Algorithms – Spring '25

DP:

BSTs (again)

DP on trees

Recap

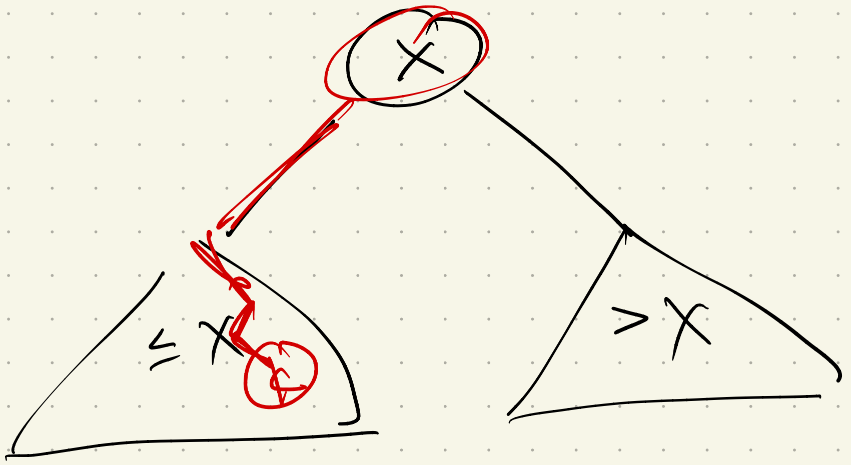
- HW3 posted
- HW1 graded
  ↳ are comments visible now in Gradescope?
- Readings up through next week
- Sub next Monday
  ↳ my office hours will move to Tues & Wed.

Balanced search trees (again)

Recall:

What is the "best" one?

Recap:



Time to search for k in T

$= O(\text{depth in tree of } k)$

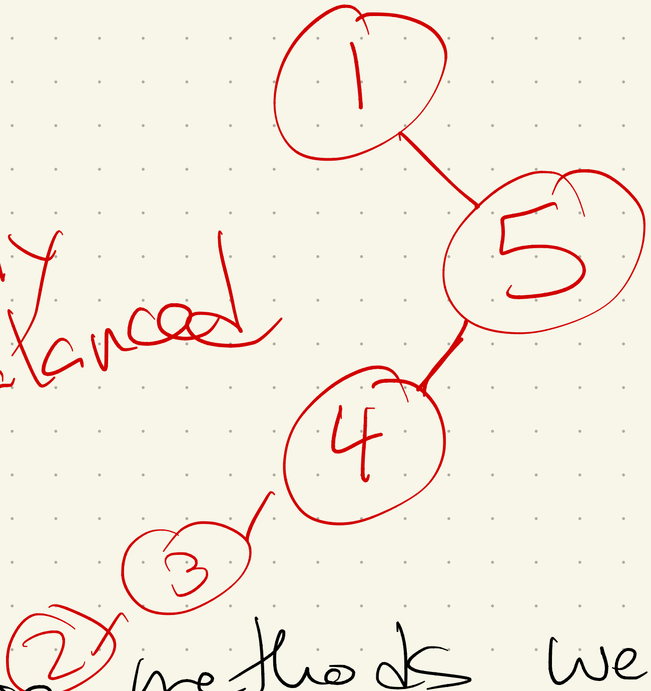
Goal: Given frequencies, built best BST for those frequencies.

Example:

f: 100, 1, 1, 2, 8

A: 1, 2, 3, 4, 5 ↙ assume sorted

Many BSTs: which is best?



highly unbalanced

Construction methods we've studied in data structures:

↳ balanced

His notation:

OptCost[i, k) = 15 ,

Best tree for slice of array from i...k

Ex:

f				
100	1	1	2	8

← frequencies

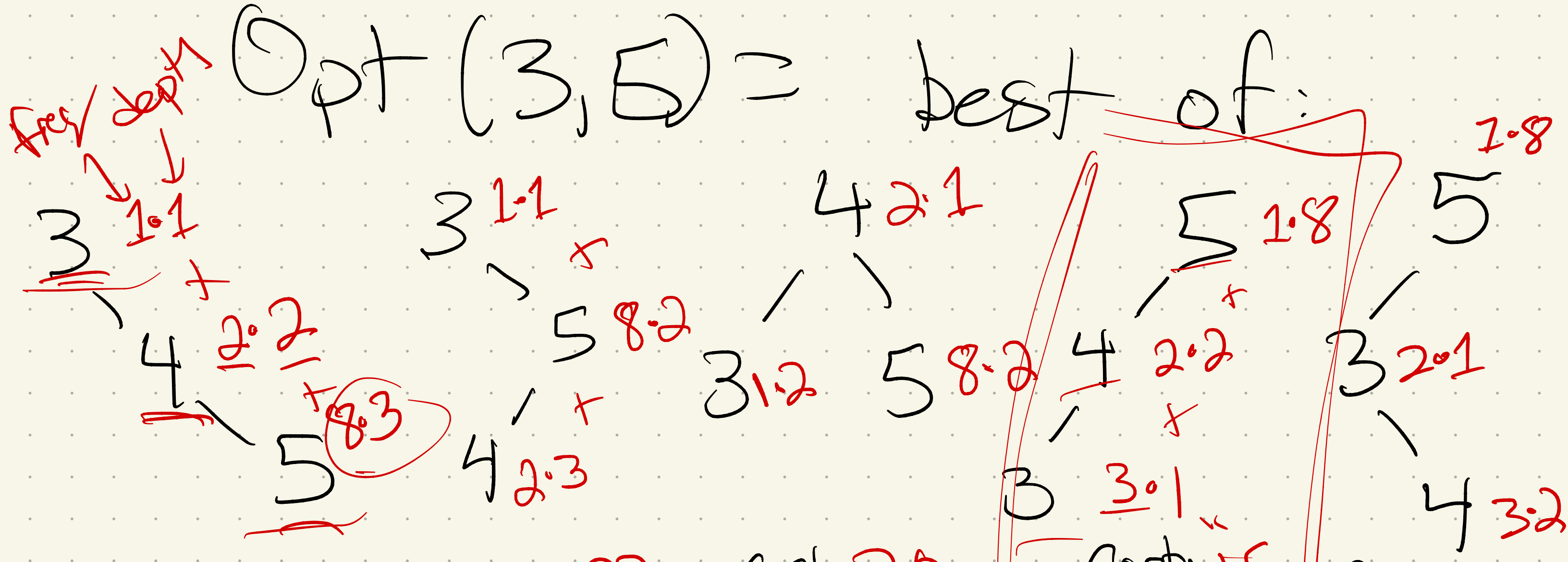
search entries → A

1	2	3	4	5
1	2	3	4	5

← sorted input!

Think brute force!

Opt(3,5) = best of:



Cost: 27 Cost: 23 Cost: 20 Cost: 15 Cost: 16

Here: given $X[1..n]$

$F[1..n]$

element $X[i]$ will have $F[i]$ searches.

Intuitively — want higher $F[i]$ to be closer to the root!

Last chapter:

# of comp w/ root

left items

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

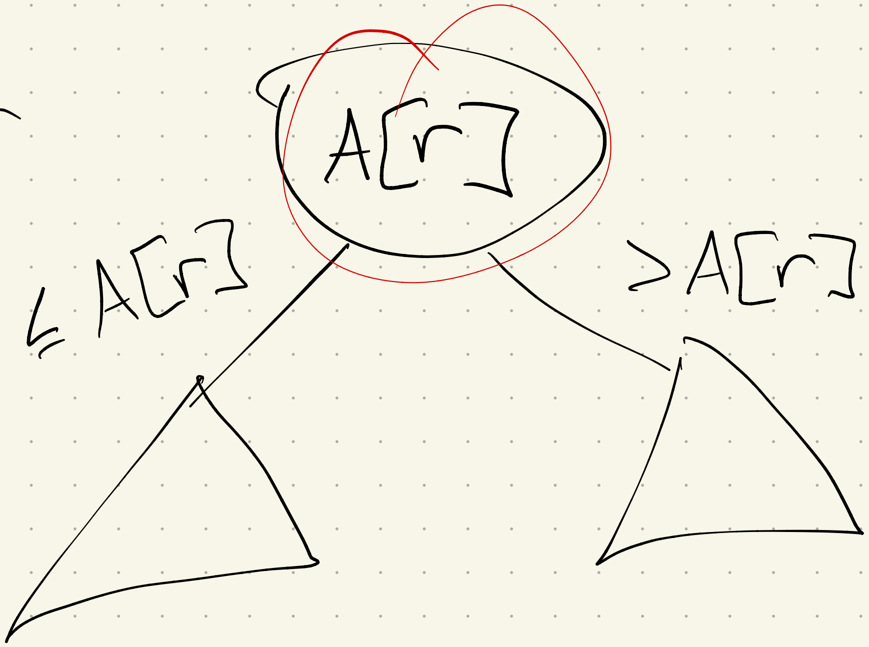
right items

$\Longrightarrow$ $x[r]$ is done

recursion

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

Why??



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

So: we're regrouping!

$$\text{Cost}(T) = \sum_{i=0}^{n-1} F[i]\cdot(\text{depth in tree})$$

$$= \sum_{\text{levels } i \text{ in tree}} (\text{sum of frequencies of nodes in level } i \text{ or deeper})$$

Here: level 0 = root $\sum F[i]$

rest: recursion

Optcost(1, n)

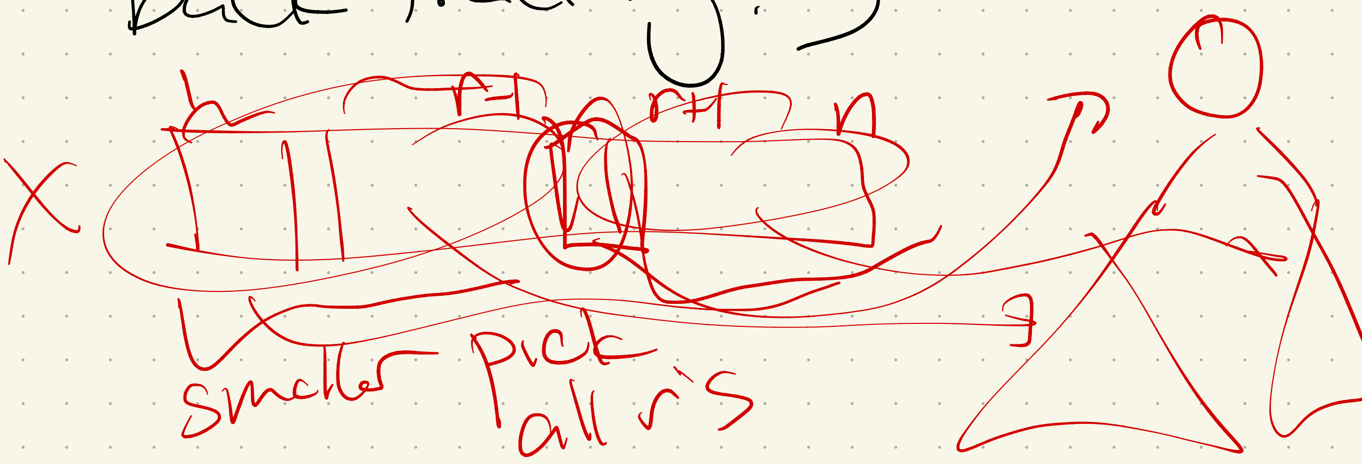
$$\boxed{OptCost(i,k)} = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \boxed{\min_{i \le r \le k}} \begin{Bmatrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Use this to build the "best" tree:

Choose root. r

Recursively find best left subtree, + best right subtree.

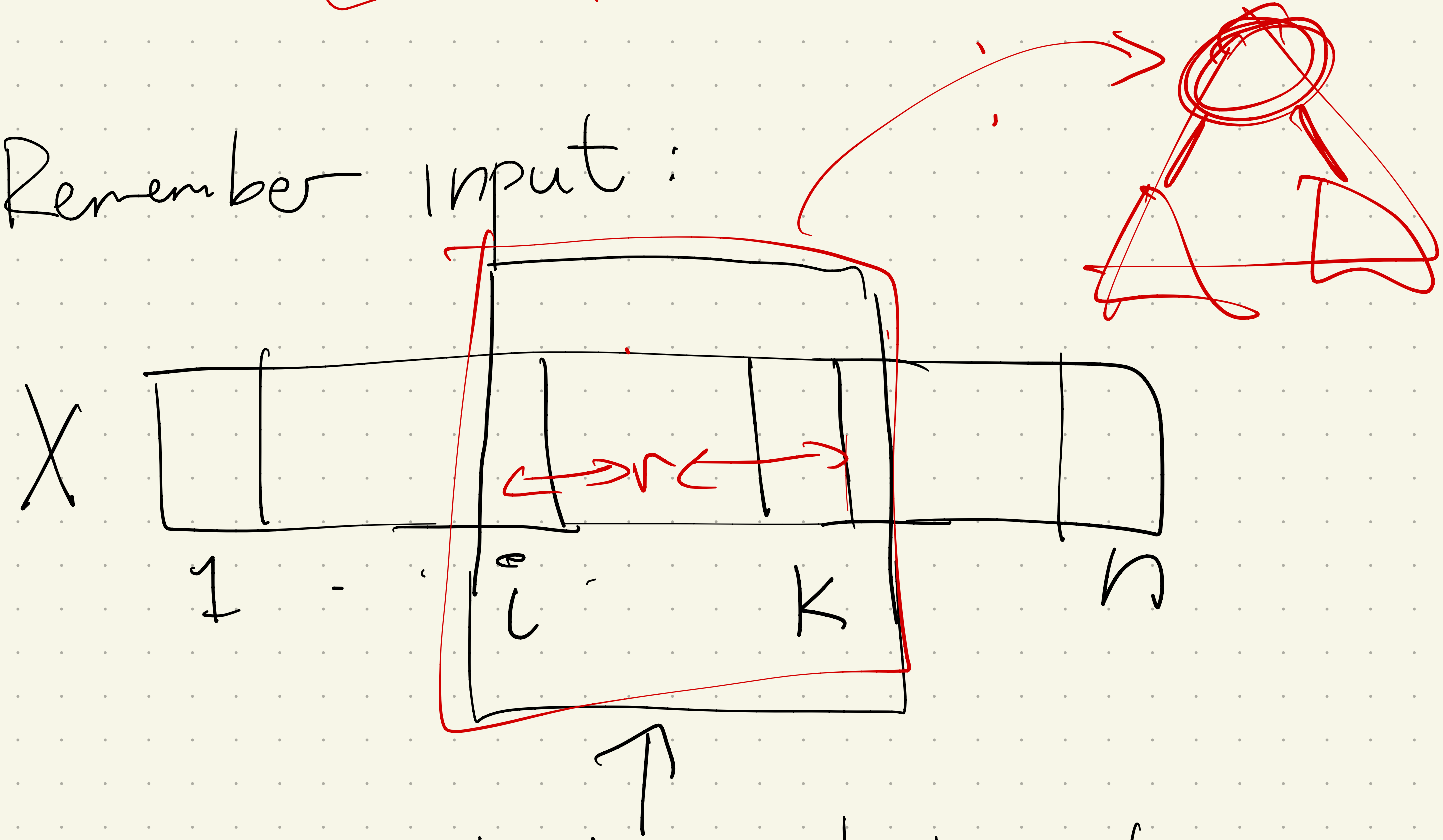
(Note: try all roots in backtracking!)



How to memoize?

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

Remember input:

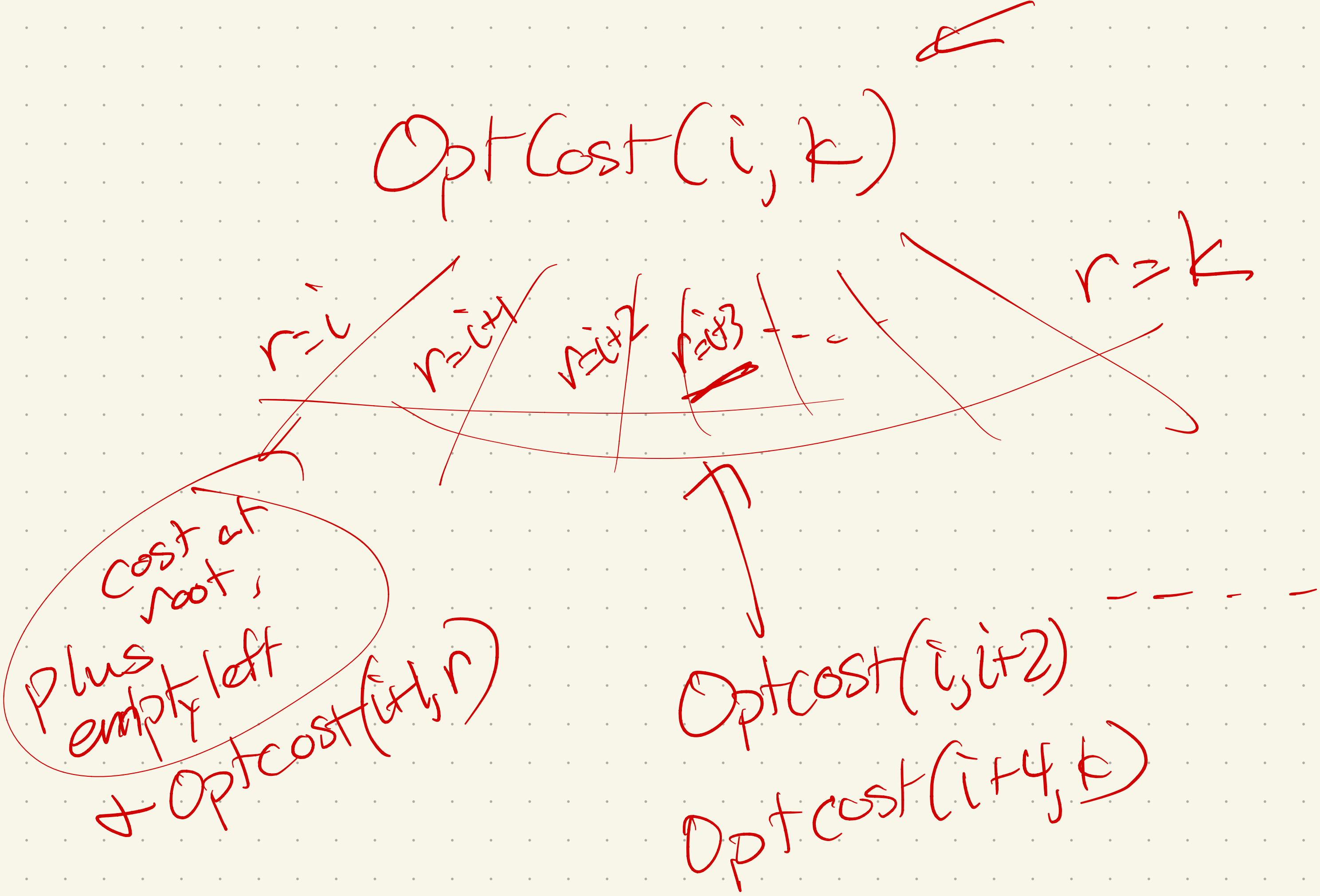


build best tree here

Everyone here pays $\sum_{j=i}^{k} f[i]$,

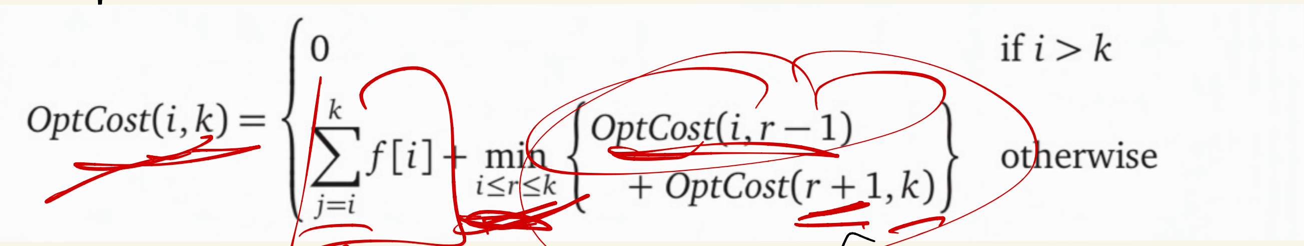
so first precompute & store these sums.

Time/space: $O(n^2)$


OptCost(i, k)
r=i
r=i+1
r=i+2
r=i+3
- - -
r=k
cost of root, plus empty left
+ Optcost(i+1, r)
Optcost(i, i+2)
Optcost(i+4, k)
- - - -

Let $F[i][k] = \sum_{j=i}^{k} f[j]$

Now:



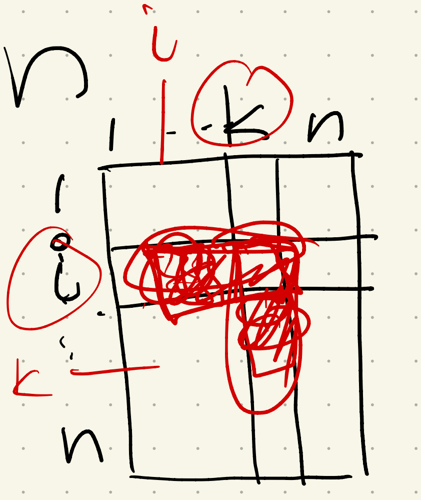
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{c} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

$F[i][k]$ ⇓

$$OptCost(i,k) = \begin{cases} 0 \\ F[i][k] + \underline{\qquad} \end{cases}$$

Memoize: $0 \le i \le k \le n$

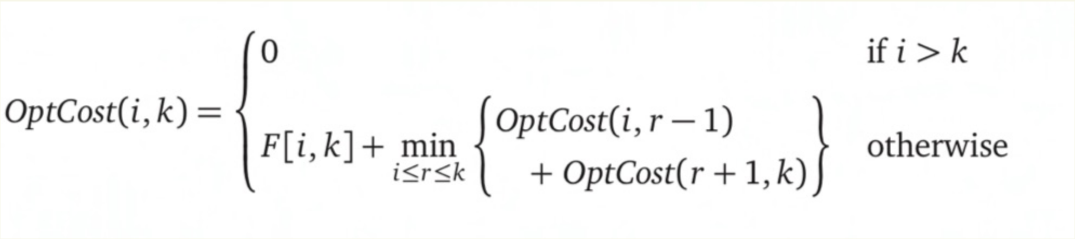
So: 2d table!



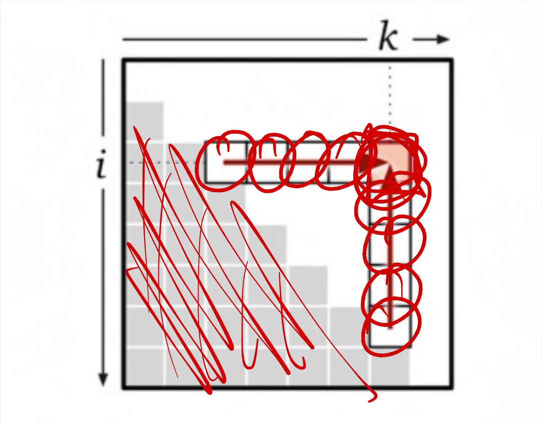
Each $O[i][k]$ needs:

- $F[i][k]$
- and lookup slice of row + column it lives in

His picture (prettier):

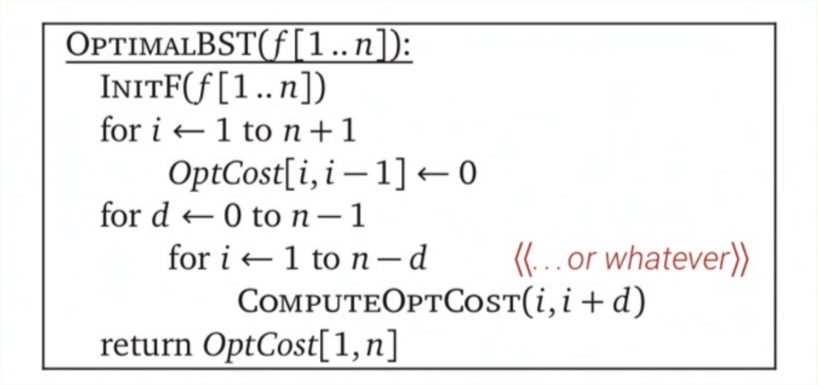


$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$



$x[i \dots k]$

So:



```
OPTIMALBST($f[1..n]$):
    INITF($f[1..n]$)
    for $i \leftarrow 1$ to $n+1$
        $OptCost[i, i-1] \leftarrow 0$
    for $d \leftarrow 0$ to $n-1$
        for $i \leftarrow 1$ to $n-d$        ⟨⟨...or whatever⟩⟩
            COMPUTEOPTCOST($i, i+d$)
    return $OptCost[1, n]$
```

Time: $O(n)$ time per cell in array

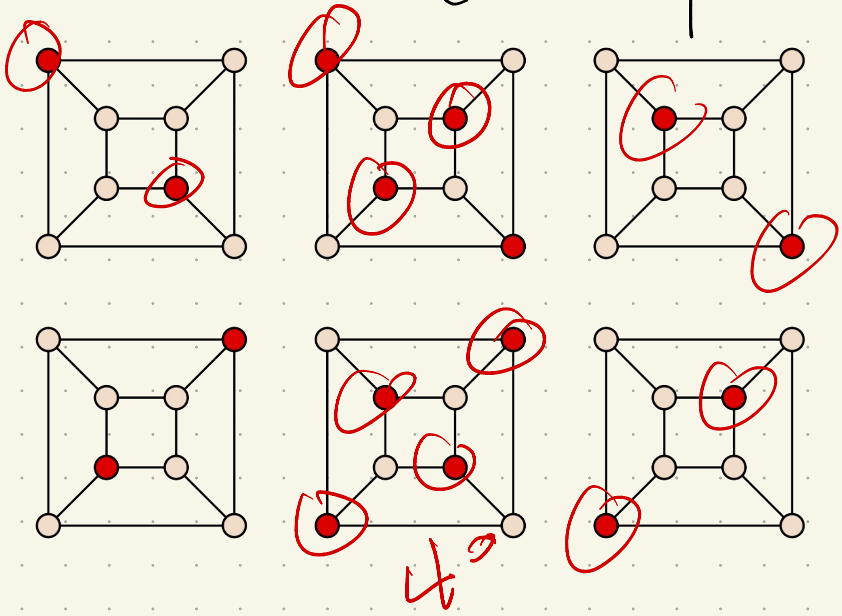
$\Rightarrow O(n^3)$

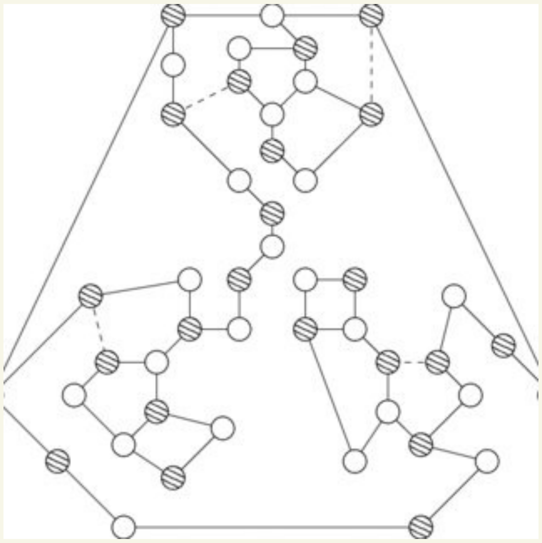
Space: $O(n^2)$

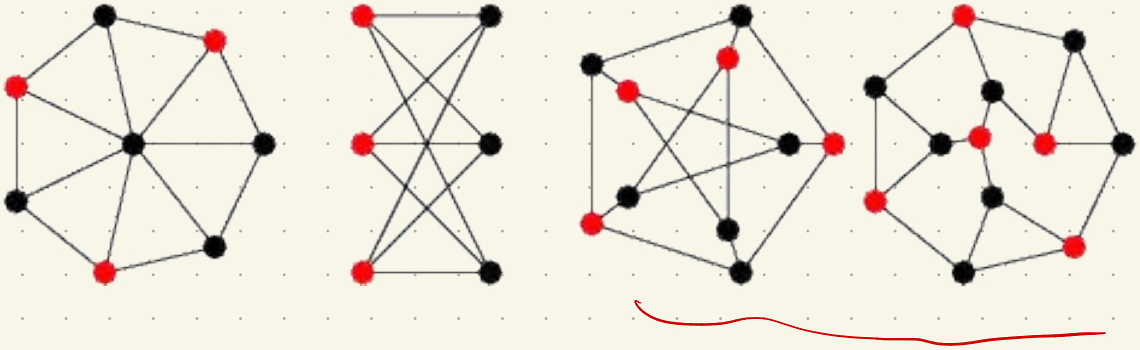
Dynamic Programming on Trees

Independent Set :

(nice preview of graphs)



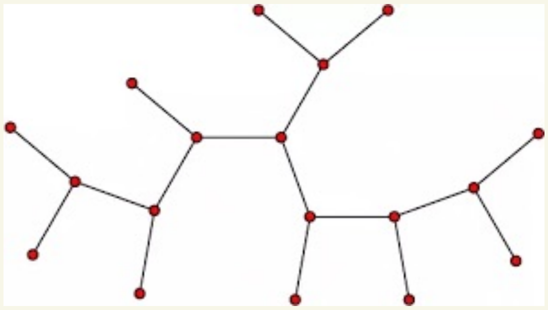


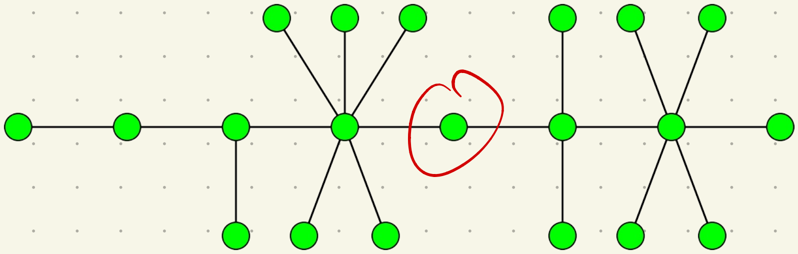


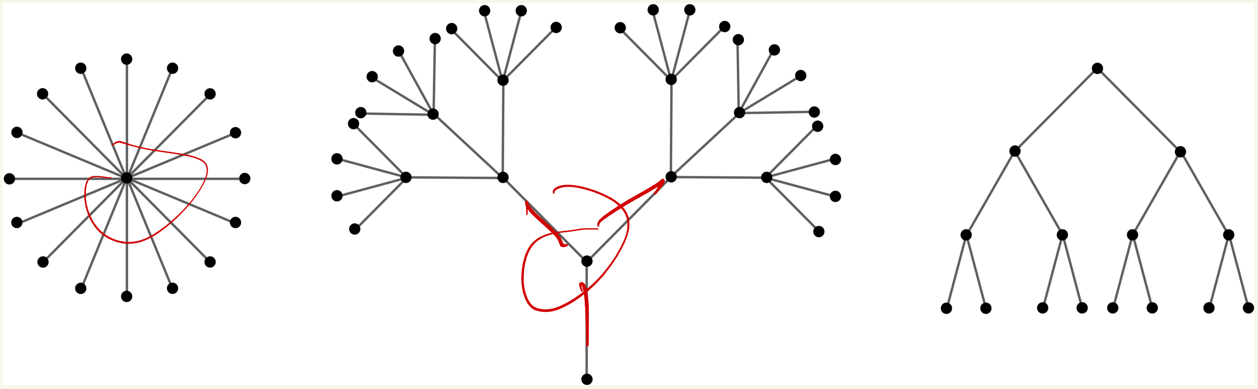
Notoriously hard!
But- can solve on simpler graphs.

Trees:

Not always binary!



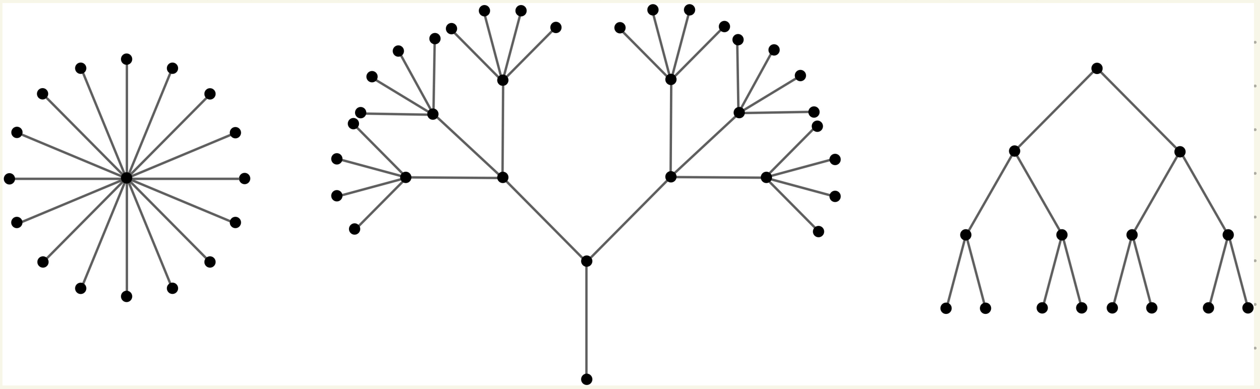




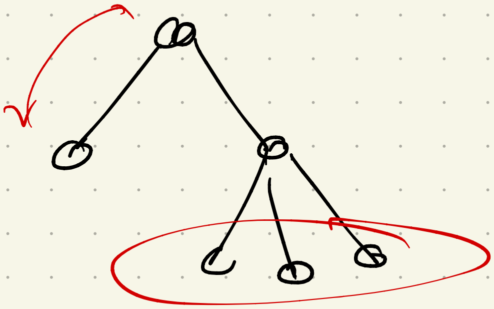
Dfn: Connected, acyclic graph.

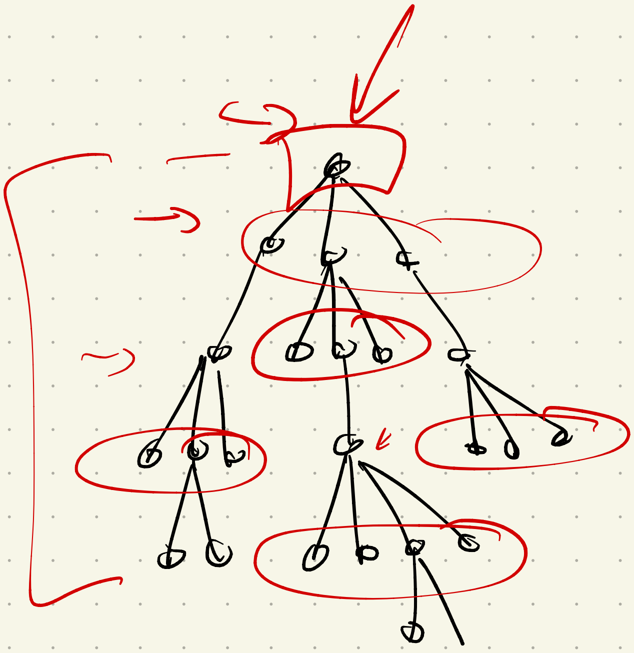
Here, we will "root" the tree.

Independent set in a tree:



Less clear:





So - not always "grab biggest level".

(ie - don't be greedy!!)

Recursive approch:

Consider the root.

Could include, or not.

Backtracking!

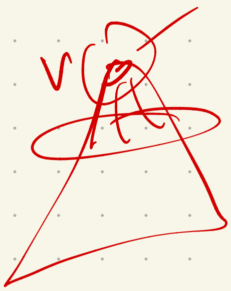
$$MIS(v) = \max \begin{cases} 1 + \sum_{w \text{ a grandchild of } v} MIS(w) & \text{include } v \text{ (must skip } v\text{'s children)} \\ \sum_{w \text{ a child of } v} MIS(w) & \text{don't include } v \end{cases}$$

for v being in set

(w ↓)

↳ could have children

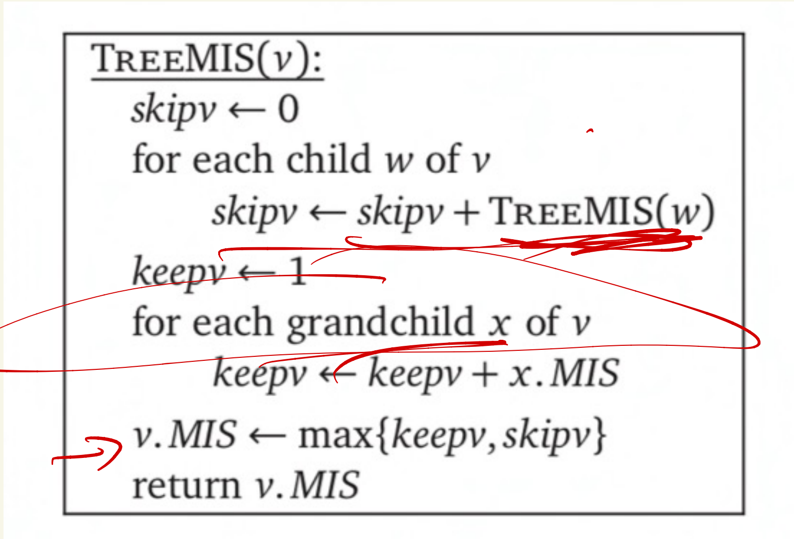
$MIS(v)$ = Max indep set in subtree rooted at node v



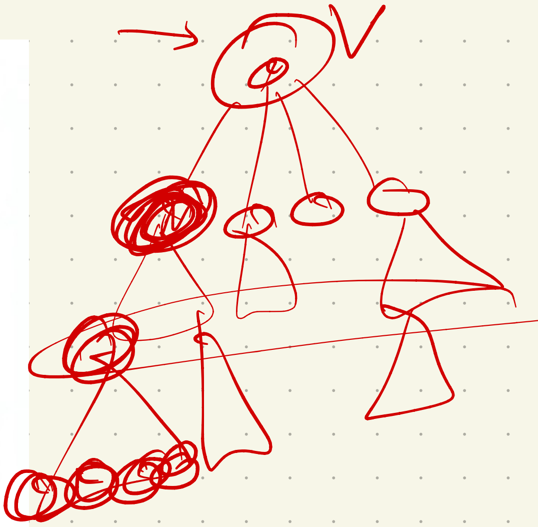
base case:

leaf: = 1

This recurrence (in code):



```
TREEMIS($v$):
    $skipv \leftarrow 0$
    for each child $w$ of $v$
        $skipv \leftarrow skipv + \text{TREEMIS}(w)$
    $keepv \leftarrow 1$
    for each grandchild $x$ of $v$
        $keepv \leftarrow keepv + x.MIS$
    $v.MIS \leftarrow \max\{keepv, skipv\}$
    return $v.MIS$
```



Q: Given this recursion, are we calling any function too often?

Yes! Each node called while a child and a grandchild → memoize!

How to memoize:

Well, for each node, need the best set in that subtree.

Even better - 2 values!
(same big-O)

For each v, store

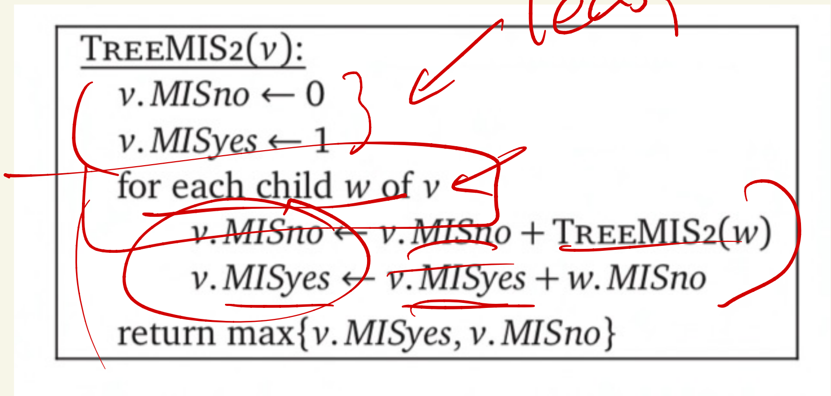
- Best set with v
- Best set without v

Think data structures:

$$\text{Node } v = \begin{cases} v.\text{with} \\ v.\text{without} \end{cases}$$

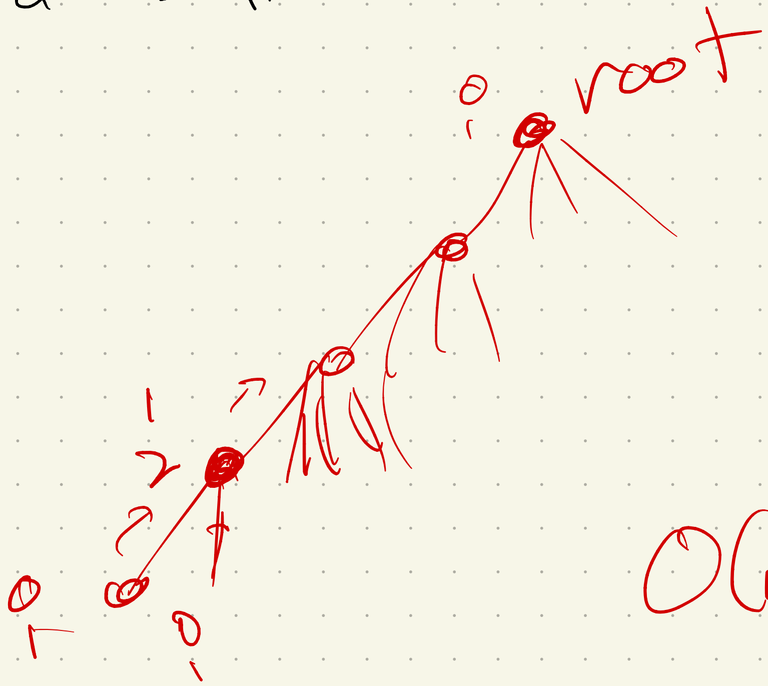
So: Use a tree for the data structure!

if leaf, done



```
TreeMIS2(v):
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + TreeMIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```

Note: At heart, still a post-order traversal.



O(n) time

O(n) space

Dynamic Programing vs Greedy

Dyn. pro: try all possibilities
↳ but intelligently!

In greedy algorithms, we avoid building all possibilities.

How?

- Some part of the problem's structure lets us pick a local "best" and have it lead to a global best.

But - be careful!

Students often design a greedy strategy, but don't check that it yields the best global one.

Overall greedy strategy:

- Assume optimal is different than greedy
- Find the "first" place they differ.
- Argue that we can exchange the two without making optimal worse.

⟹ there is no "first place" where they must differ, so greedy in fact is an optimal solution.

First example in the book:

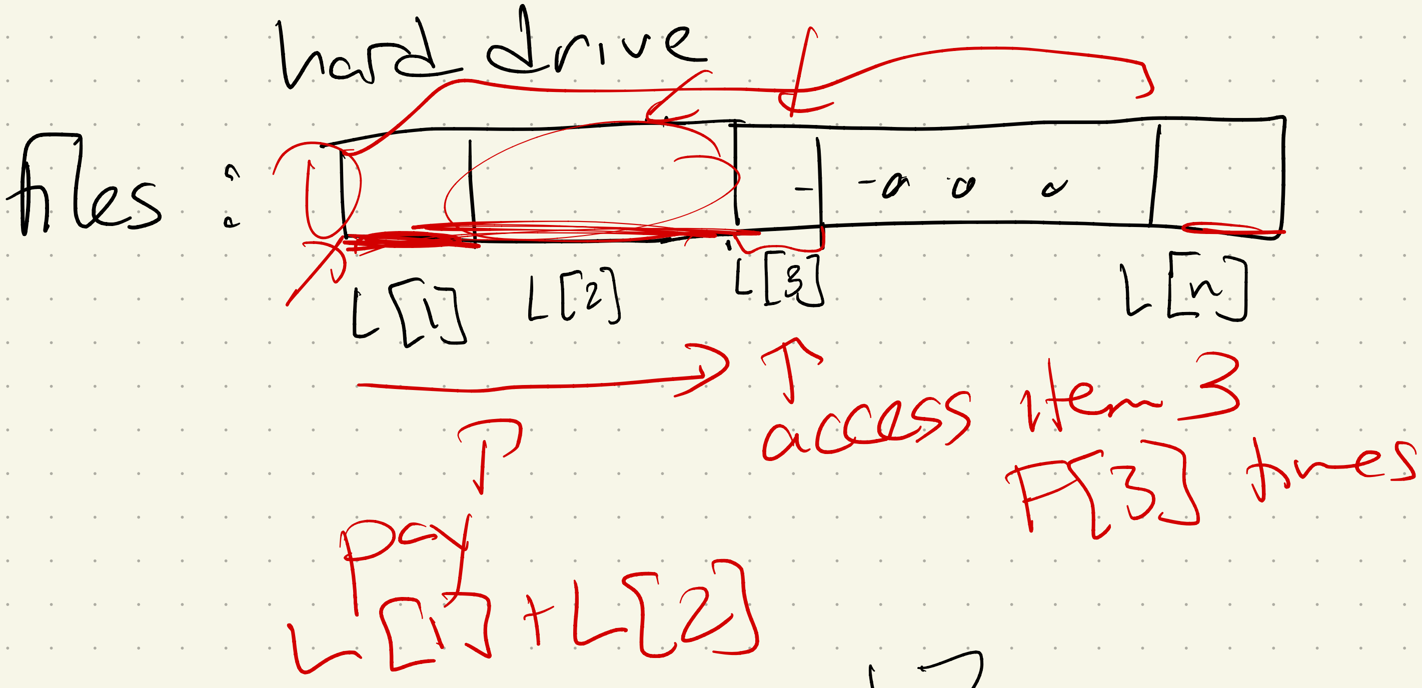
Storing files on tape.

Input: n files, each with a length & # times it will be accessed:

$L[1..n]$ & $F[1..n]$

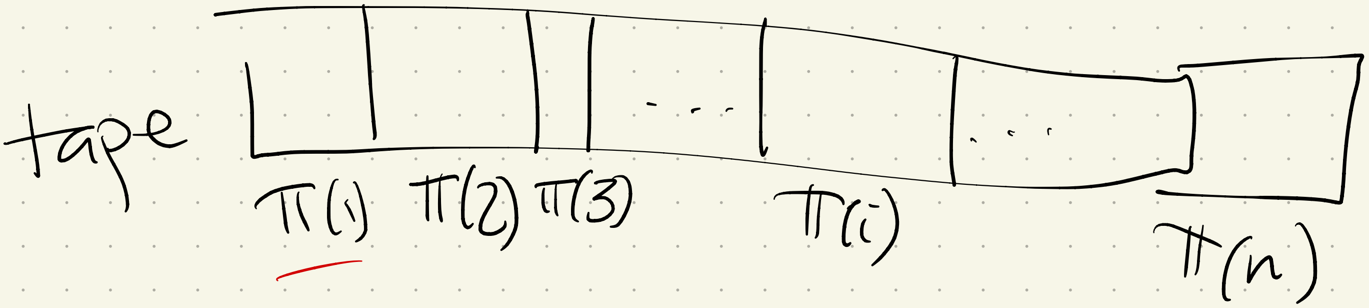
length — frequency

Goal: Minimize access time:



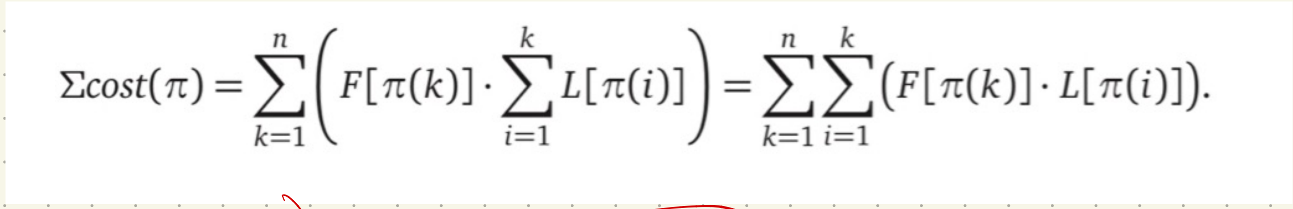
Cost? →

Files: order π:



cost to access ith one:

Total:



$$\Sigma cost(\pi) = \sum_{k=1}^{n}\left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)]\right) = \sum_{k=1}^{n}\sum_{i=1}^{k}\left(F[\pi(k)] \cdot L[\pi(i)]\right).$$

How to be greedy?
(Not immediately clear!)

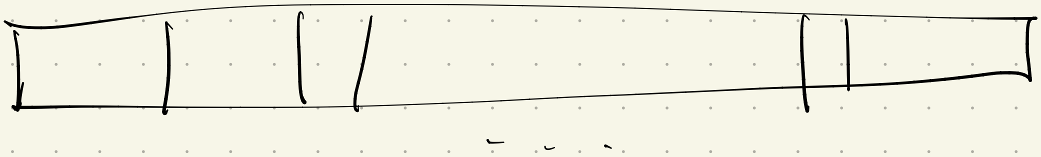
Try smallest first:

Try most frequent first:

Lemma: Sort by $\frac{L[i]}{F[i]}$

& will get optimal schedule.

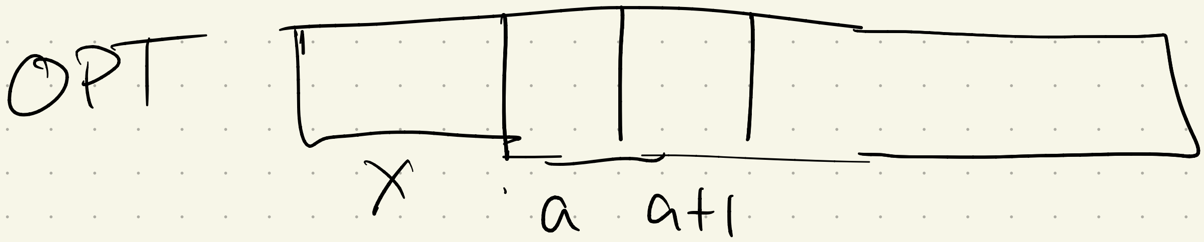
pf: Suppose we sort:



& $\forall i, \frac{L[i]}{F[i]} \leq \frac{L[i+1]}{F[i+1]}$

Suppose this is not optimal.
What does that mean?

Well, OPT must be different, so ∃ out of order pair:



with $\frac{L[a]}{F[a]} > \frac{L[a+1]}{F[a+1]}$

If OPT, must beat our "sorted" solution.

What if we swap a & a+1?

Before:

After:

difference?

Pf (cont):

So: algorithm

- Calcutate $\frac{L[a]}{F[a]}$ for all a.
- Sort, + permute order of jobs to match.

Runtime: