Algorithms (Fall 2020)

Dynamic Pro. (pt3)

- Edit distance

HW2 - grading ends today

HW1 - not graded yet.

Reading as usual

HW3 - coming soon

↳ written

Edit distance:

HUGE in bioinformatics!

One of the basic tools in sequence alignment.

(I have a book with an entire chapter on how to optimize.)

Here: insert, delete, "mutate" +1

How to begin? (Recursively!)

n

	1	2	3	4	5	6	7	8	9	10
A:	A	L	G	O	R	I	T	M		
B:	A	L	T	R	U	I	S	T	I	C

m

Start at end, & ask "obvious" question:

M & C could be "aligned" → +1

or not: M deleted, or C deleted

try them all!

1

$i-1$ i

ALGORIT M

ALTRUISTI C

$j-1$ j

mutate

$M \neq C$

+1

2

A

B

$i-1$

ALGORIT M

ALTRUISTI C

j

M is deleted

+1

3

A

B

i

ALGORIT M

ALTRUISTI

C

$j-1$ j

insert C

+1

$A[i] \neq B[j]$:
match, delete, or insert

Input: $A[1..n]$

$B[1..m]$

$$\text{Edit}(A[1..i], B[1..j])$$

case of they match

$$= \min \begin{cases} \text{match: if } A[i] = B[j], & \text{Edit}(i-1, j-1) \\ \text{if } A[i] \neq B[j] & \\ \text{mutate: } 1 + \text{Edit}(i-1, j-1) & \\ \text{insert: } 1 + \text{Edit}(j-1, i) & \\ \text{delete: } 1 + \text{Edit}(i-1, j) & \end{cases}$$

1, 2, 3, 4

+ Base cases:

"Backtracking recursion"

$$\left.\begin{aligned} A[0] + B[0] &= 0 \\ A[1] + B[0] &= 1 \\ A[0] + B[1] &= 1 \end{aligned}\right\}$$

this way:

base cases

delete

insert

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$

+1 or +0

mutate

$A[1 \ldots i]$, $\underset{j=0}{B[\,]} = \phi$ → $+i$

$+j$: $A[\,] = \phi$, $B[1 \ldots j]$

memoize!

So: what's our "memory" data structure?

$A: 0 \le i \le n$

$B: 0 \le j \le m$

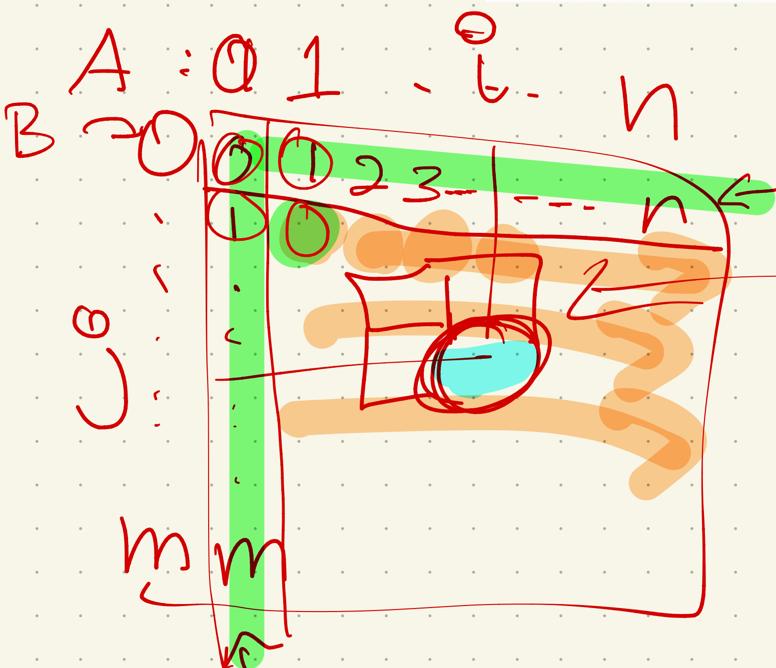
↳ requires storing 1 value for each (i,j)

⇒ $n \times m$ table

Then, our algorithm:

- start w/ base case (row & column)
- Fill in:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$



If these 3 are computed already, then O(1) to fill in cell (i,j)

either row by row or column by column order

Result:

```
EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          ins ← Edit[i, j − 1] + 1
          del ← Edit[i − 1, j] + 1
          if A[i] = B[j]
              rep ← Edit[i − 1, j − 1]
          else
              rep ← Edit[i − 1, j − 1] + 1
          Edit[i, j] ← min {ins, del, rep}

  return Edit[m, n]
```

base case

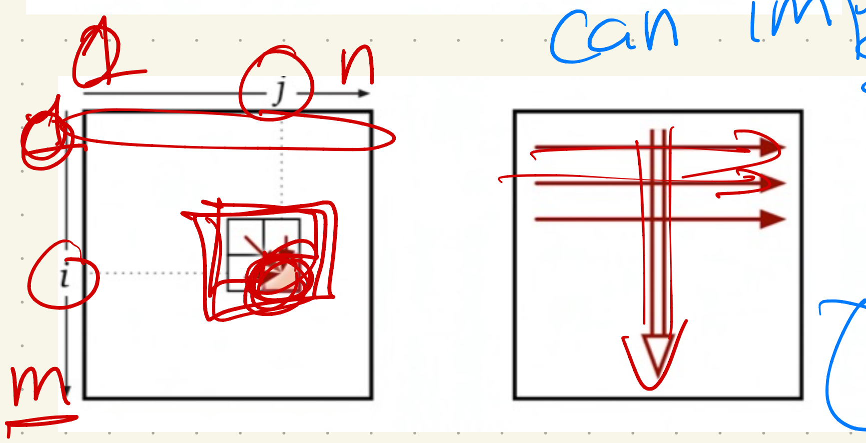
fill top to bottom

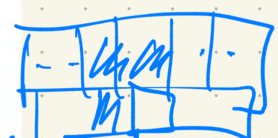
fill a row

gets uglier

can improve space!

Picture:





use 2 rows only

Time: nested for loops

$n \times m$ cells: $O(1)$ each

(3 lookups, a min, some +1's)

→ $O(mn)$

Space: $n \times m$ table

$= O(mn)$ (size of each value)

Subset sum (revisited)

Key takeaway (I think):
Sometimes, our backtracking recurrences can be memoized.
(Note: Sometimes, they can't!
Think n queens.)

Recall:

Given a set $X[1..n]$ of numbers + a target T, find a subset of X whose sum is $= T$.

Ch2 solution

p. 77

78?

```
⟨⟨Does any subset of X sum to T?⟩⟩
SubsetSum(X, T):
  if T = 0
      return True
  else if T < 0 or X = ∅
      return False
  else
      x ← any element of X
      with ← SubsetSum(X \ {x}, T − x)     ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X \ {x}, T)         ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

```
⟨⟨Does any subset of X[1..i] sum to T?⟩⟩
SubsetSum(X, i, T):
  if T = 0
      return True
  else if T < 0 or i = 0
      return False
  else
      with ← SubsetSum(X, i − 1, T − X[i])   ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X, i − 1, T)          ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

same thing, but x = last thing in array X

The recursion:

(Note: same thing as code!!)

rewrite

$$SS(i,t) = \begin{cases} \text{True} & \text{if } t = 0 \\ \text{False} & \text{if } t < 0 \text{ or } i > n \\ \underbrace{SS(i+1,t)}_{\text{wout}} \vee \underbrace{SS(i+1,t-X[i])}_{\text{with}} & \text{otherwise} \end{cases}$$

hint: storage

How to memoize?

2d array: 1 entry per i + t

$SS(i, t) = T$ or F

$0 \le i \le n$

item in X

$0 \le t \le T$

possible sum

So: another 2-d table!

To decide:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \lor SS(i+1, t - X[i]) & \text{otherwise} \end{cases}$$

if $t<0$, False

fill in this cell

look at these 2 cells.

one note: if $t - X[i] < 0$, wasting time! Equivalent to:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \lor SS(i+1, t - X[i]) & \text{otherwise} \end{cases}$$

$\lor F$

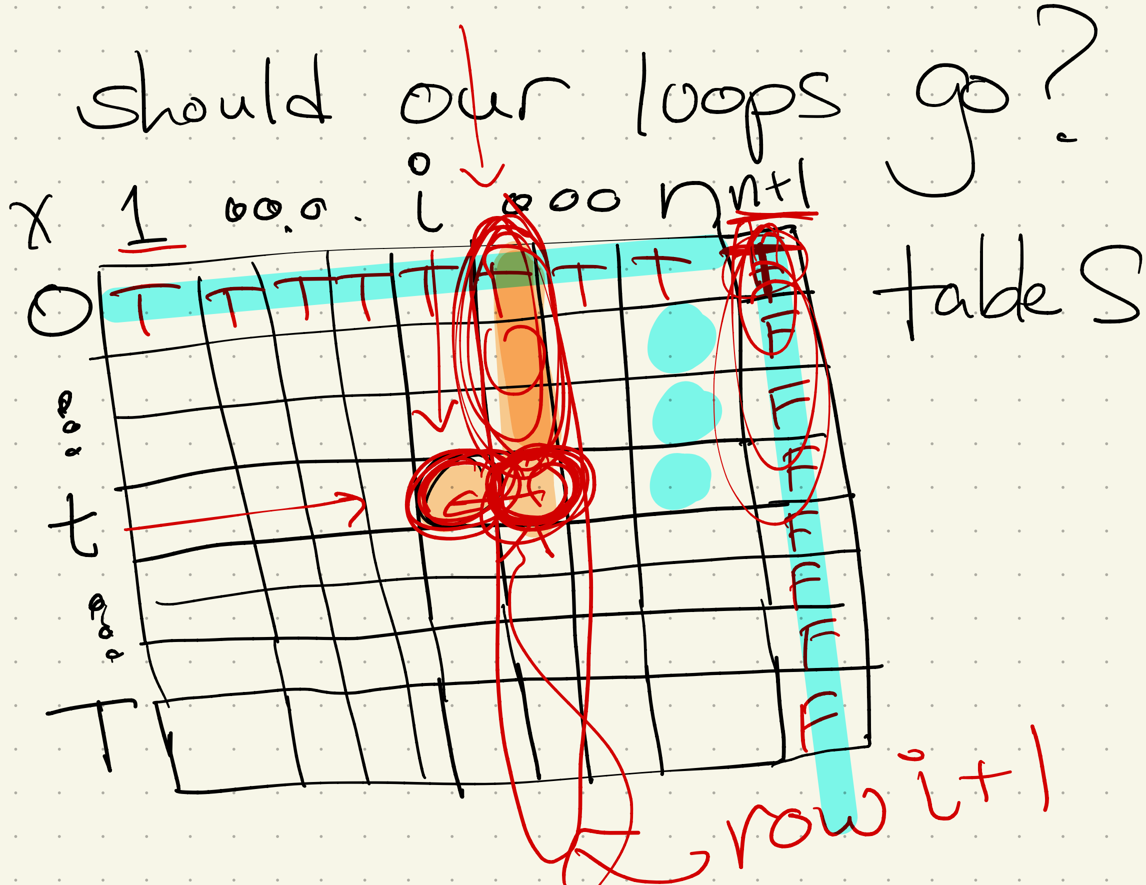
$\Rightarrow t - X[i] < 0$

Now - need to code this:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \vee SS(i+1, t - X[i]) & \text{otherwise} \end{cases}$$

$< t$

How should our loops go?



Fill: check if $t < X[i]$

↳ in org. input

↳ then $SS(i,t)$ ← $S(i+1,t)$

If not: also need $S(i+1, t - X[i])$

His code:

```
FastSubsetSum(X[1..n], T):
    S[n+1,0] ← True
    for t ← 1 to T
        S[n+1,t] ← False

    for i ← n downto 1
        S[i,0] = True
        for t ← 1 to X[i]−1
            S[i,t] ← S[i+1,t]             ⟨⟨Avoid the case t < 0⟩⟩
        for t ← X[i] to T
            S[i,t] ← S[i+1,t] ∨ S[i+1, t−X[i]]

    return S[1,T]
```

first do easy

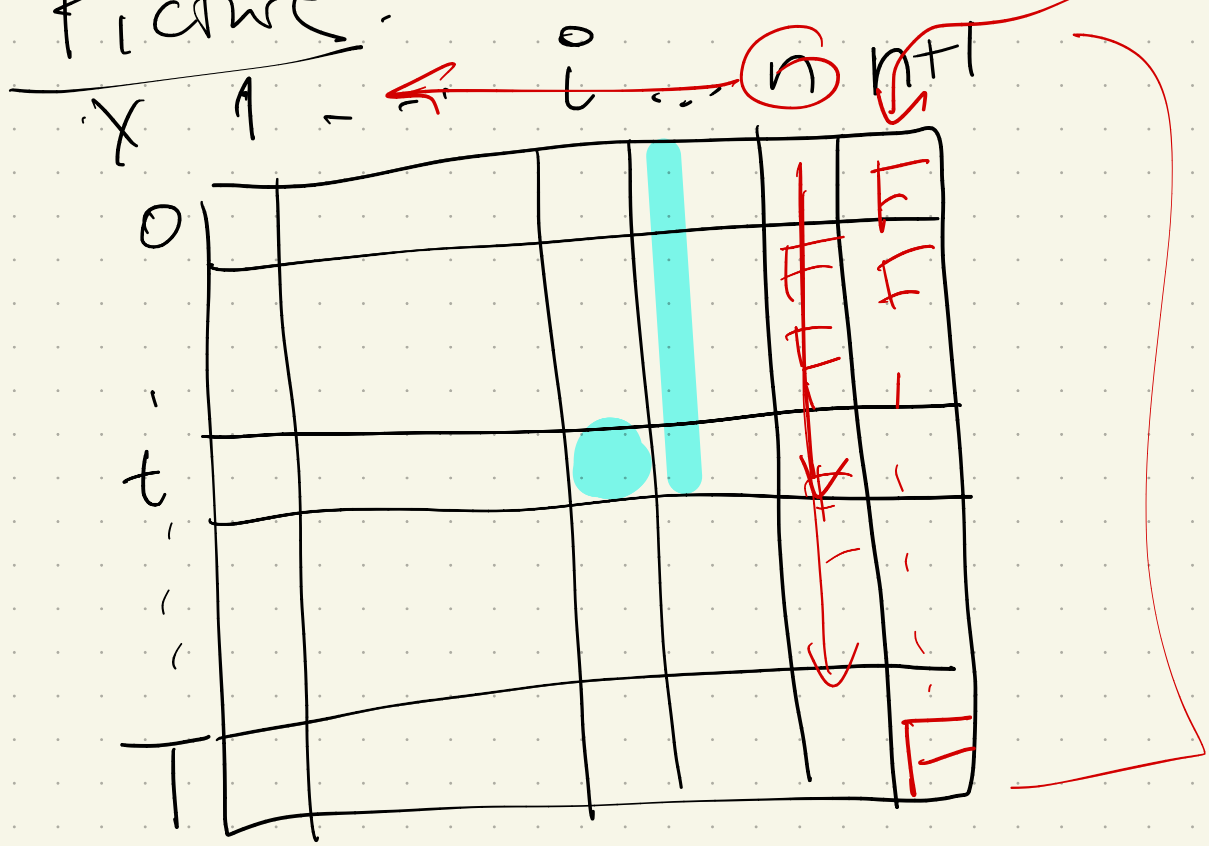
t

t

$O(t)$

n

Picture:



time & space: $O(nT)$

Wait, though:

Do we need this table?!

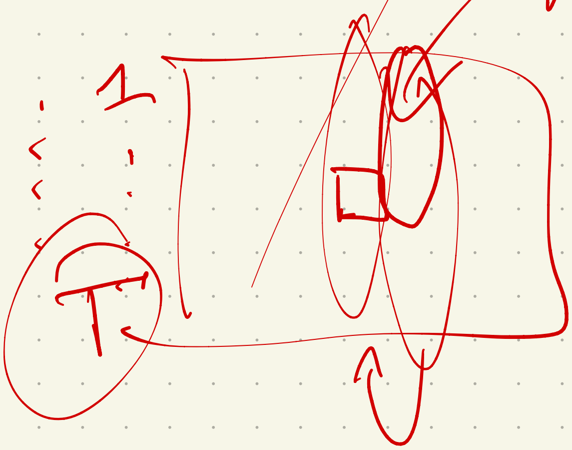
- It has a row for every value $1 \ldots T$.

But – T can be huge!

Ex: list of 1000 #s, all ≤ 9000

But T can be in the millions!

(& most of those $1 \ldots T$ are impossible to hit.)



most of these are empty!

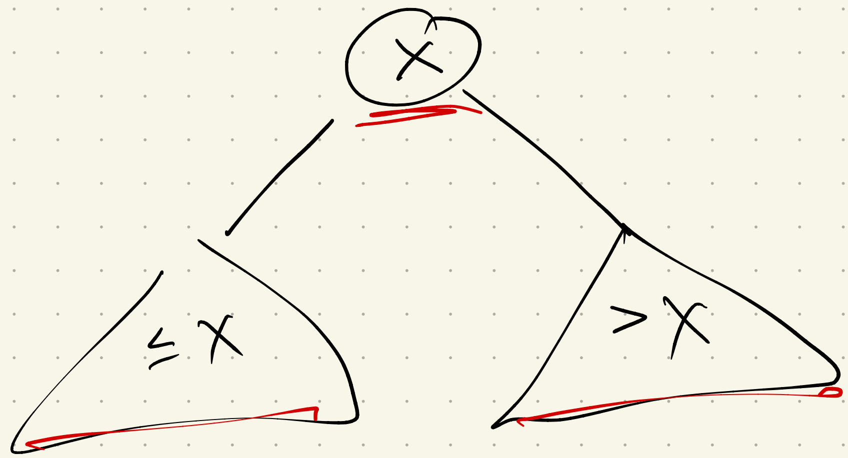
2^{1000} might be better than $1000 \times$ millions

Balanced search trees (again)

Recall:

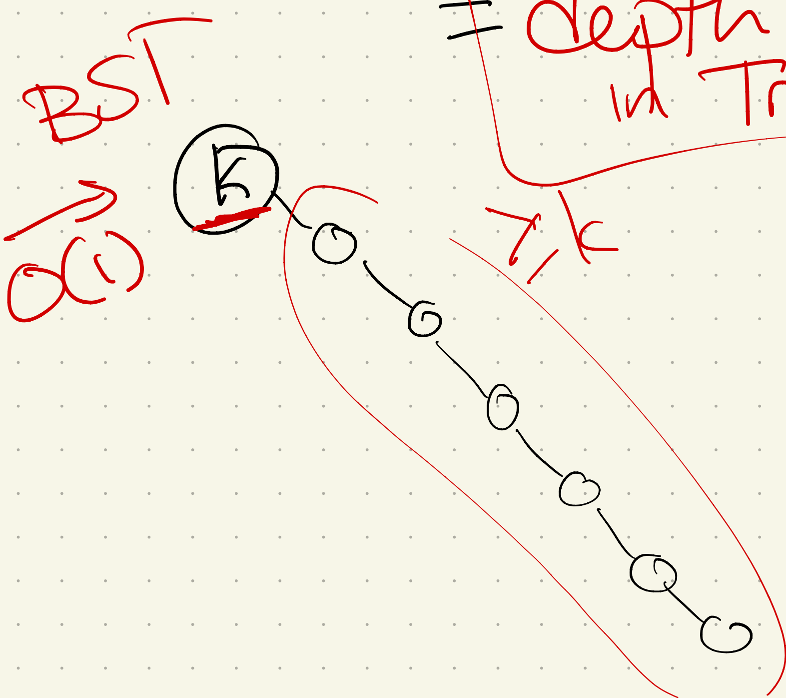
What is the "best" one?

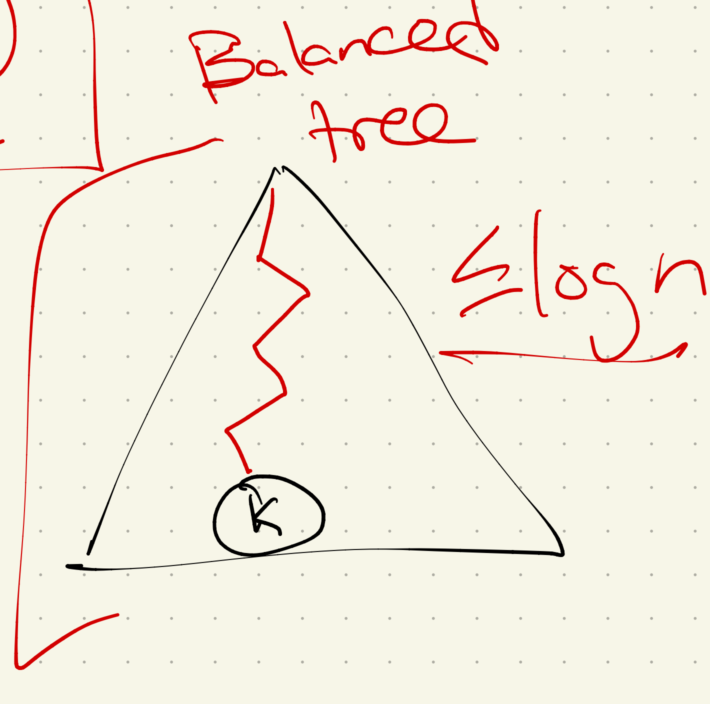
Recap:



Time to search for k in T

$= \boxed{\text{depth}(k) \text{ in Tree}}$





Stepping back even more.

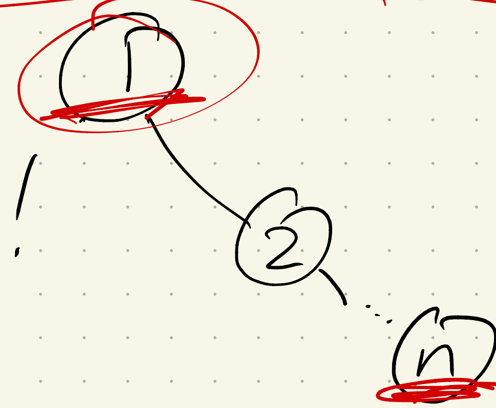
Suppose T holds $1..n$

& searches are $x_1, \dots, x_m$

Some searches are "easier":

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

then $T =$

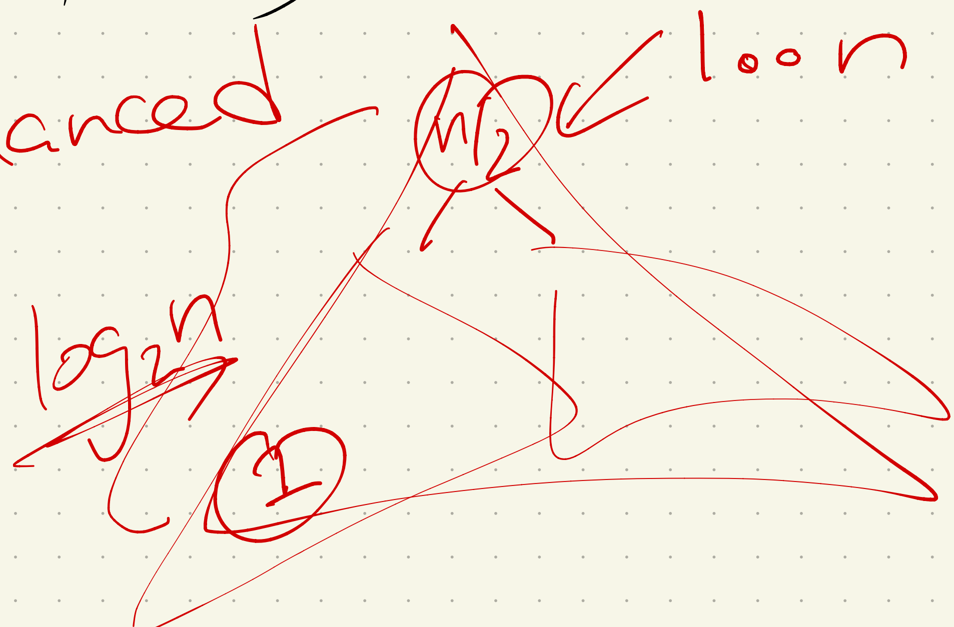


IS optimal!

Why?

So "best" can change depending on the searches.

Balanced

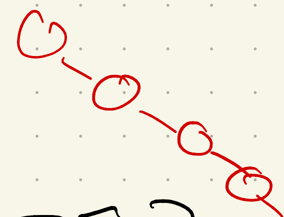


Here: given $X[1..n]$ ← #s (sorted)

$F[1..n]$ ← freq.

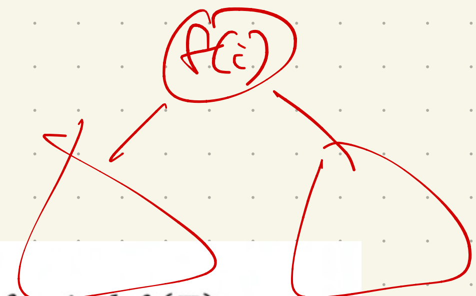
$n \ldots 1$

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



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

Last chapter:



root r

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

$\Longrightarrow$

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

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

Use this to build the "best" tree:

Choose root.

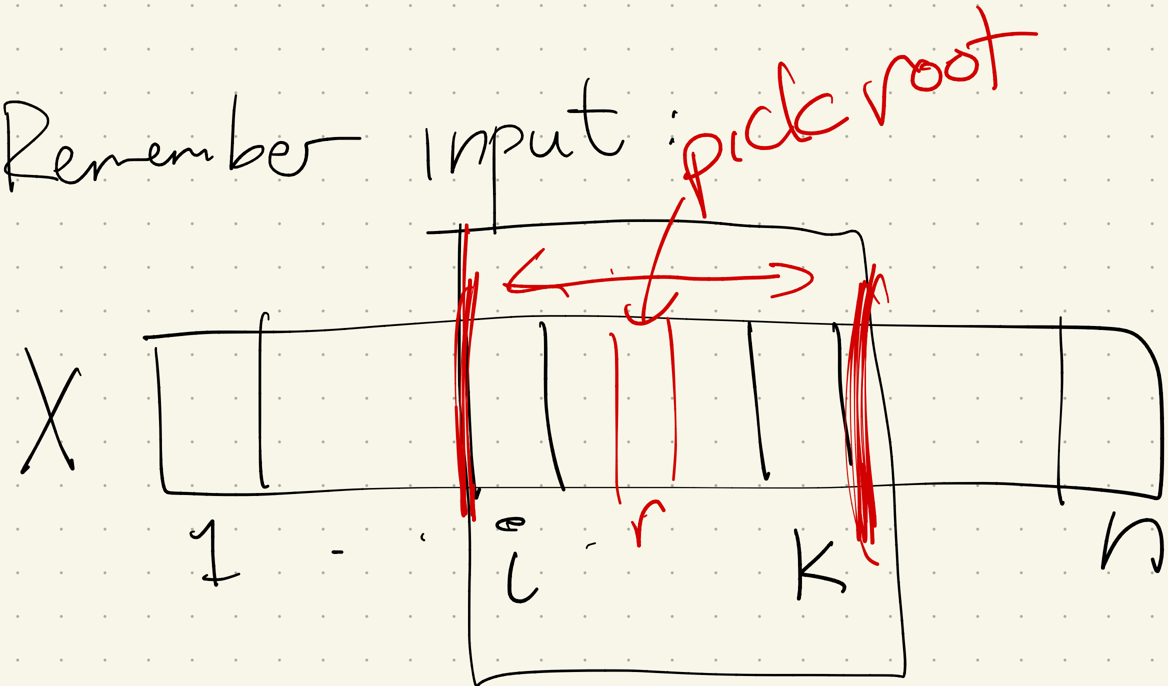
Recursively find best left subtree, + best right subtree.

(Note: try all roots in backtracking!)

How to memoize?

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

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:

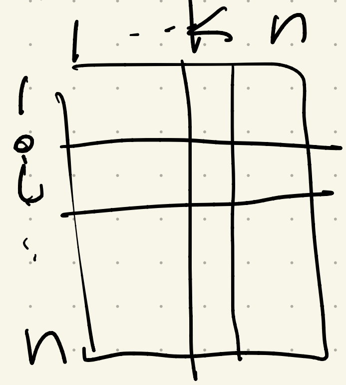
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \begin{Bmatrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{Bmatrix} & \text{otherwise} \end{cases}$$

$\Downarrow$

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

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

So: 2d table!

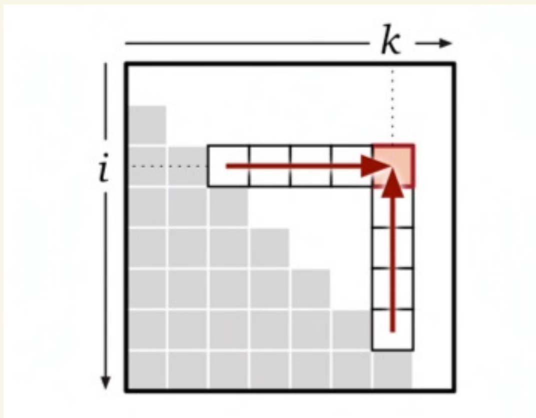


Each $O[i][k]$ needs:

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

His picture (prettier):

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



So:

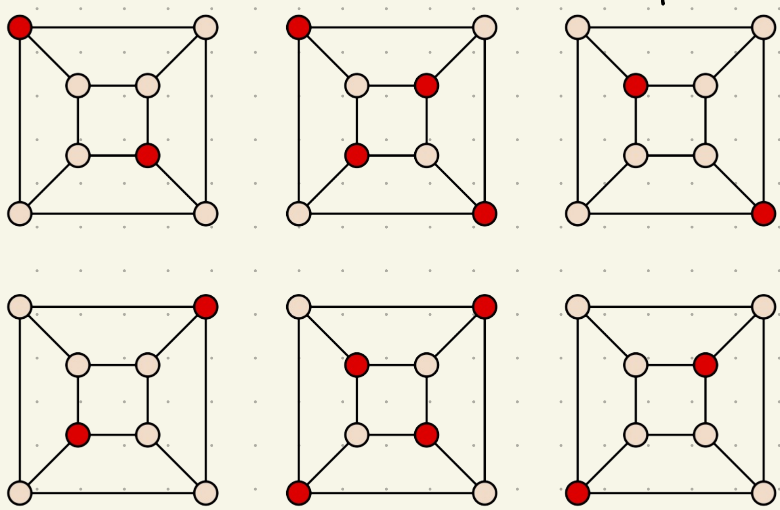
```
OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n+1
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d        ⟨⟨...or whatever⟩⟩
          ComputeOptCost(i, i+d)
  return OptCost[1, n]
```

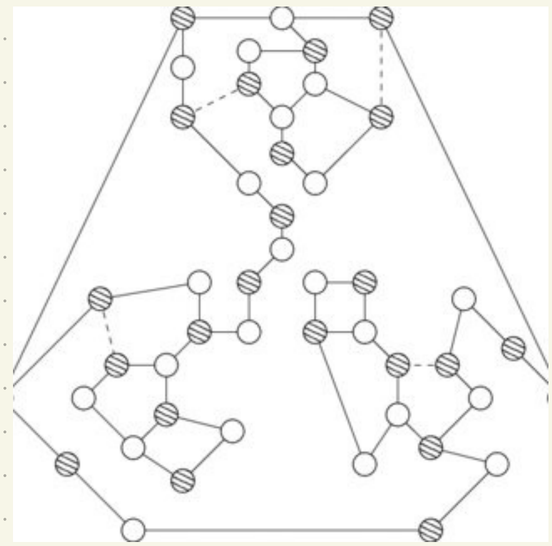
Time:

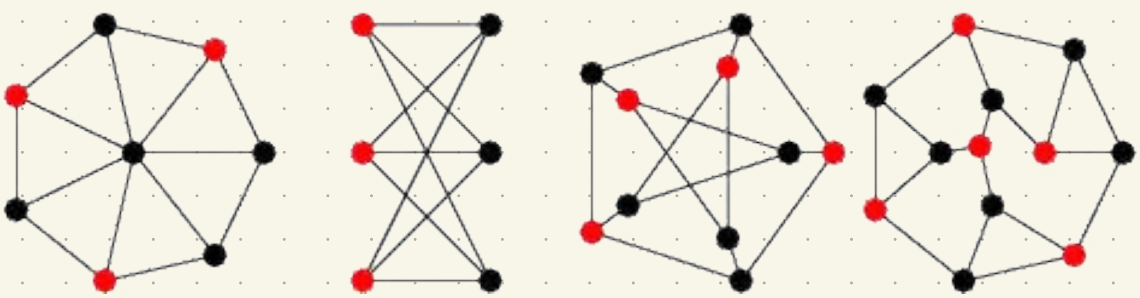
Space:

Dynamic Programming on Trees

Independent Set:
(nice preview of graphs)



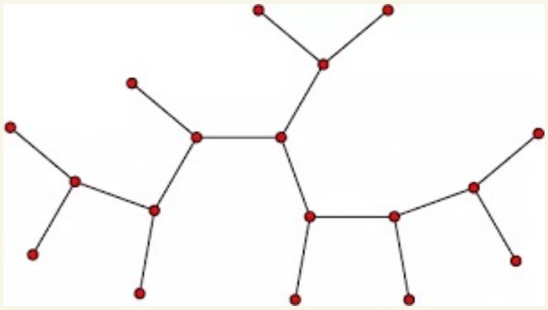


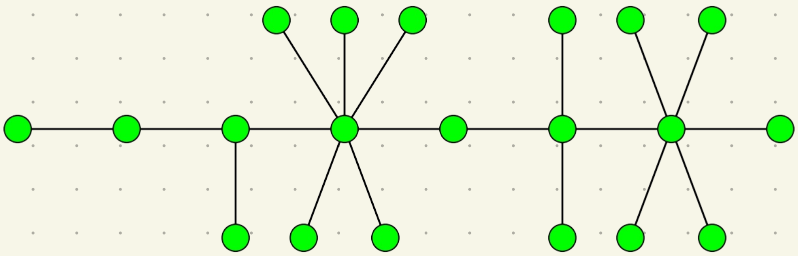


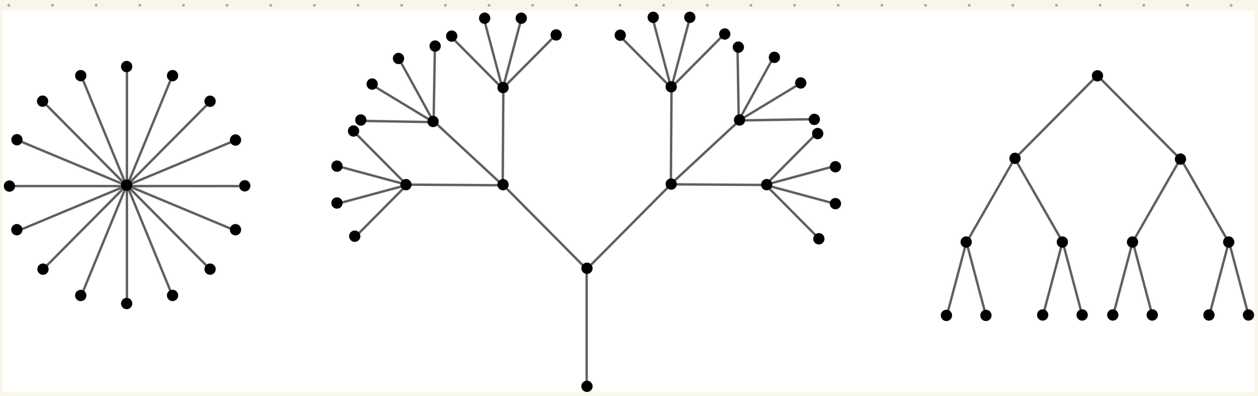
Notoriously hard!
But - can solve on simpler graphs.

Trees:

Not always binary!



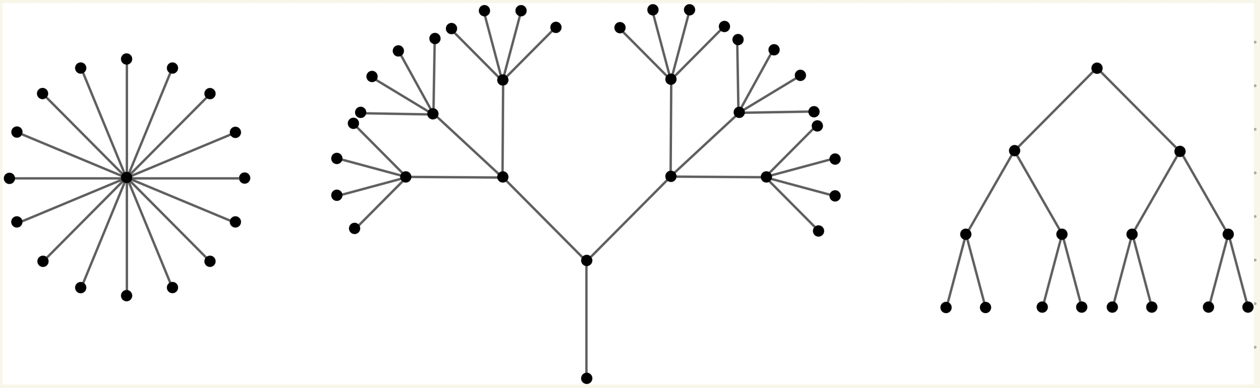




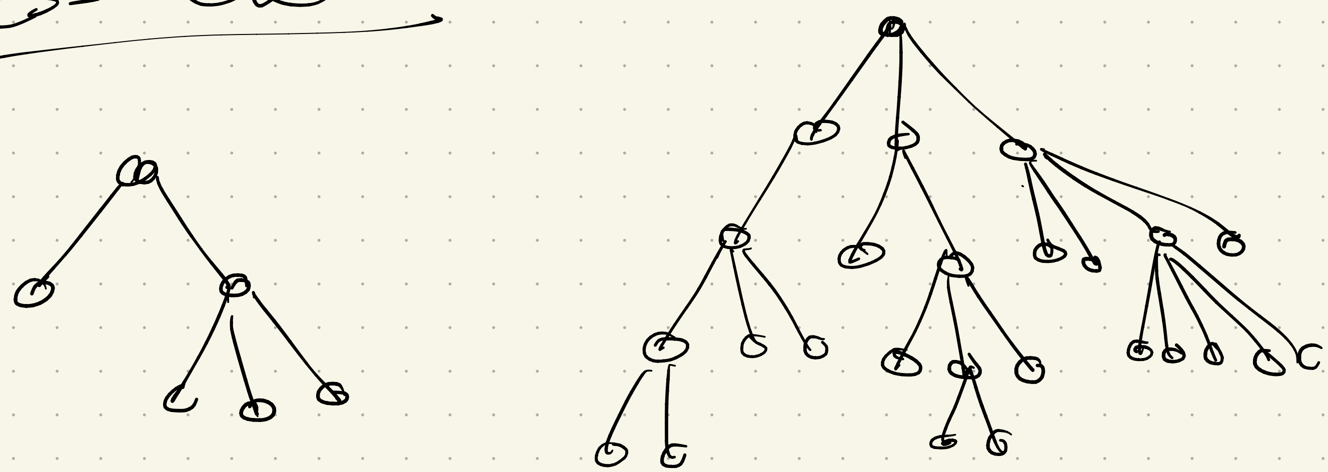
Dfn:

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!

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