Algorithms

Dynamic Programming (Part 4)
Trees

Recap

- HW 3 posted
  - due Monday, Sept 28, don't forget groups!
- Reading as usual
- Next week: all about greed
- Afterwards, we switch to the graph chapters.

  Ch. 5+6 are mainly recap:

  module
  - data structures
  - DFS / BFS

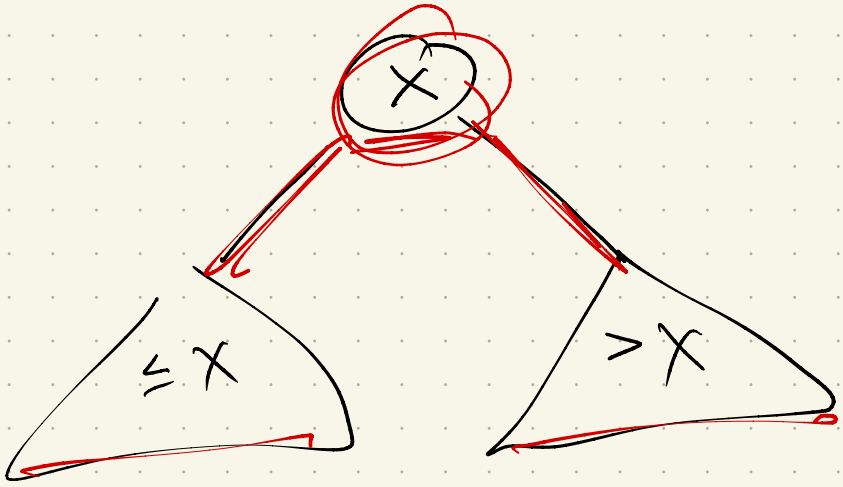
- New zoom link coming soon

Balanced search trees (again)

Recall: input: list of #s

What is the "best" one?

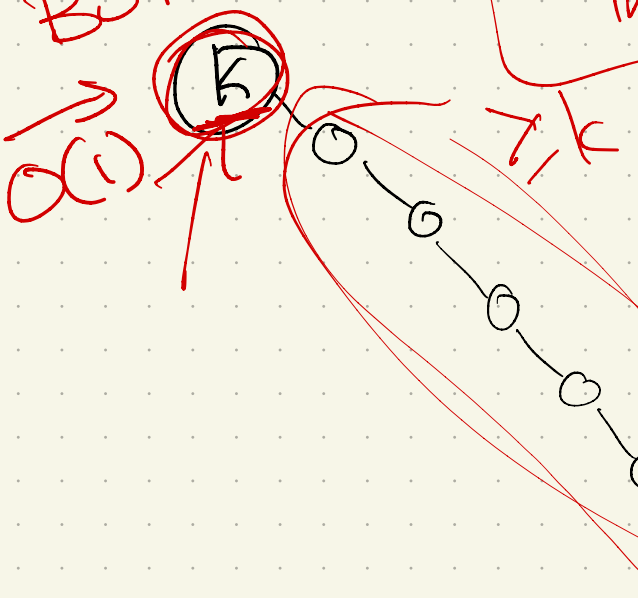
Recap:



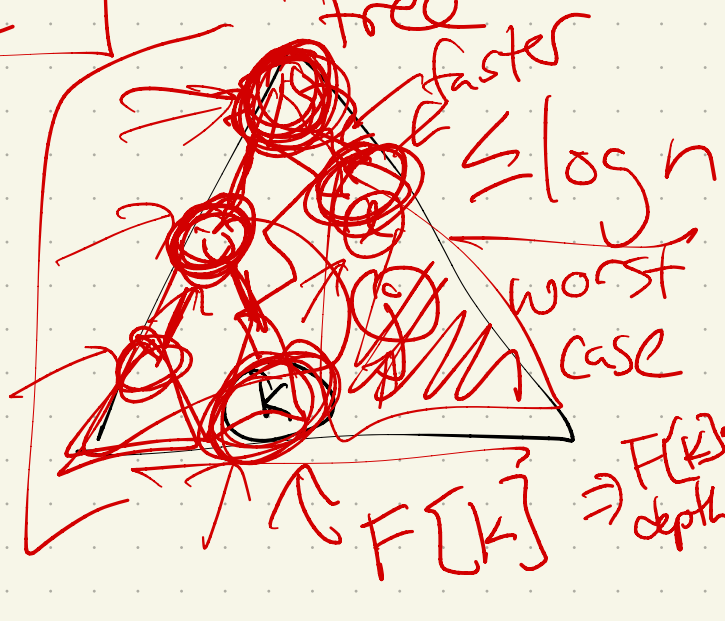
Time to search for k in T

= depth(k) in Tree

BST



Balanced Tree



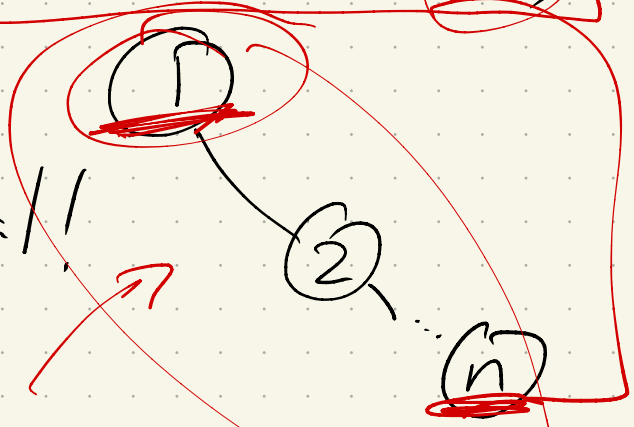
Stepping back even more.

Suppose T holds 1..n & searches are $x_1, \ldots, x_m$

Some searches are "easier":

if $x_1 = x_2 = \cdots = x_m = 1$,

then T =

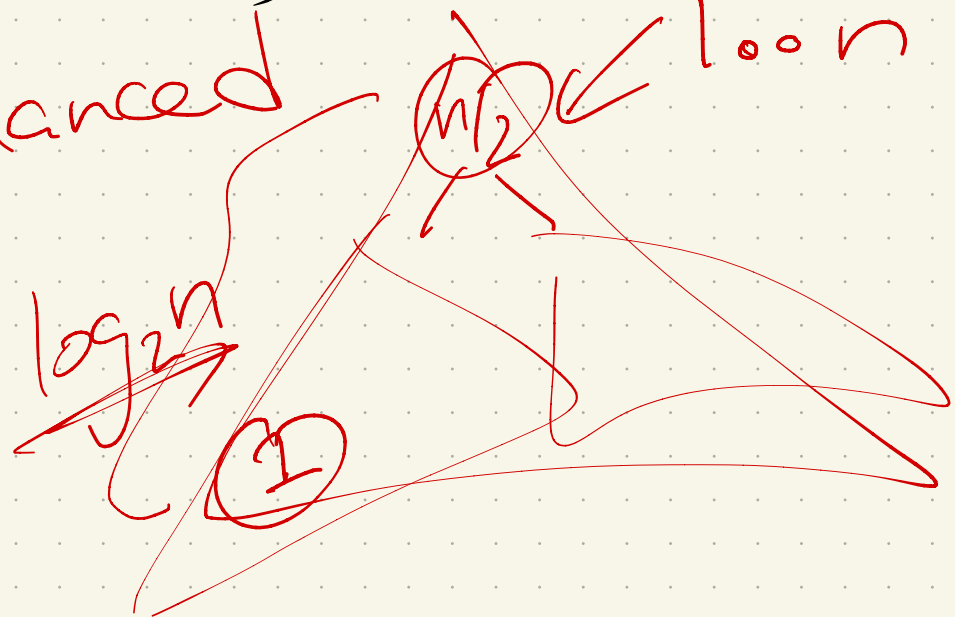


IS optimal!!

Why?

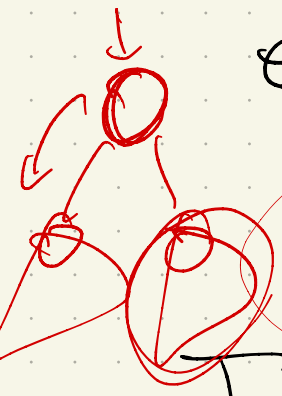
So "best" can change depending on the searches.

Balanced

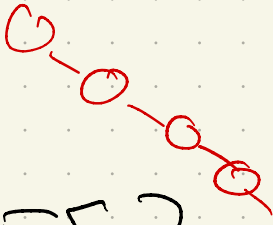


Here: Given $X[1..n]$ ← #s

(assume X is sorted) $F[1..n]$ ← freq. n...1

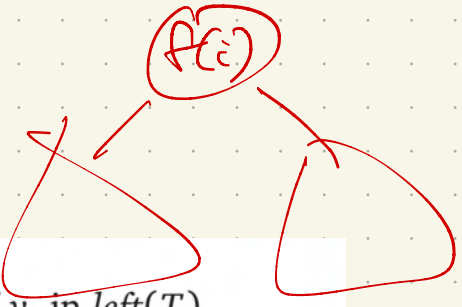


element $X[i]$ will have $F[i]$ searches.



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

Last chapter:



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

root r

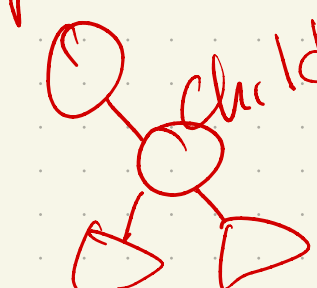
pay at root

Searches root

$\leq r$

$\geq r$

$\Rightarrow$



Pick r:

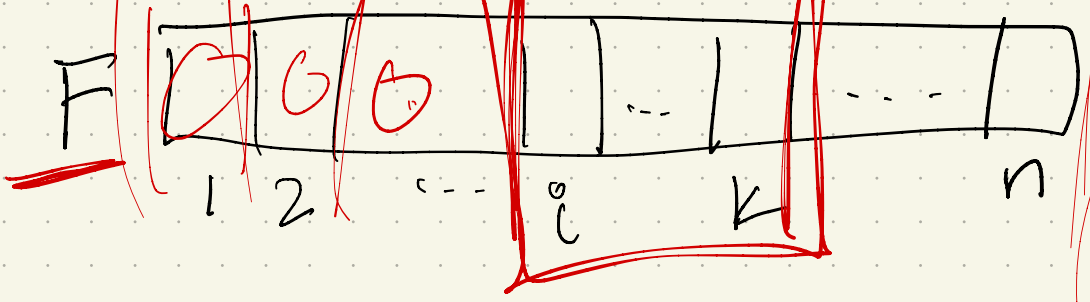
left

right

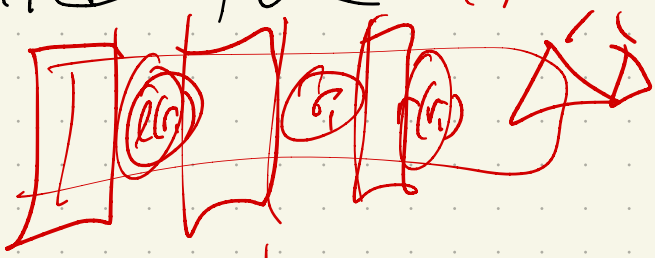
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \{ OptCost(i, r-1) + OptCost(r+1, k) \} & \text{otherwise} \end{cases}$$

recursive

"optimal cost binary tree built on elements i to k of array (w/ freq. F)"



Use this to build the "best" tree:



Choose root.

input: subarray from i to j

Recursively find best left subtree, & best right subtree.

(Note: try all roots in backtracking!)

How to memoize?

$1 \le i \le n$

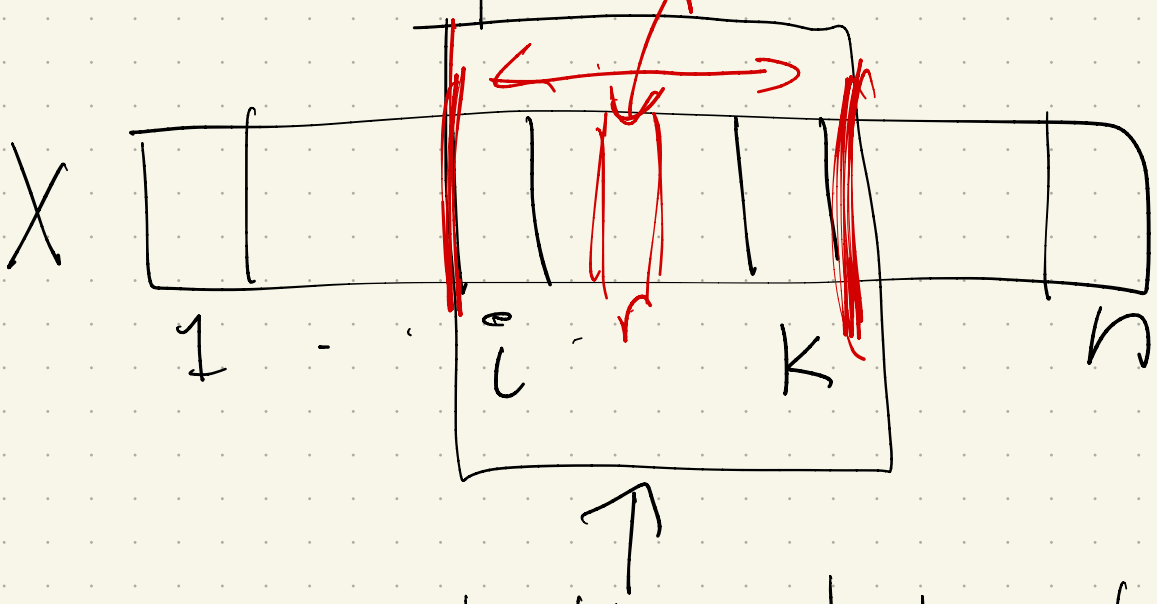
$1 \le k \le n$

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

not $O(i)$ of lookups

Store this somewhere

Remember input: pick root



build best tree here

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

so first precompute & store these sums.

Time/space: ??

space: $n \times n$ array will do

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

Now: $O(n)$

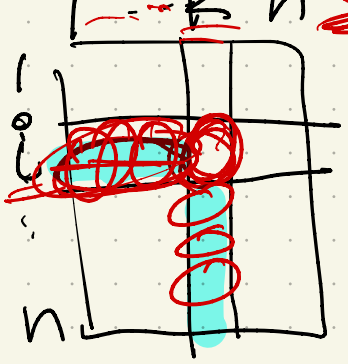
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \{ OptCost(i, r-1) + OptCost(r+1, k) \} & \text{otherwise} \end{cases}$$

$\Downarrow$

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

table lookup: precompute in $O(n^2)$

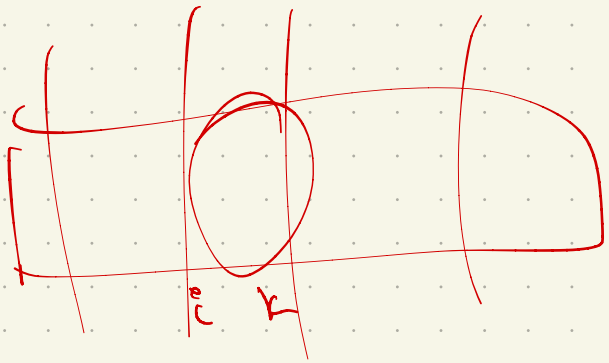
Memoize: $0 \le i \le k \le n$ (letter)



So: 2-d table!

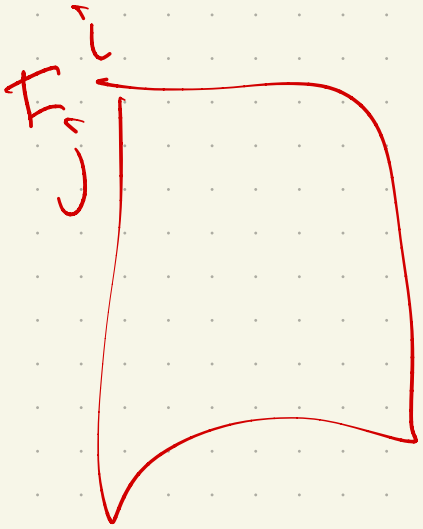
Each $O[i][k]$ needs:

- $F[i][k]$: $O(1)$ lookup
- and min of $O[i][r-1]$ $+ O[r+1][k]$ for $i \le r \le k$

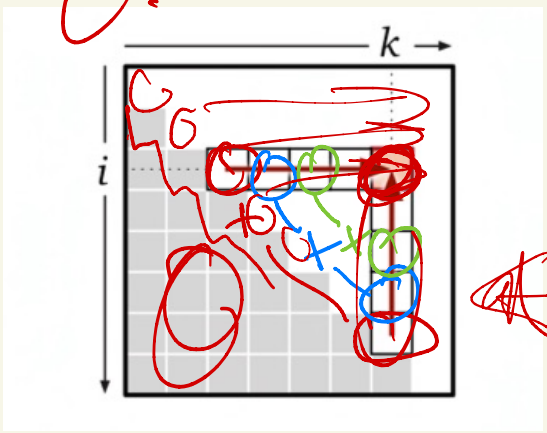


Hts picture (prettier):

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



O:



& take the min

So:

initializing F

```
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]
```

below diagonal = 0

Time: $O(n)$ per table entry $\Rightarrow O(n^3)$

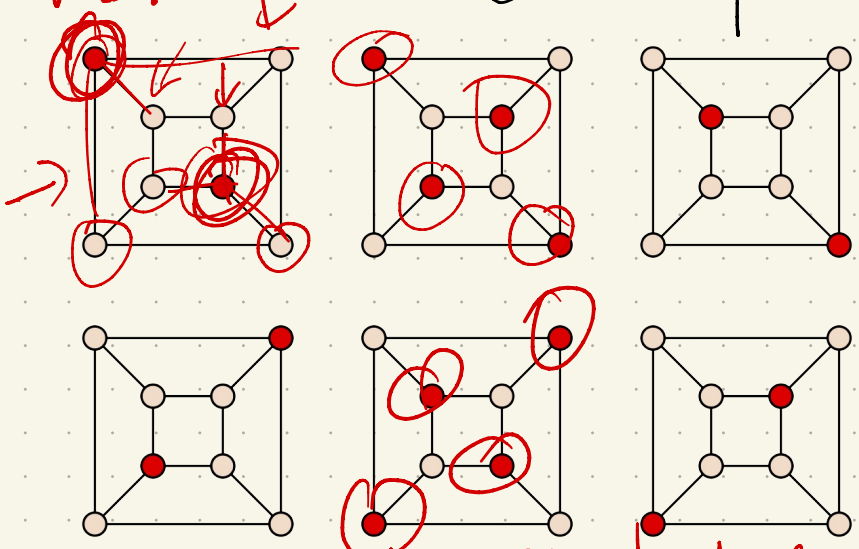
Space: $O(n^2)$

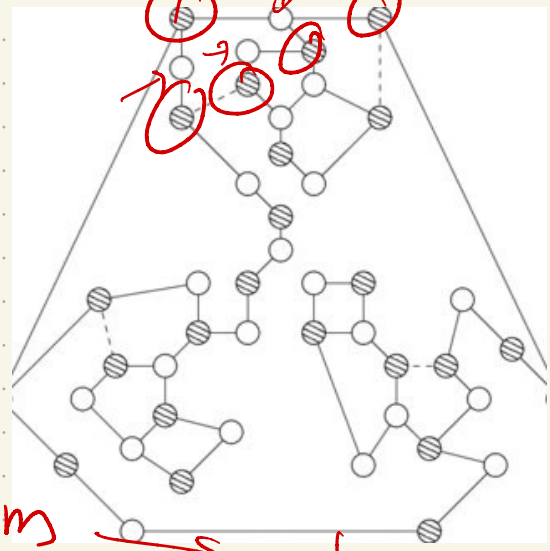
Dynamic Programming on Trees

Independent Set: subset of vertices w/ 0 edges

(nice preview of graphs)

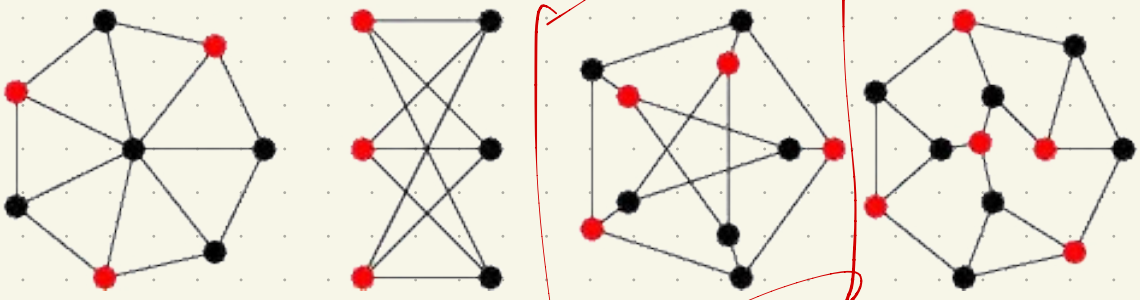
maximal





maximal & maximum

not unique



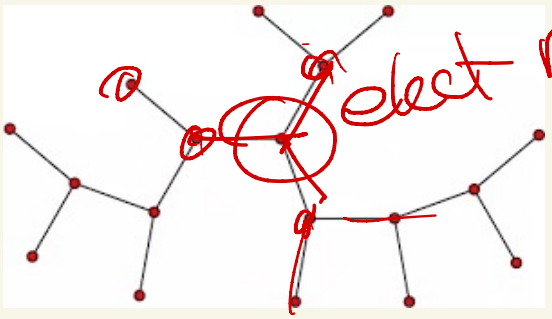
Notoriously hard!

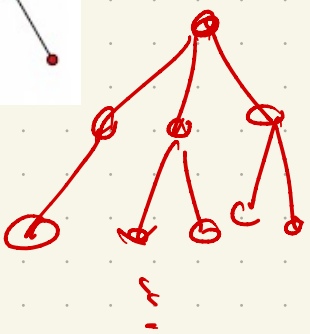
But- can solve on simpler graphs.

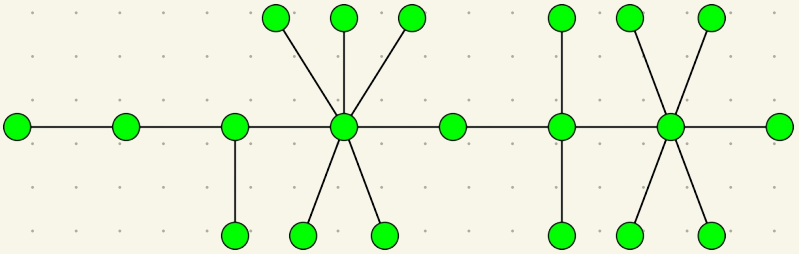
Trees: near + dear to CS majors

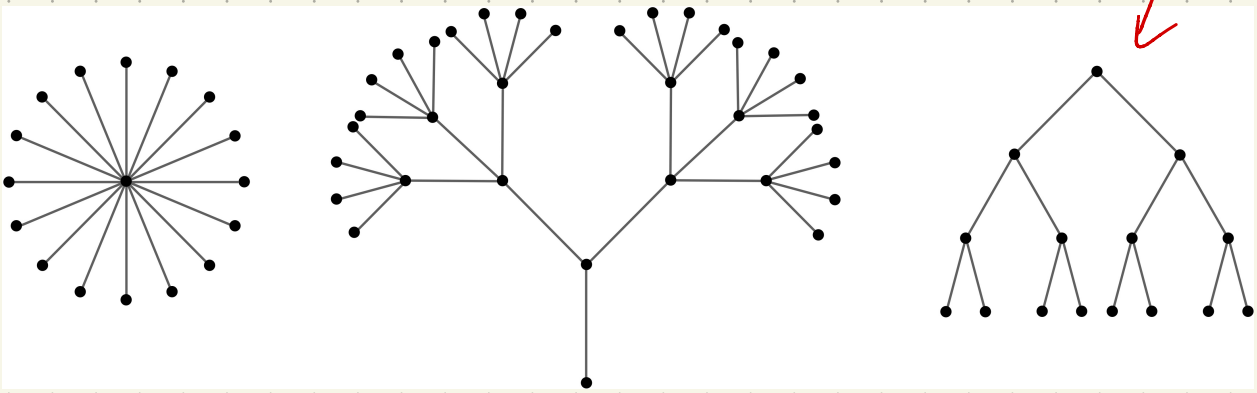
Not always binary!

not always rooted





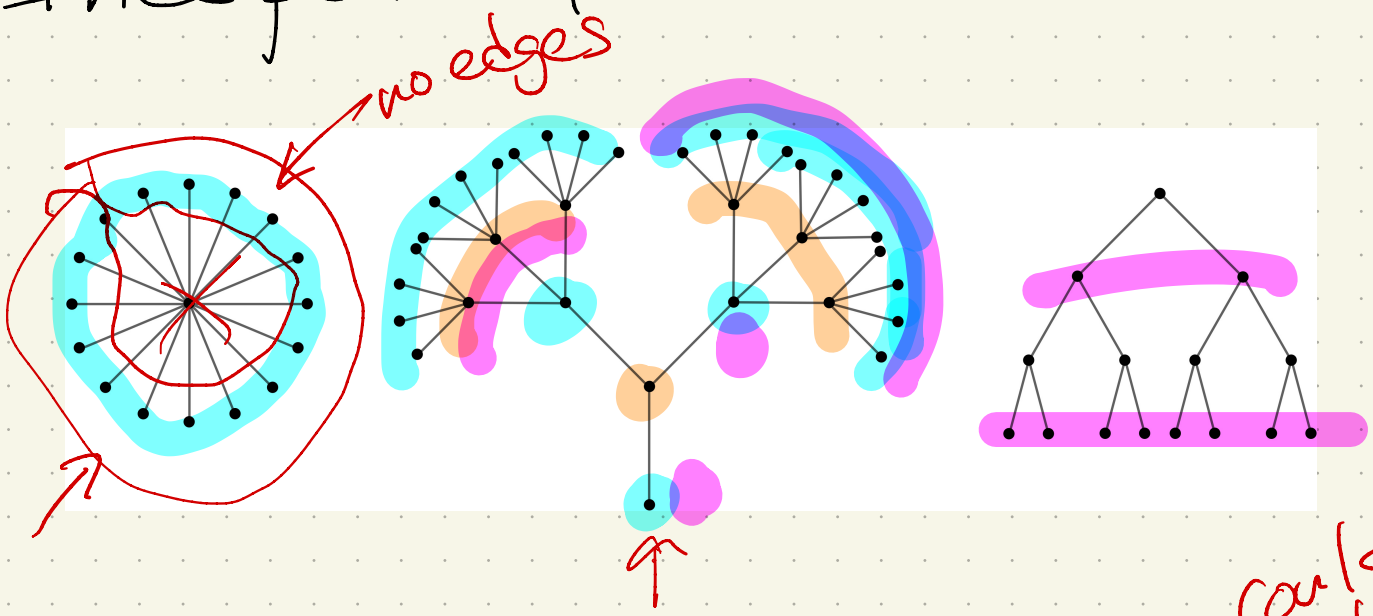




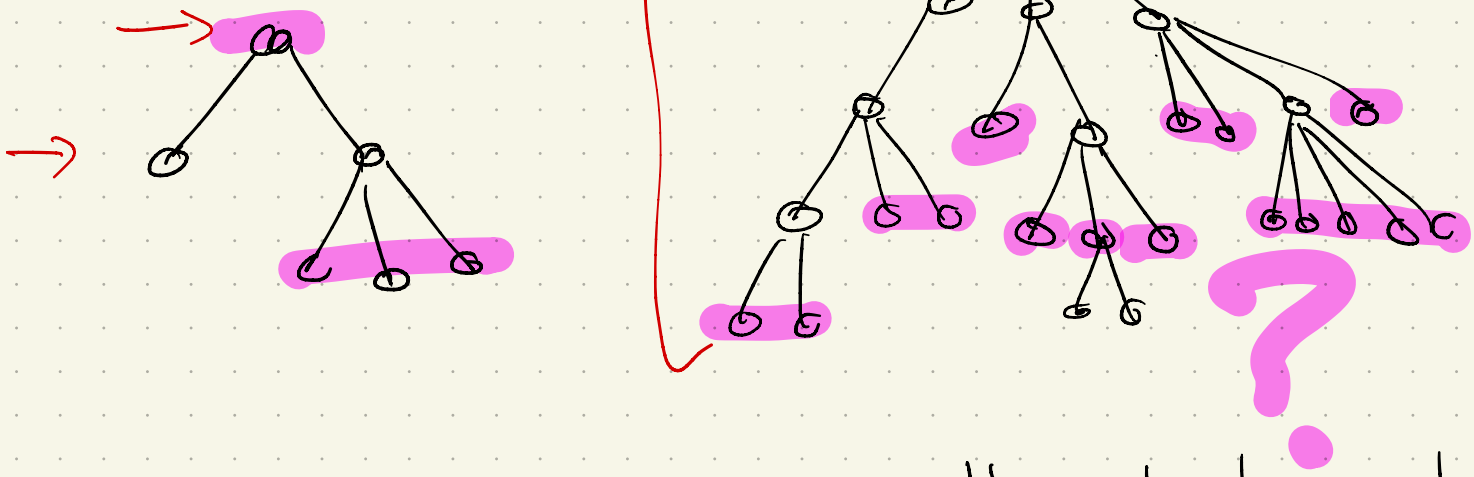
Dfn: graph w/ no cycles.

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.

Back tracking!

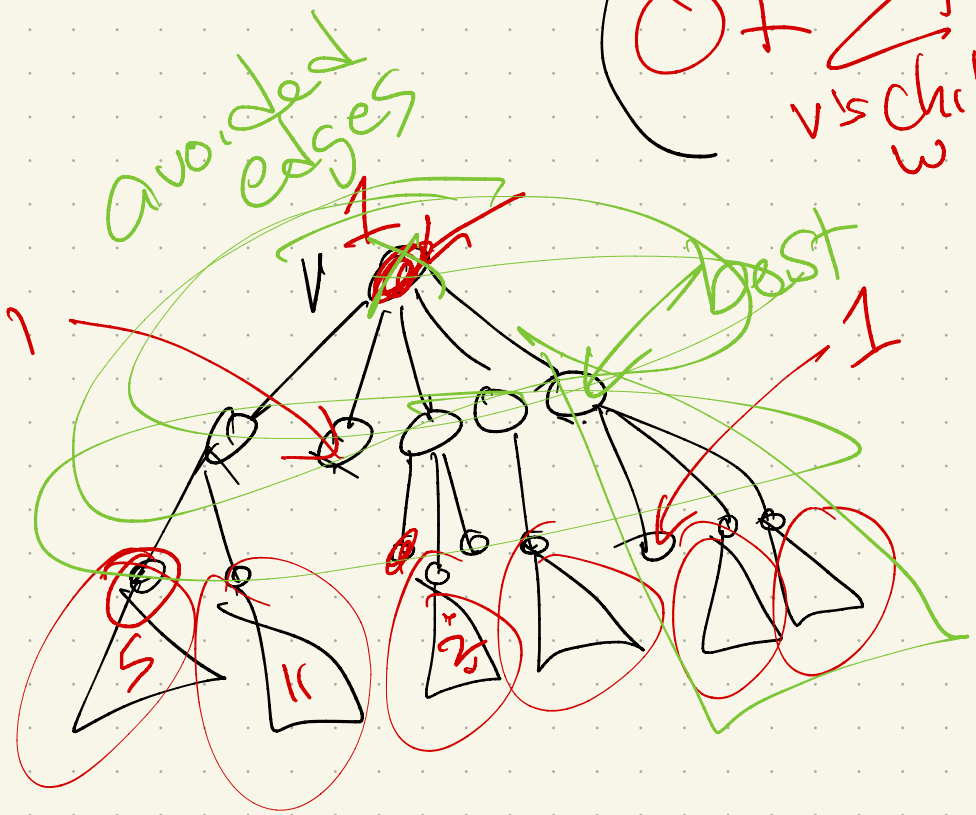
$$MIS(v) = \begin{cases} 1 + \sum_{w} MIS(w) & \text{include } v \\ 0 + \sum_{w} MIS(w) & \text{don't include } v \end{cases}$$

w: v's grandchildren

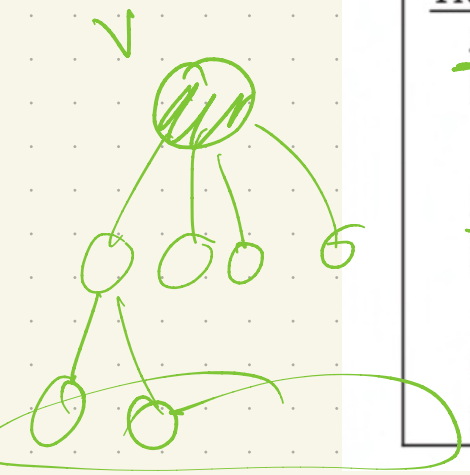
include v → no children

w: v's children

don't include v → could include children



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 + TREEMIS(x)
  v.MIS ← max{keepv, skipv}
  return v.MIS
```

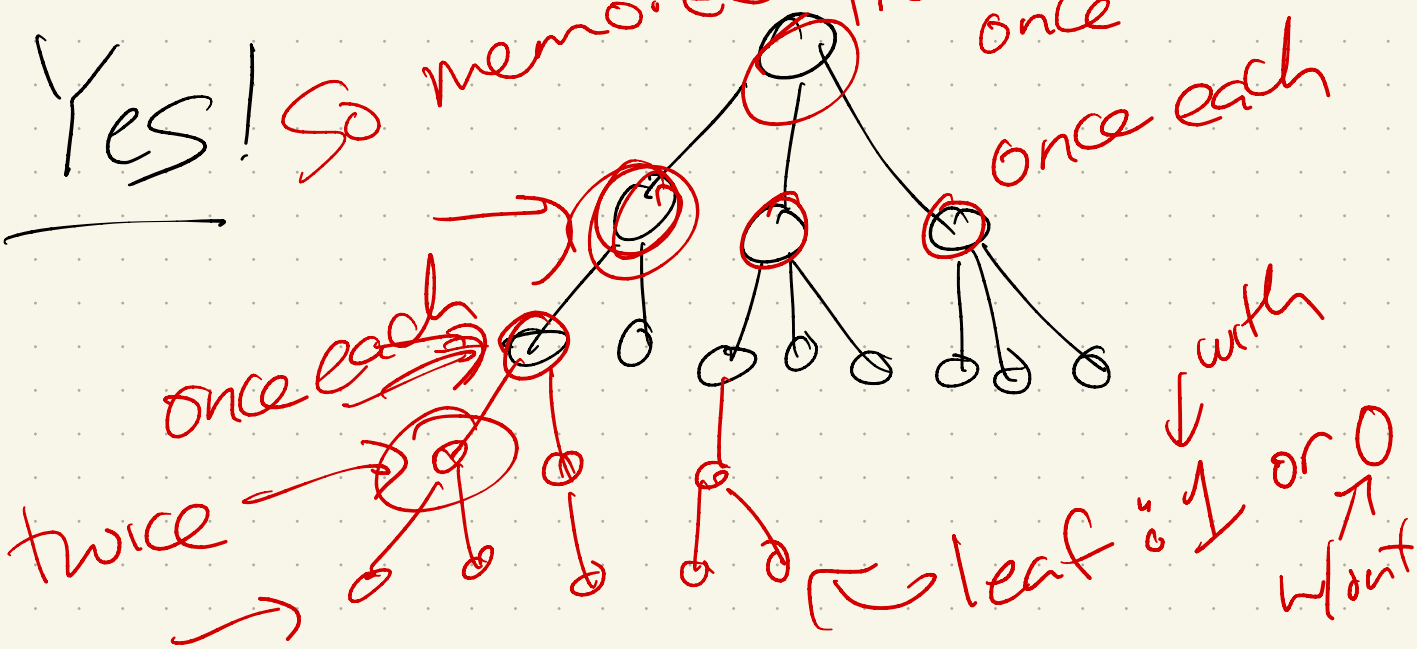
recurse on children

with v

w/out v

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

Yes! So memoize TREEMIS(r)



How to memoize:

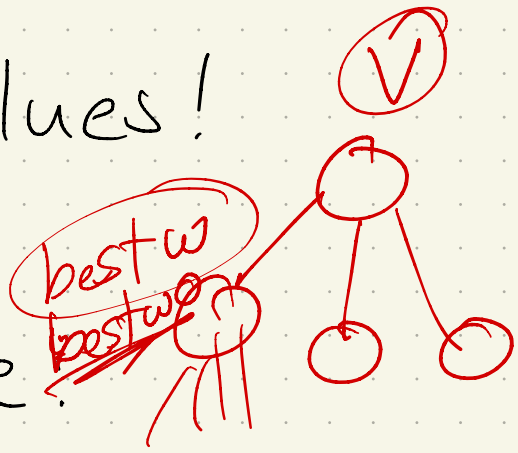
Well, for each node, need the best ind set in that subtree.

Even better – 2 values!
(same big-O)

For each v, store:

- Best set with v
- Best set without v

↳ post order traversal



Think data structures:

extend

Node v = { data, left, right, parent, height / balance factor, etc. }

add 2 fields with #s without

V. MIS

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}
```

recursion

update myself

fill in my answer

fields in DS node

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

→ $O(n)$ on any binary tree

Space: added 2 fields per node → $O(n)$