Algorithms - Spring '25

Dynamic Pro:
Edit dist
SubsetSum
Trees

Recap

- Back tracking HW - due today
- Next HW - up later today, due Wed, Feb. 19 (cover DP)
- Readings set for Friday and next week (through Ch 5)

Note: a couple of gaps

- HW #1a: subset of X
  1b: can just return weight

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.)

Also: spell checkers, word prediction, etc.

How to begin? (Recursively!)

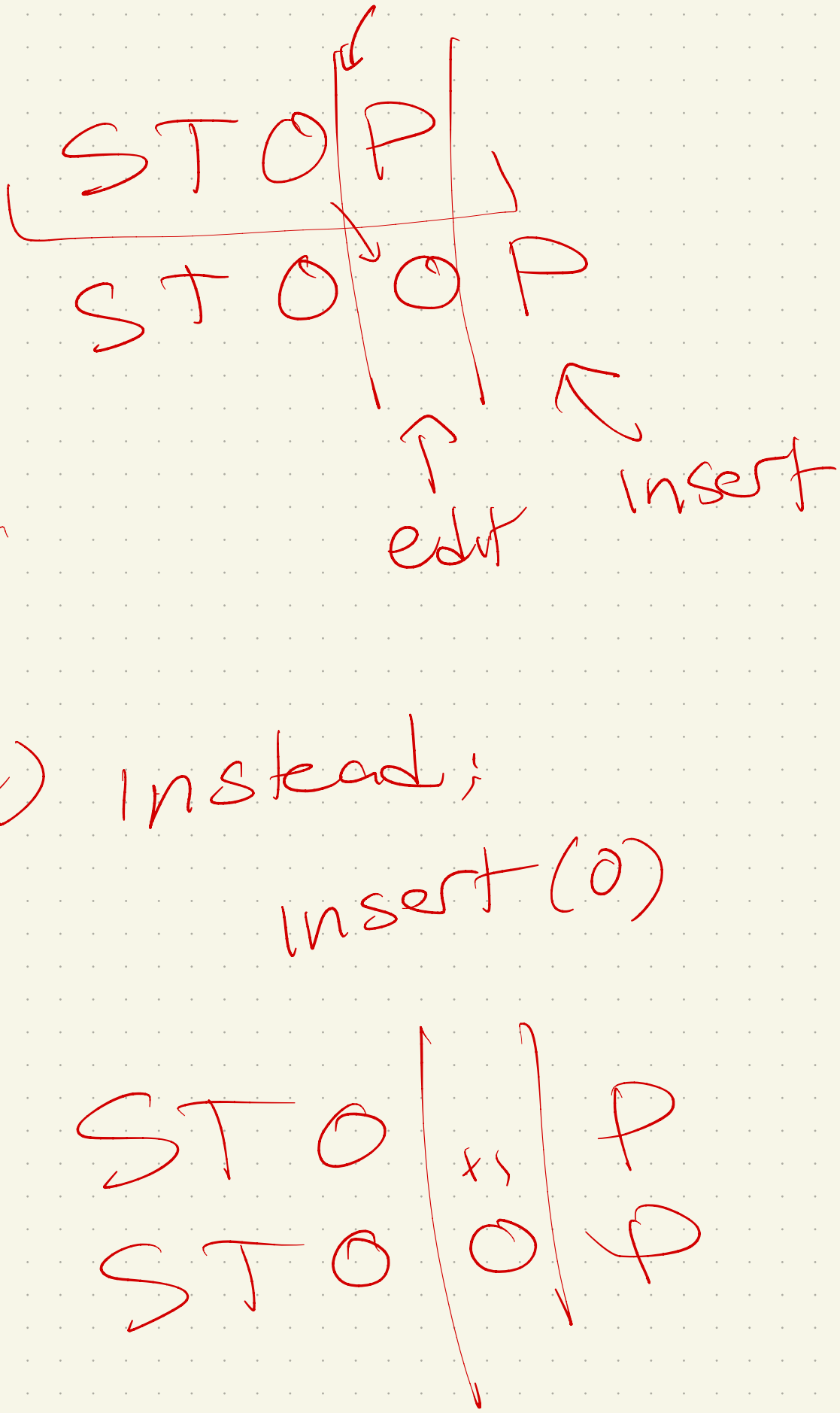
ALGORITM

ALTRUISTIC

Start at end, + ask "obvious" question:

insert, delete, edit

try them all!


STOP
STOOP
edit
insert
instead;
insert(0)
STO
++
P
STOO
P

Let's try:

A: ALGORITM

B: ALTRUISTIC

Start at end:

Align: edit A[m] to B[n]

↖ down 2 letters (both A+B shorter)

(or +1 if ≠)

M | +1 C

Insert

↑ B is shorter ← M

(A is not)

M | = ← | +1 C

Delete:

A shorter

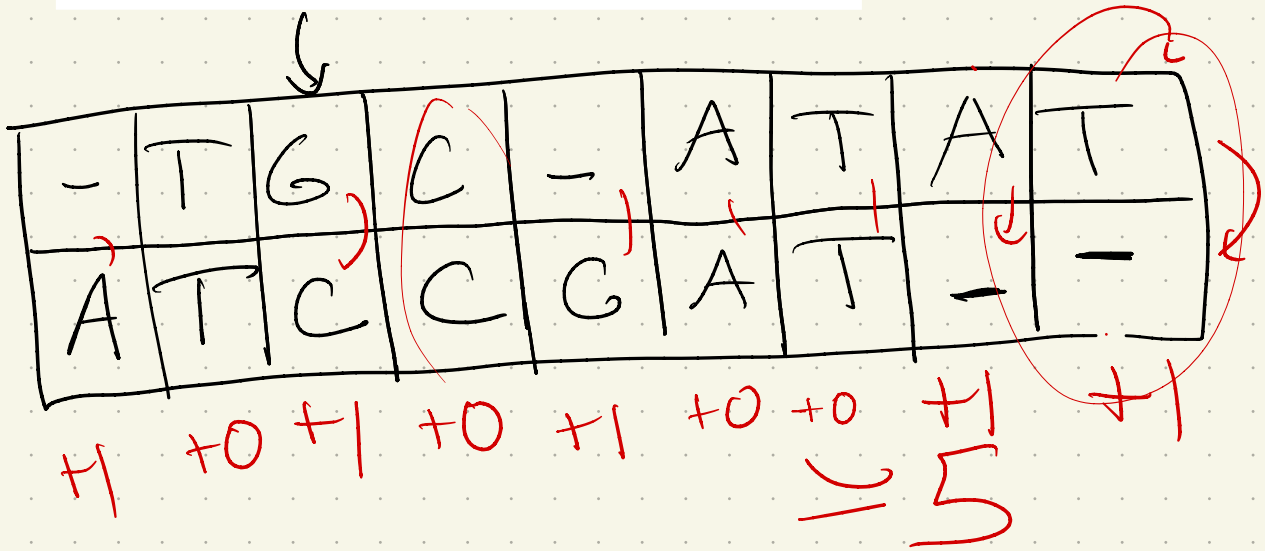
(B is not) — — C

M | = ↓

Example: TGCATAT to ATCCGAT

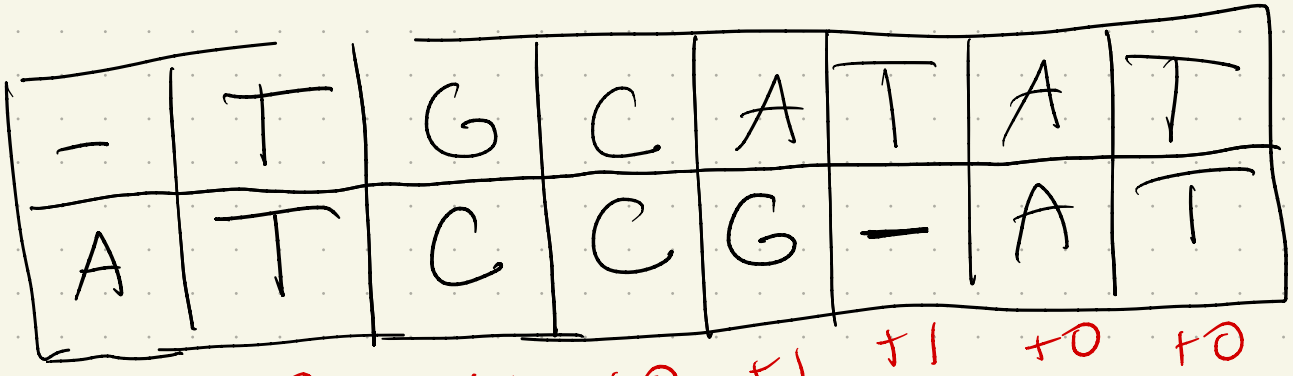
TGCATAT
↓ delete last T
TGCATA
↓ delete last A
TGCAT
↓ insert A at the front
ATGCAT
↓ substitute C for G in the third position
ATCCAT
↓ insert a G before the last A
ATCCGAT

TGCATAT
↓ insert A at the front
ATGCATAT
↓ delete T in the sixth position
ATGCAAT
↓ substitute G for A in the fifth position
ATGCGAT
↓ substitute C for G in the third position
ATCCGAT



+1 +0 +1 +0 +1 +0 +0 +1 +1

= 5



+1 +0 +1 +0 +1 +1 +0 +0

↳ cost 4

Input: $A[1..m]$

$B[1..n]$

best edit just of $A[1..i]$ + $B[1..j]$ $\quad 1 \le i \le m$, $1 \le j \le n$

$Edit(i, j)$

$= \min$ {

edit/match:
if $A[i] = B[j]$: $Edit(i-1, j-1)$
if $A[i] \neq B[j]$: $1 + Edit(i-1, j-1)$

insert:
$1 + Edit(i, j-1)$

delete:
$1 + Edit(i-1, j)$

}

+ Base cases:

if $i = 0$: insert $B[1..j]$
return j

if $j = 0$: B is empty; delete A's letters → cost i

(or both: return 0)

this way:

B is empty

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

A is empty

insert / delete

edit / match

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

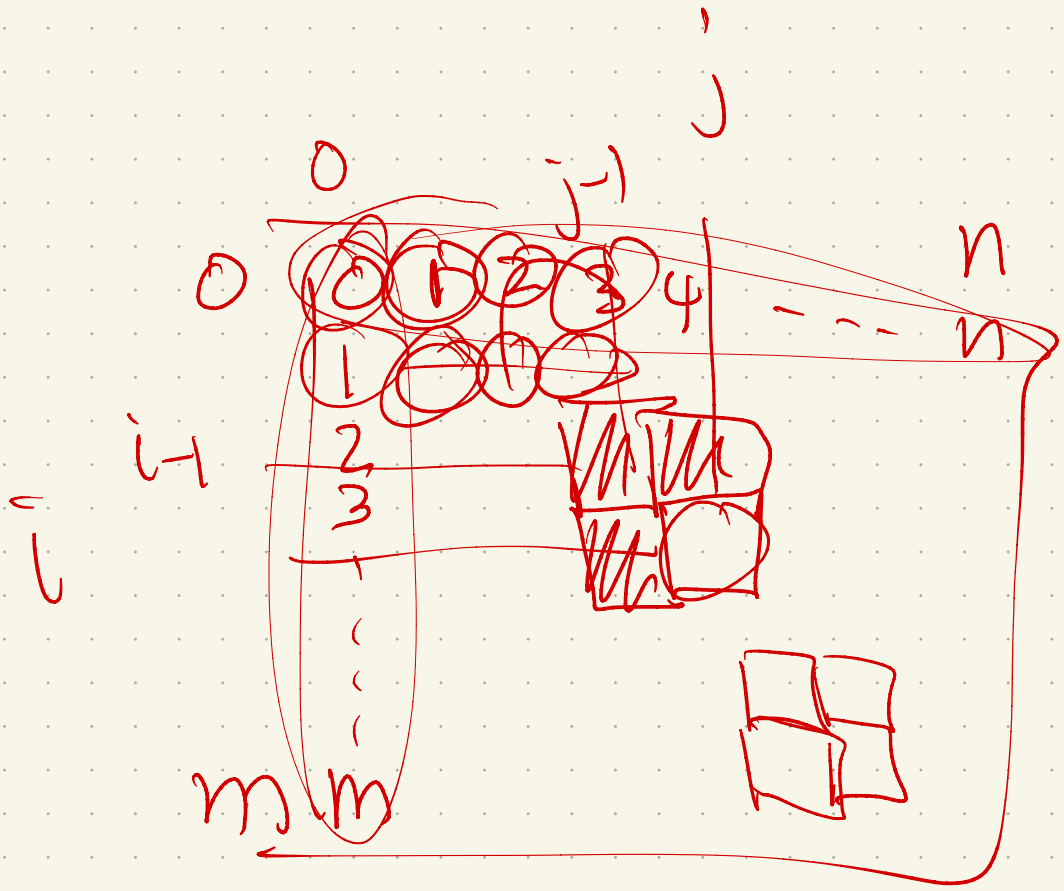
For each i, j pair, store a #

$\Rightarrow$ m x n array

Then, our algorithm:

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

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

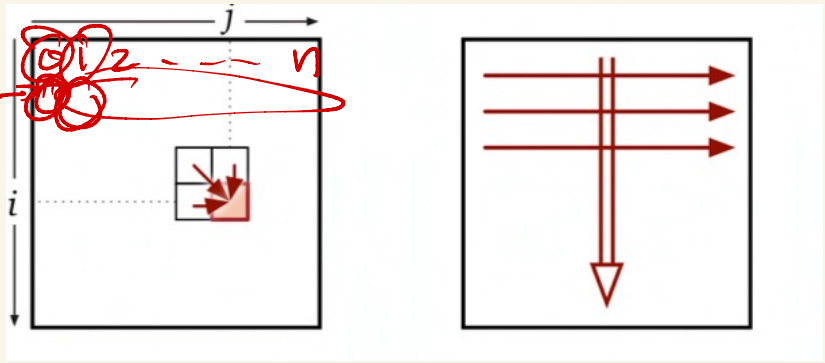


Result:

```
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

Picture:



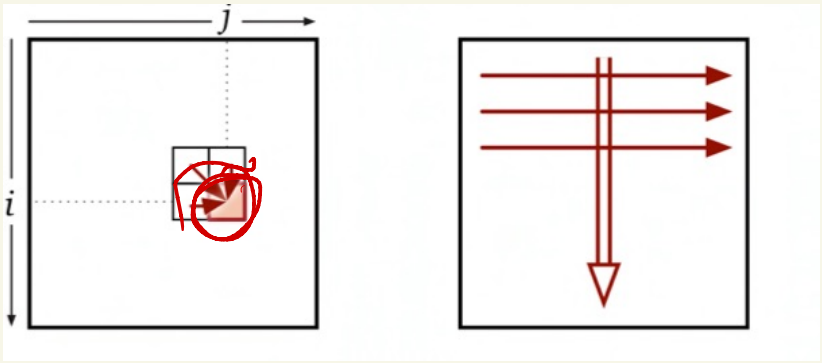
Result:

```
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]
```

Picture:



space: $n \times m$

$= O(nm)$

time: $O(mn)$

Back to an example:

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The memoization table for *Edit*(ALGORITHM, ALTRUISTIC)

```
A L G O R   I   T H M
A L T   R U I S T I C
```

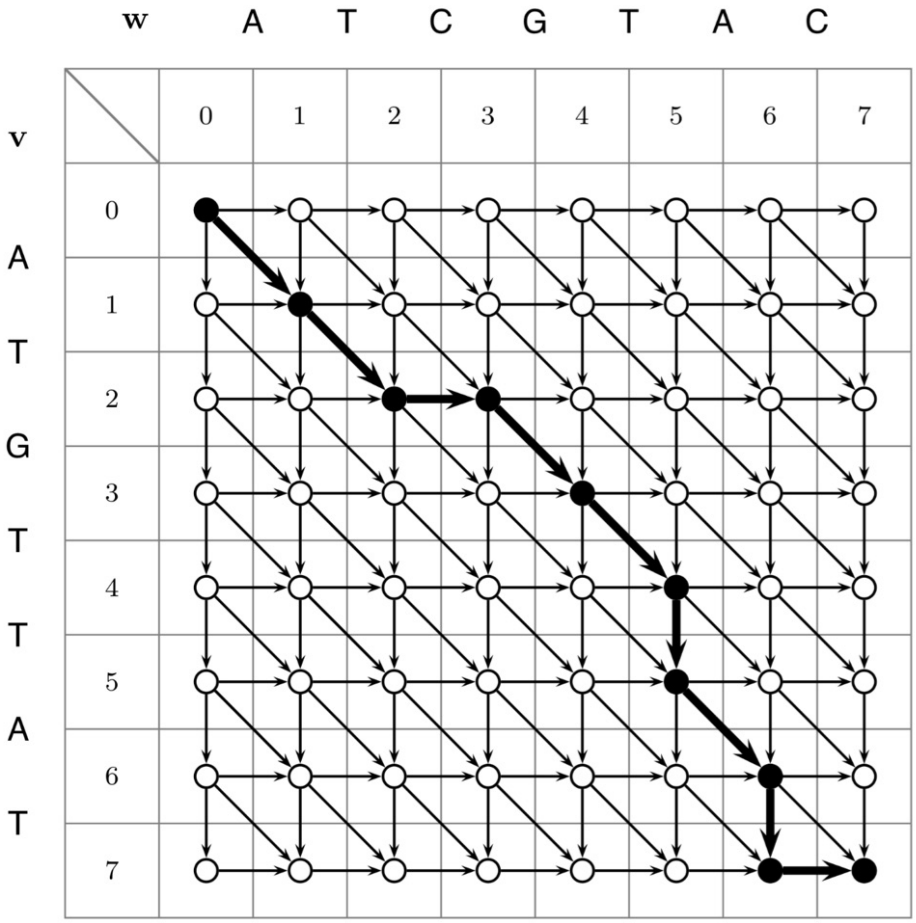
If non-uniform costs:

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

very by letter

Another (from Bioinformatics book):

		0	1	2	2	3	4	5	6	7	7
v	=		A	T	-	G	T	T	A	T	-
			|	|		|	|		|		
w	=		A	T	C	G	T	-	A	-	C
		0	1	2	3	4	5	5	6	6	7



↘	↘	→	↘	↘	↓	↘	↓	→
A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Question:

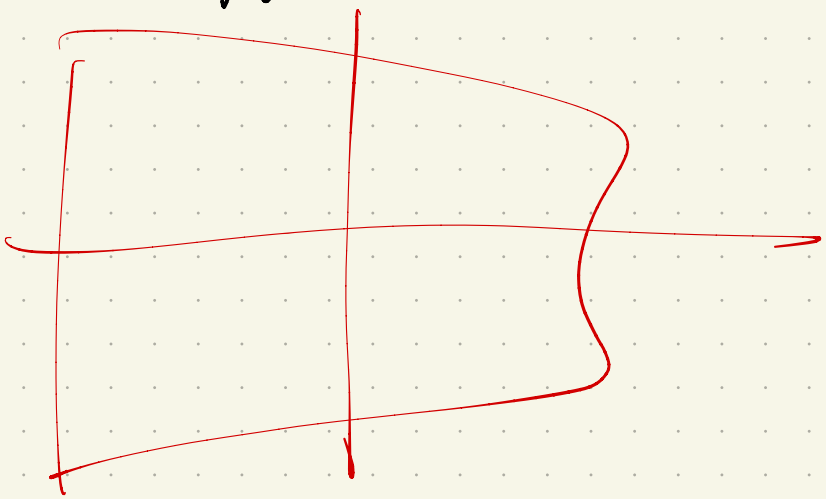
Can we do better?

A really good question!

Lots of attention in bioinformatics.

Clever divide an conquer can reduce space.

↳ but will give #, not sequence, w/out some nice tricks



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 $= T$.

Ch 2 solution

p. 78 of book

```
⟨⟨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)
```

The recursion:

(Note: same thing as code!!)

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

without or with

$SS(i,t)$ = True or False

$0 \le i \le n$

$0 \le t \le T$

So: another 2-d table!

To decide:

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

look at these 2 cells.

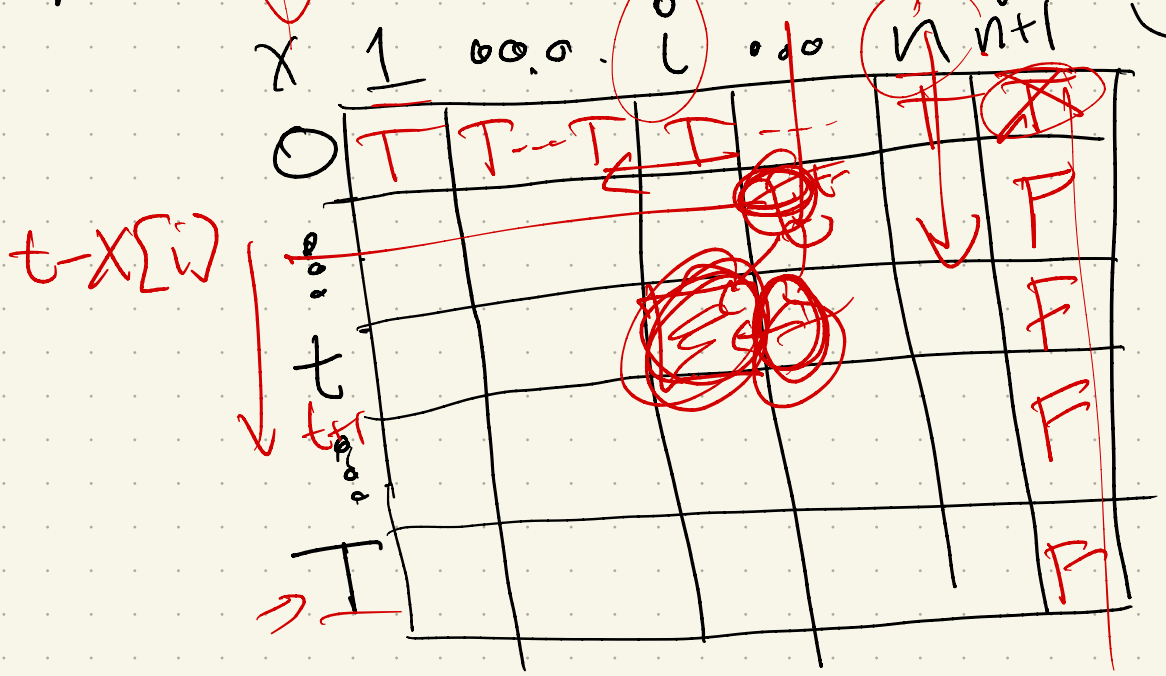
one note: if $t - X[i] < 0$, wasting time! Equivalent to:

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

Now – need to code this:

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

How should our loops go?



Fill: top to bottom
right to left

Its 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]
```

Correctness:

induction (brute force backtracking)

(same as

Time/Space Analysis:

Space: $O(nT)$

+ time $O(nT)$

Note:

How big is this, & is it even a good idea??

input: number T and array X[1..n]

log T

table has a column for every number 1..T.

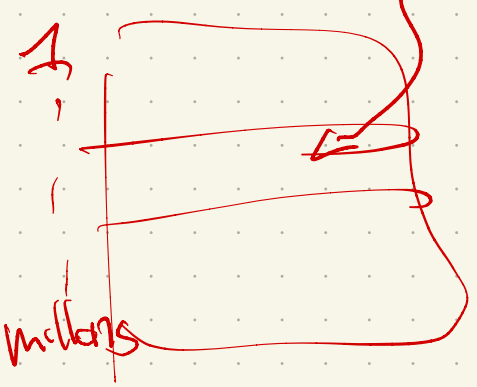
exp in terms of input

How bad?

Well, X could be a list of 5000 #s, but T could be in the millions!

(lots of empty columns, many of which are impossible to hit!)

only rows w/ #s will be T - X[i]

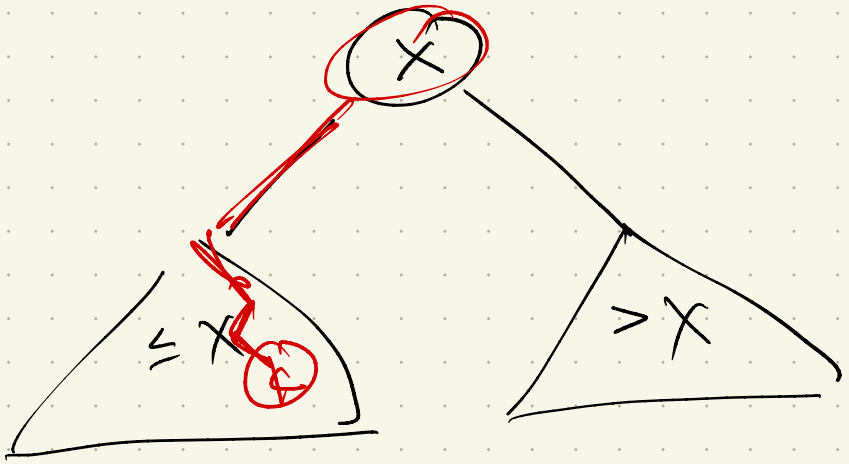


Balanced search trees (again)

Recall:

What is the "best" one?

Recap:



Time to search for k in T

= O(depth in tree of k)

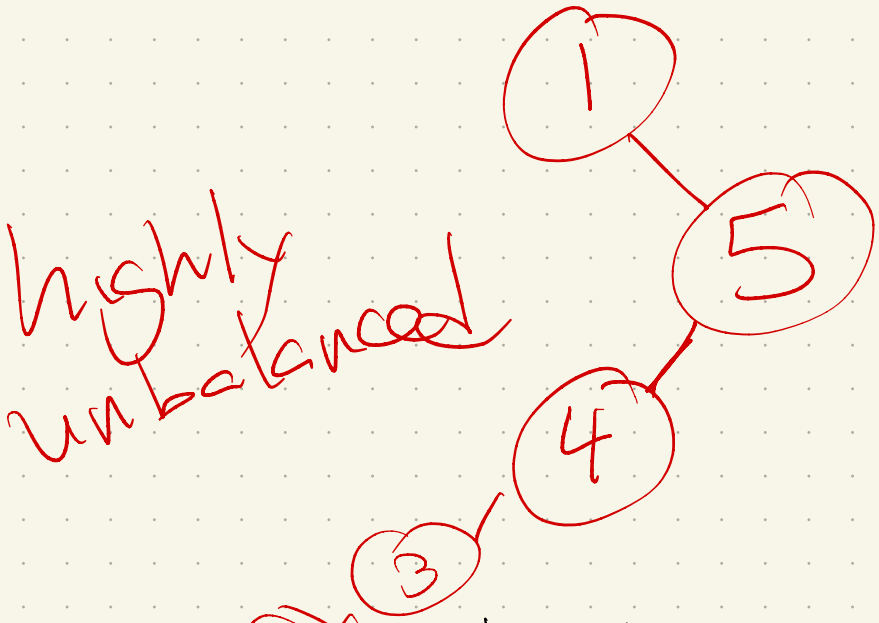
Goal: Given frequencies, built best BST for those frequencies.

Example:

f: 100, 1, 1, 2, 8

A: 1, 2, 3, 4, 5 ← assume sorted

Many BSTs: which is best?



highly unbalanced

Constructor 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!

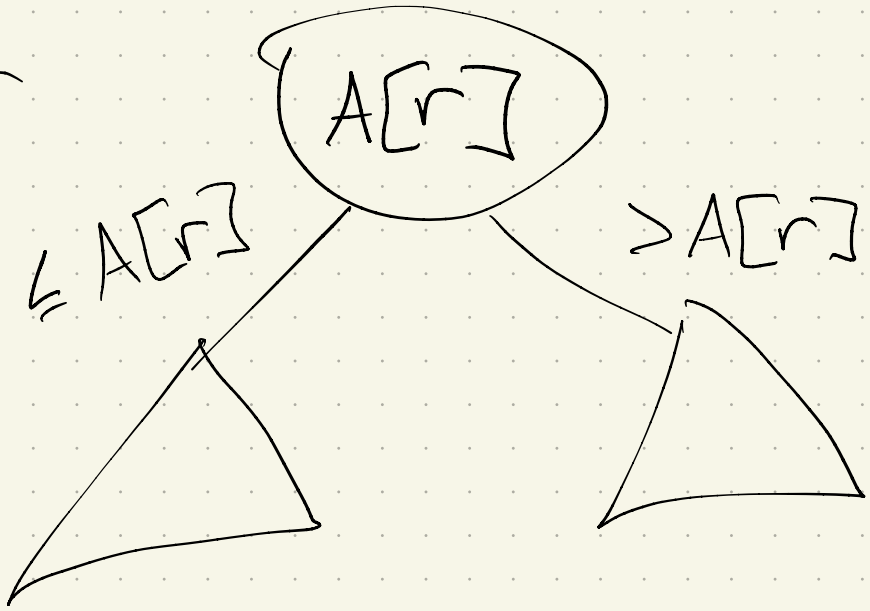
Last chapter:

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

$\Rightarrow$

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

Why??



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

So: we're regrouping!

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

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

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

Use this to build the "best" tree:

Choose root.

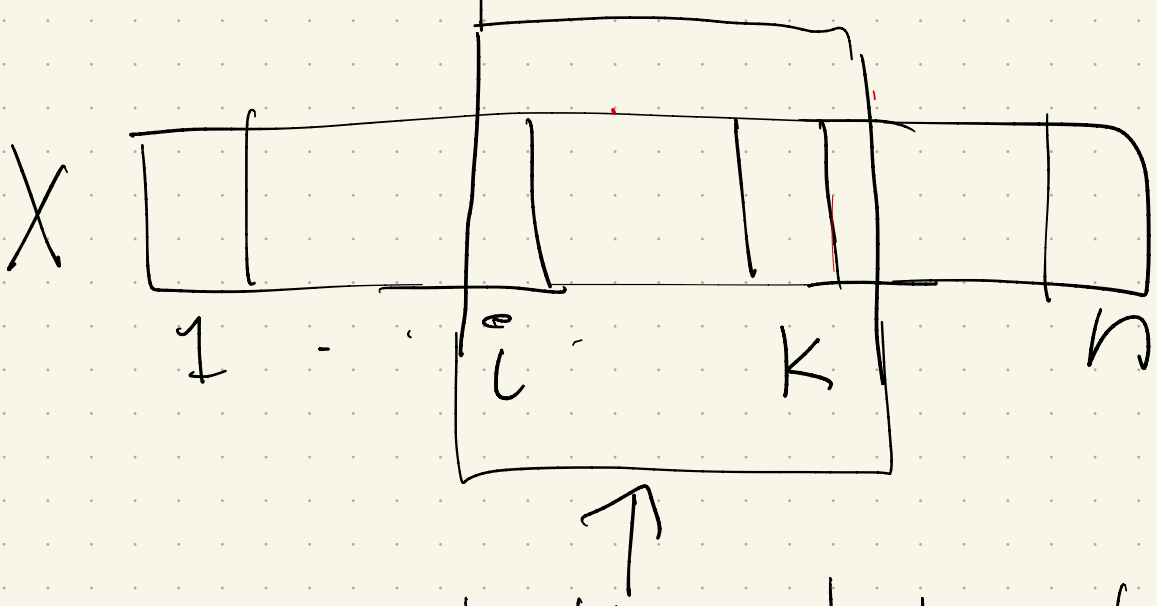
Recursively find best left subtree, + best right subtree.

(Note: try all roots in backtracking!)

How to memoize?

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

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:

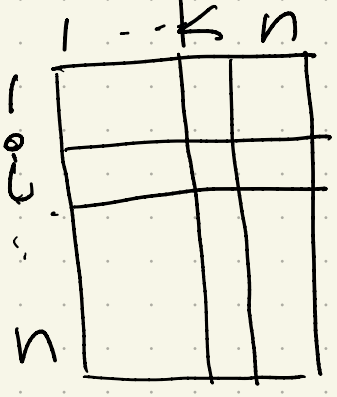
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{matrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{matrix} \right\} & \text{otherwise} \end{cases}$$

$\Downarrow$

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

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

So: 2d table!

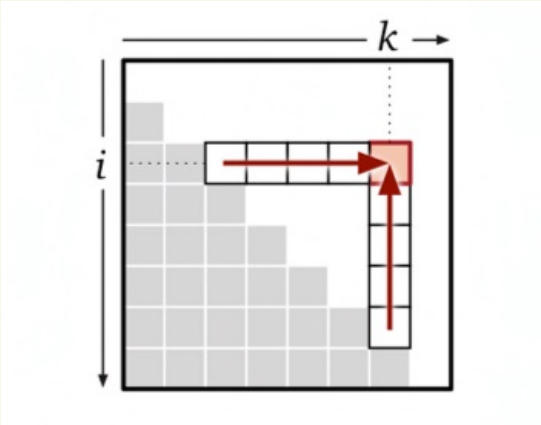


Each $O[i][k]$ needs:

- $F[i][k]$
- 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: