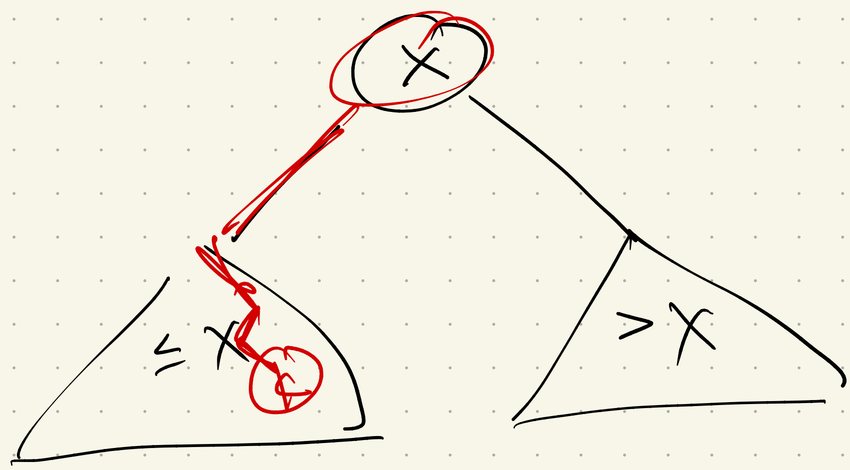
Algorithms – Spring '25

Balanced search trees (again)

Recall:

What is the "best" one?

Recap:



Time to search for k in T

= O(depth in tree of k)

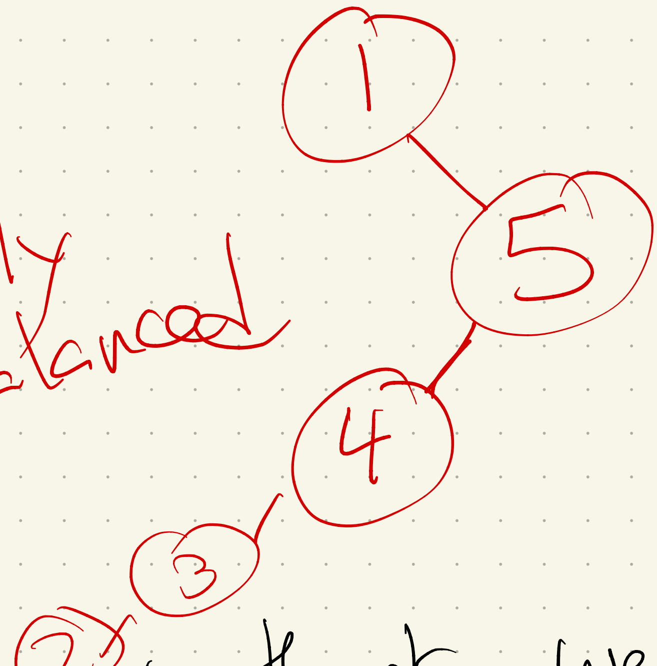
Goal: Given frequencies, built best BST for those frequencies.

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

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!

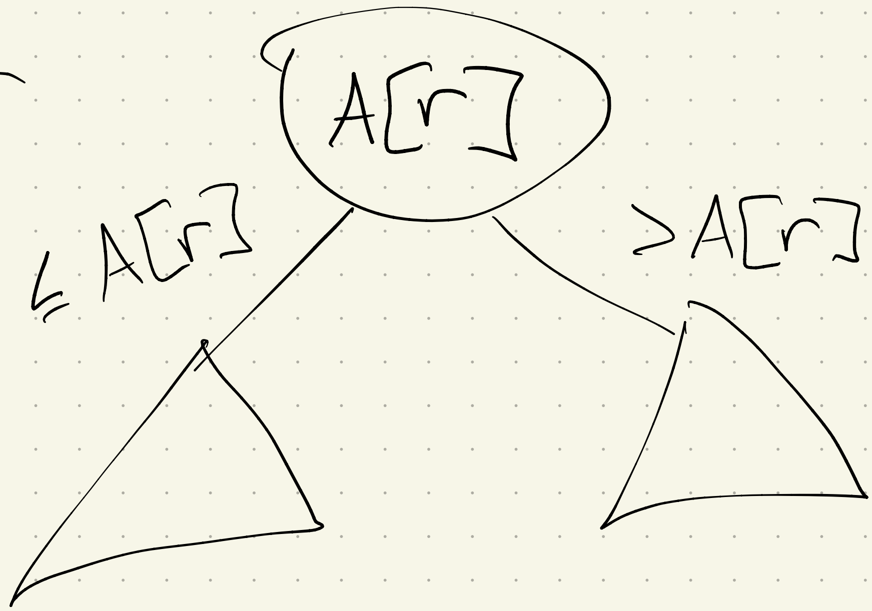
Last chapter:

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

$\Rightarrow$

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

Why??



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

So: we're regrouping!

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

$$= \sum_{\substack{\text{levels } i \\ \text{in tree}}} \left(\begin{array}{l}\text{sum of frequencies} \\ \text{of nodes} \\ \text{in level } i \\ \text{or deeper}\end{array}\right)$$

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

Use this to build the "best" tree:

Choose root.

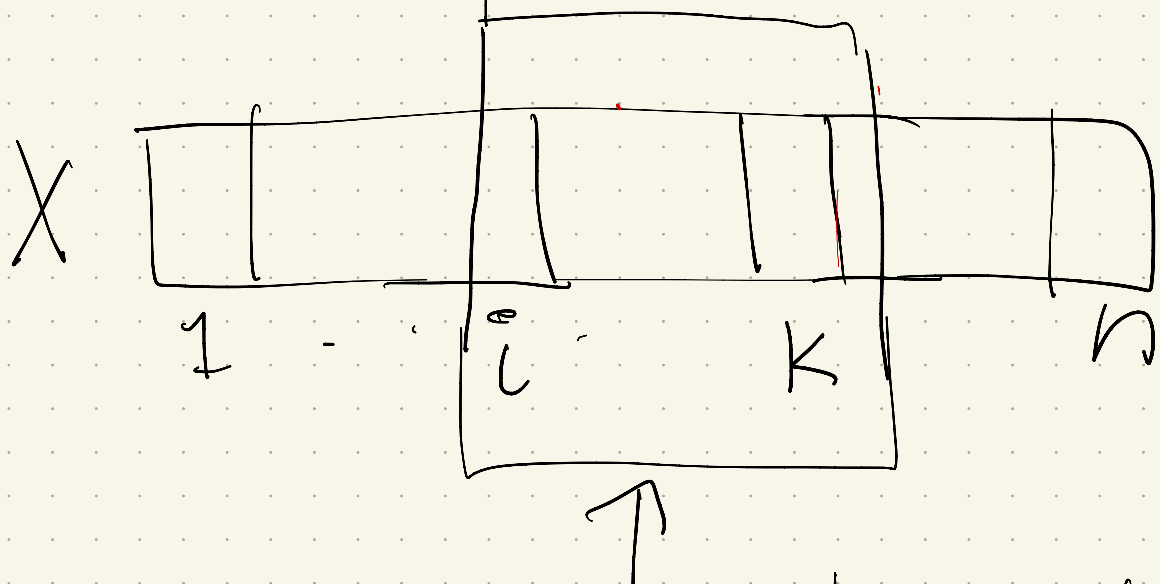
Recursively find best left subtree, + best right subtree.

(Note: try all roots in backtracking!)

How to memoize?

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

Remember input:



build best tree here

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

so first precompute & store these sums.

Time/space:

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

Now:

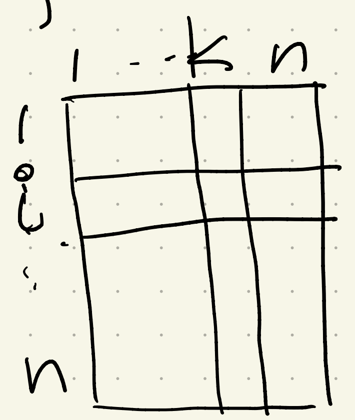
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{matrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{matrix} \right\} & \text{otherwise} \end{cases}$$

$$\Downarrow$$

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

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

So: 2d table!

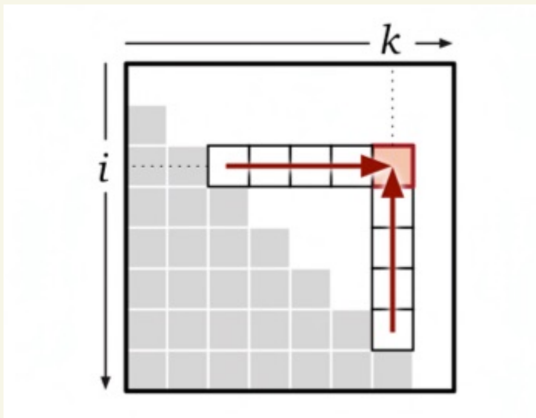


Each $O[i][k]$ needs:

- $F[i][k]$
- and

His picture (prettier):

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



So:

```
OptimalBST(f[1..n]):
   InitF(f[1..n])
   for i ← 1 to n+1
        OptCost[i, i−1] ← 0
   for d ← 0 to n−1
        for i ← 1 to n−d          ⟨⟨...or whatever⟩⟩
             ComputeOptCost(i, i+d)
   return OptCost[1, n]
```

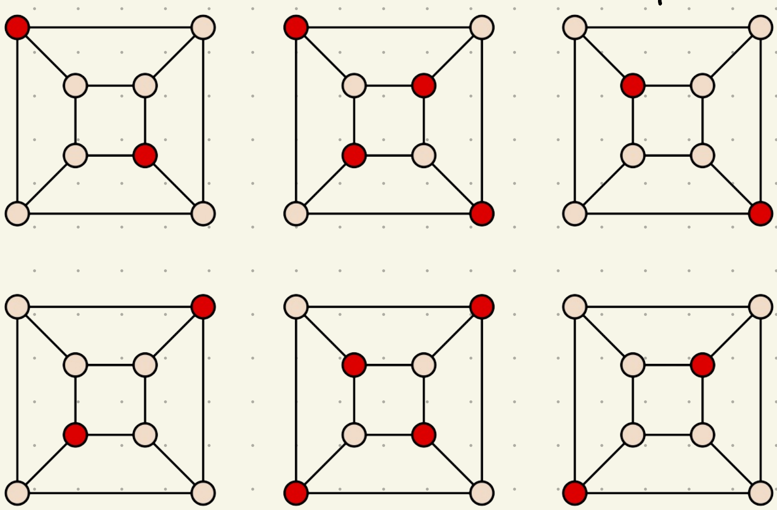
Time:

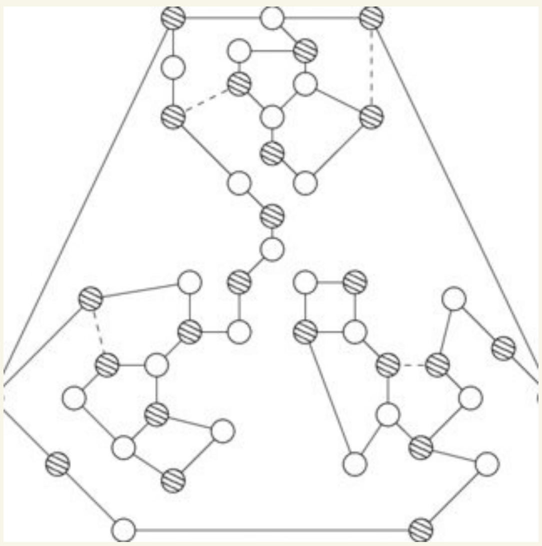
Space:

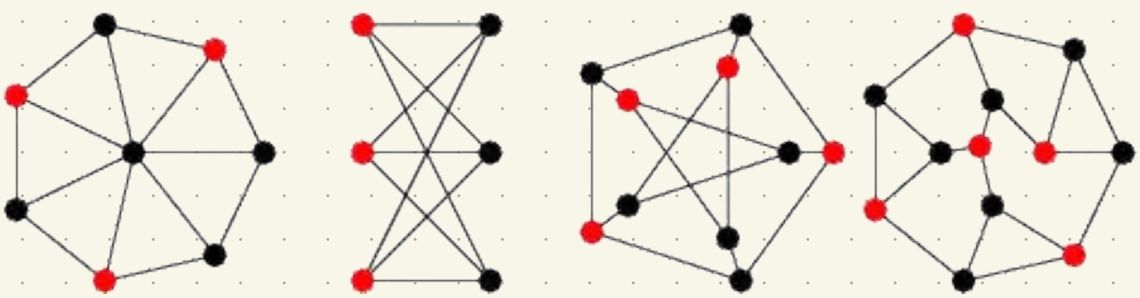
Dynamic Programming on Trees

Independent Set:

(nice preview of graphs)





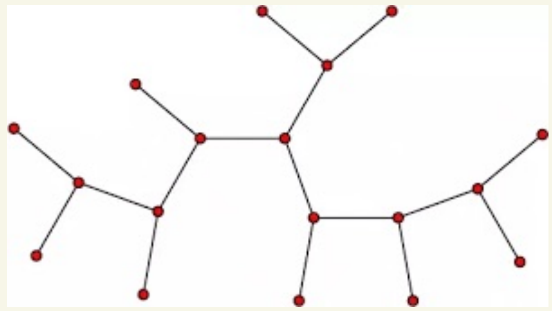


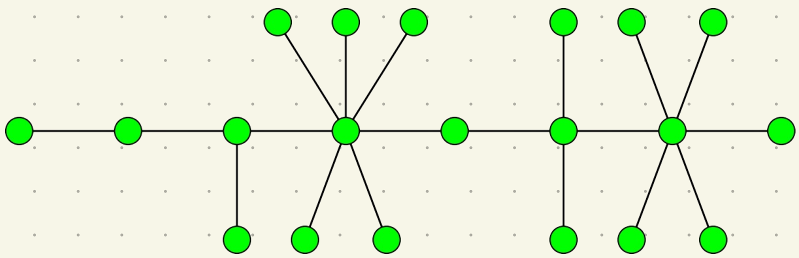
Notoriously hard!

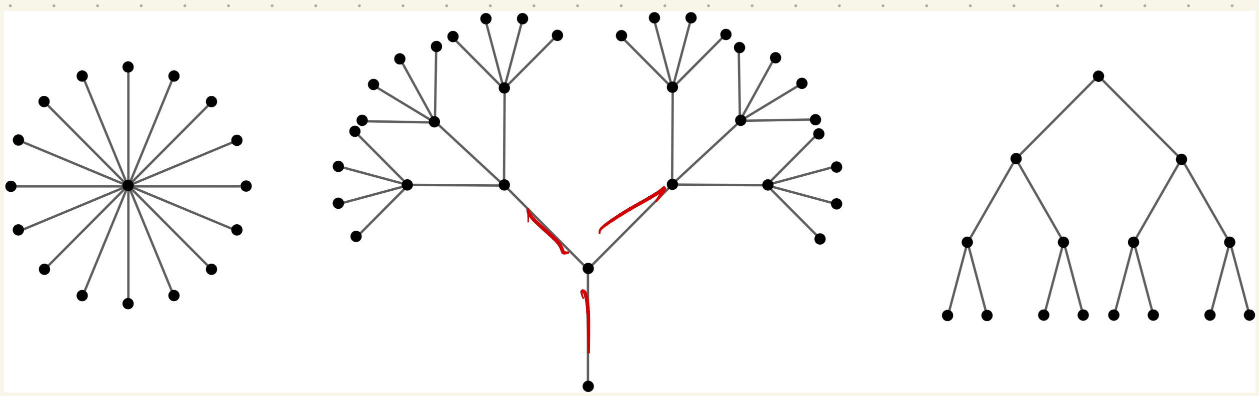
But- can solve on simpler graphs.

Trees:

Not always binary!



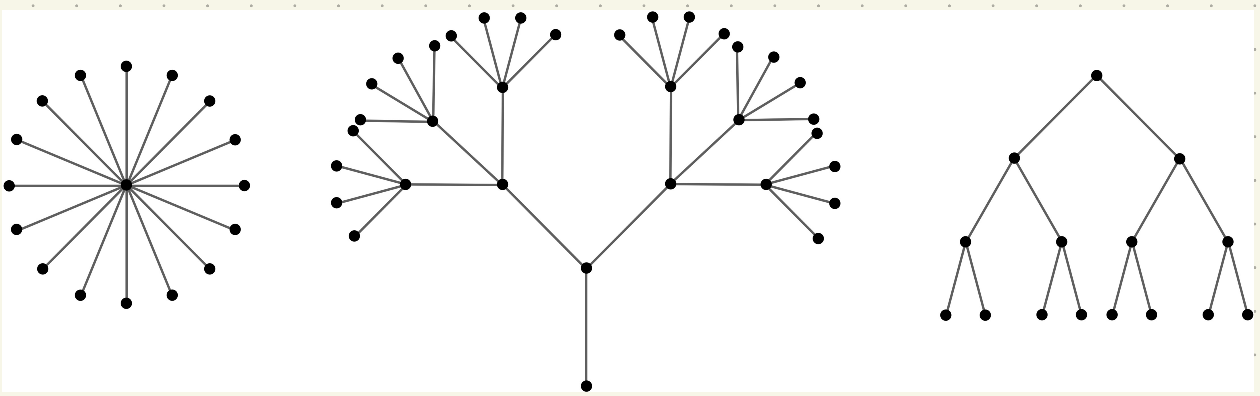




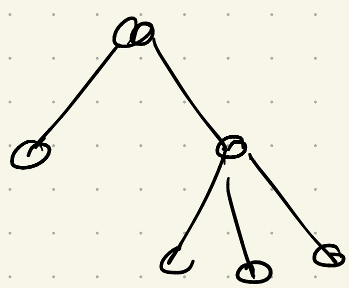
Dfn:

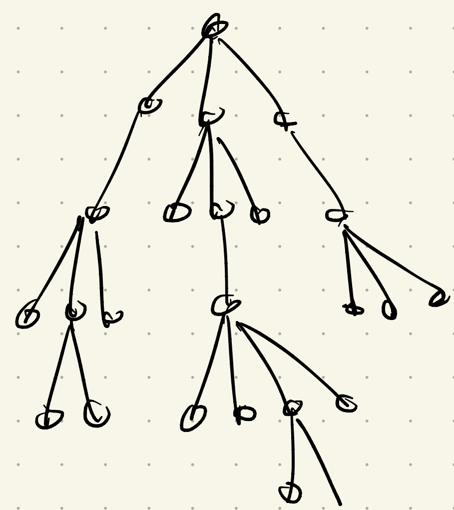
Here, we will "root" the tree.

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!

$$MIS(v) = \begin{cases} & \text{include } v \\ & \text{don't include } v \end{cases}$$

His recurrence (in code):

```
TREEMIS(v):
  skipv ← 0
  for each child w of v
      skipv ← skipv + TREEMIS(w)
  keepv ← 1
  for each grandchild x of v
      keepv ← keepv + x.MIS
  v.MIS ← 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 <u>with</u> v
- Best set without v

Think data structures:

Node v = {

So: Use a tree for the data structure!

```
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.