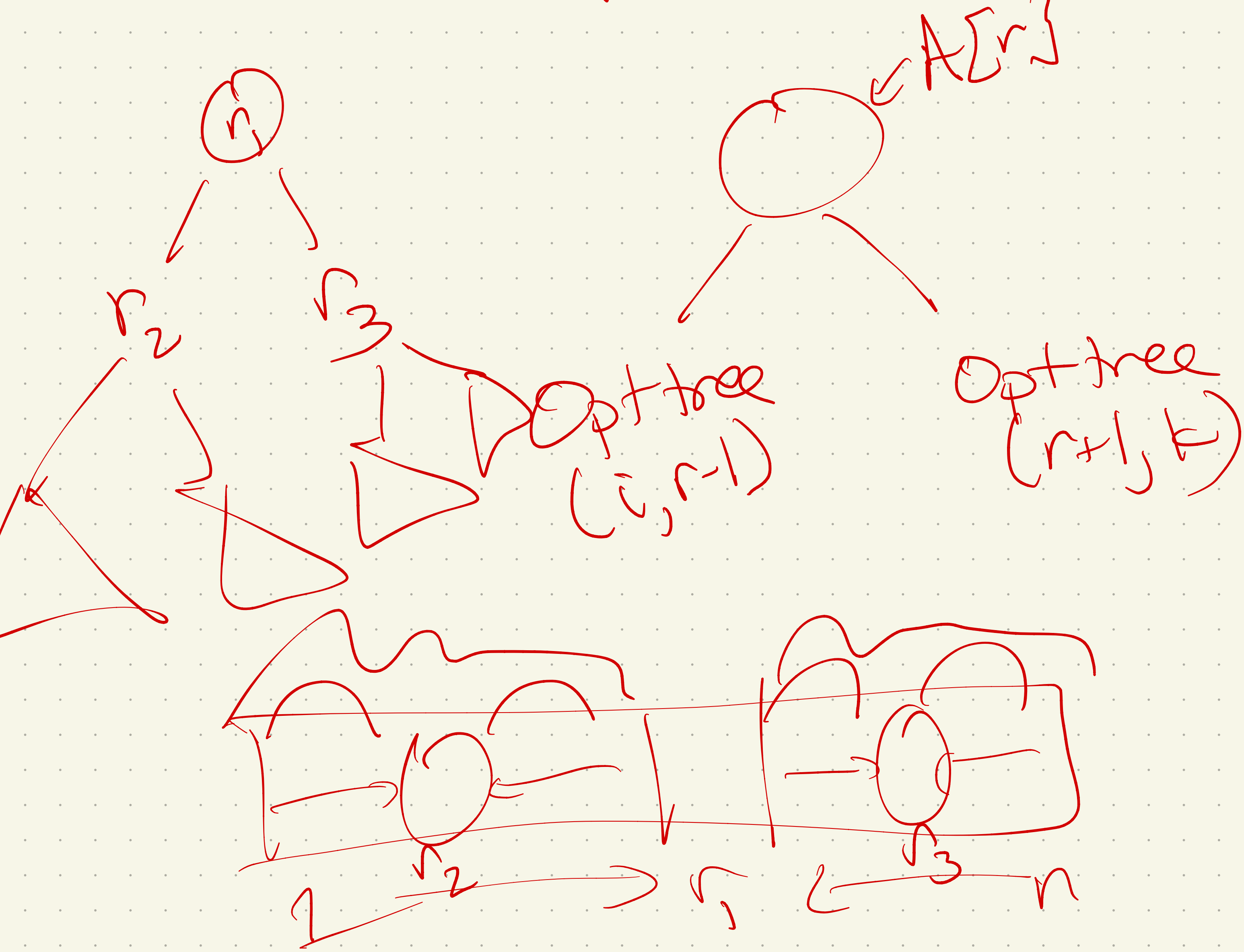
find best root

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

i — roots — k



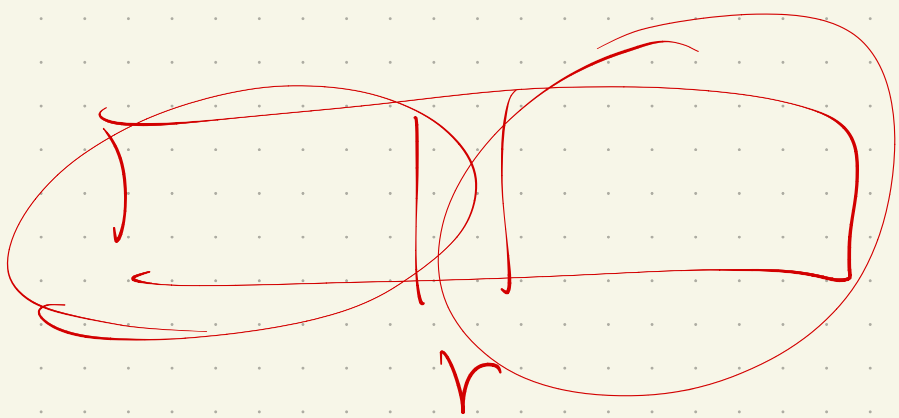
find root r



Recurrence:

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

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



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

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

Directly get an algorithm:

```
FIB(n):
if n < 2:
    return n      0 or 1
else
    return FIB(n-1) + FIB(n-2)
```

Fib(10) → FIB(9) → FIB(8)
FIB(9) → FIB(7)
Fib(10) → FIB(8)

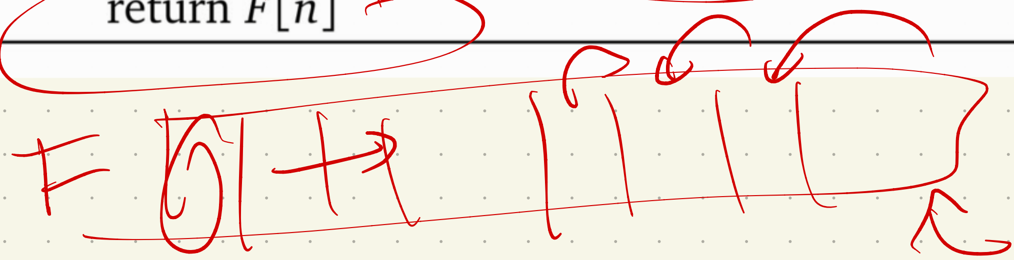
Runtime:

$$F(n) = F(n-1) + F(n-2) + O(1)$$

$$= \Theta(\phi^n) \text{ exponential}$$

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]
```



First time, do the recursion

Later times

↳ look up in table

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

single for loop

$O(n)$

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: