Lecture 4: Dynamic Programming

[A basic alg. paradigm like Greedy, Divide-and-Conquer.]

[Those have names making great sense. DP's name chosen in '40s literally to be obfuscatory, to make CS research sound cool to army bosses.]

[Better names:]
- Memo(r)ization
- Recursion w/o repetition

[Hallmark:] [Like Div & Conq but with] few, nested subprobs.

[Speaking of recursion, most traditional intro to it uses...]

Fibonacci numbers.. 0, 1, 1, 2, 3, 5, 8, 13, ...
0^{th} 1^{st} 2^{nd} 3^{rd}

[next is sum of two prev.]

[any guesses? 100^{th} ≈ 354 quintillion]

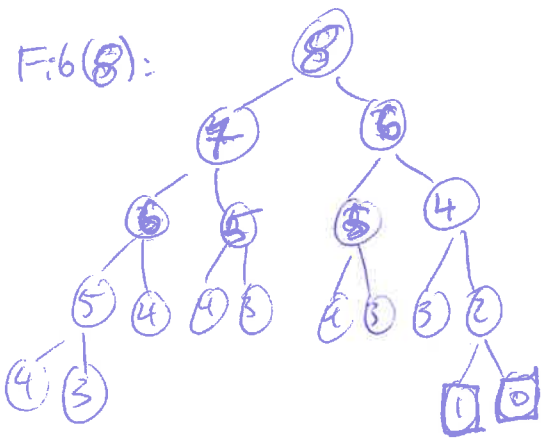
$F_{100} = F[100] =$

$F[n] \approx n/5$ digits

[recursive alg:]

```
def Fib(n):
    if n ≤ 1 return n
    else return Fib(n-1)+Fib(n-2).
```

Fib(8):



[correct "by def^n" Time?]

Let $T(n)$ = # rec. calls made by Fib(n).

$T(n) = T(n-1) + T(n-2)$!

[Diff base case... $T(0) = T(1) = 1$, $T(2) = 2$.]

$T(n) \approx 2 \cdot F[n-1]$

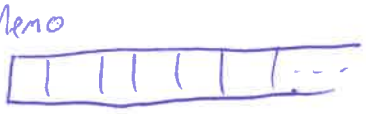
about $10^{n/5}$

$\approx \phi^n$ ← 1.6... [exponential]

[But my god, look at tree. Computing same result over & over!]

"Memoization": Fib(n):

if Memo[n] ≠ null, return Memo[n]
if n ≤ 1: answer := n
else answer := Fib(n-1) + Fib(n-2).
Memo[n] := answer
return answer.



[Now how many recursive calls total?
→ every time we do recursive call, we do 2, but we fill one entry of Memo[·]
∴ rec calls ≤ 2(n+1)
2 # entries memo.] [exponential improvement!]

"Top-down D.P.": Define recursive alg., memoize.
[Can sort of always do it, but when does it really help? When "few" (O(n), O(n²)) subprobs.]

[Conversely, think abt. recursive MergeSort...
subprobs are like unsorted lists → expon many]

[Say,] How does recursion actually work? [Not magic. Your chip just serially does instructions. It effectively builds the above tree...]
Actually fills Memo in L-to-R order

0, 1, 1, 2, 3

[Can do this intentionally...]

Bottom-Up D.P.: fill out memoization results in order recursion would...

Fib(n):

F[0] := 0, F[1] := 1

{ for i = 2 ... n:

F[i] := F[i-1] + F[i-2]. // duh!

Time: ~~O(n)~~ $\Theta(n^2)$ // adding two O(n)-digit nums!

Space? n array entries of ≤ n digits ~ $O(n^2)$

Often space can be reduced in D.P.: selective forgetting

{ deallocate memoization results you're sure you won't need!

// just need two remem 3, not n, #'s for Fib(n)

if n ≤ 1 return n:

prev := 0

curr := 1

for i = 2 ... n:

next := curr + prev // think about it!

prev := curr

curr := next

return curr

// Another important alg. for comp bio/genomics...

L.C.S. = Longest Common Subsequence.

Input: two strings x, y: e.g. x = GACATTACGA (len. "n")
y = GCTACAGT (len. "m")

[[diff lens ok!]]

Out: L.C.S. [[not substring; discontig ok]]

length of

[[for now, makes it easier...]] ↳ GCAAG (5)

[[Like you can match chars up but in non crossing matching way]]

[[Brute force bad, 2^n subseqs of x]]

[[Basic first idea of D.P.]]: Find recursive definition/alg.

[[Not totally obv., but some greed is good.]]

- if $x[1] = y[1]$, may as well "take it"
  [[can never hurt; scores a pt., blocks nothing]]
  ↳ ans. is $1 + LCS[x[2...n], y[2...m]]$
- else...
  $LCS(x[1...n], y[2...m])$ or
  $LCS(x[2...n], y[1...m])$, whichever better.

How to "index"

Q: the subprobs?

A: $x[i...n]$, $y[j...m]$. ←

```
def: LCS(i,j):   // find LCS len. of
     if i>n or j>n, return answer:=0   // got down to at least one empty string
     else if x[i]=y[j]: answer:= 1+LCS(i+1,j+1)
     else answer:= max(LCS(i+1,j), LCS(i,j+1))
     return answer.
```

[[exercise: show # of rec. calls $= 2^n \cdot 2^m$ ☺]]

Memoize!

→ if Memo[i,j] = null:

↳ Memo[i,j] := answer.
return answer

Time? Each 1 or 2 rec. calls gives Memo entry
↳ m·n.

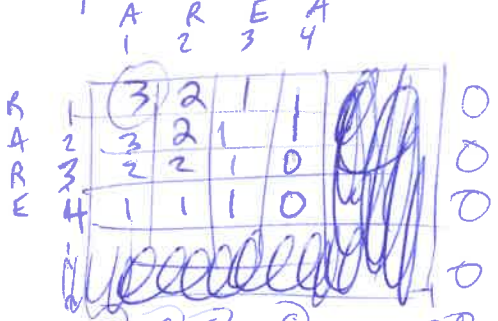
["quadratic" ≠ ("few subprobs")]
n^2

"Bottom-up"? [in what order are entries actually filled...?]

[Better to think: what's a sensible order to fill so that I'll always have the prior answers I'll need?]

eg: x=RARE
y=AREA

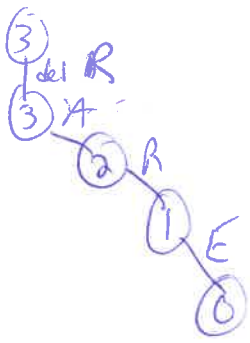
needs



By rows or cols OK, just from m,n ↓ 1.

```
for i = 1...n+1
    LCS[i,n+1] := 0
for j = 1...m+1
    LCS[n+1,j] := 0
for j = m, m-1, ... 1
    for i = n, n-1, ... 1,
        if x(i) = x(j): LCS[i,j] = 1+LCS[i+1,j+1]
        else
```

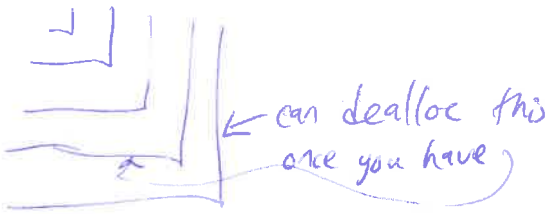
Q: Given Memo[,] how to find actual subseqs?
A: Start at top-left, "follow decisions".



Improve Space usage?

∴ can get space to $O(m+n)$

[[Just for finding len.]]



[[NOT so simple to see how to actually find the subseq. w/ only using $O(m+n)$ space, but can be done. Clever!

<< quadratic time? No! So sad, but ≈ 5 years ago proved "impossible" (assuming a $P \neq NP$ variant).]]

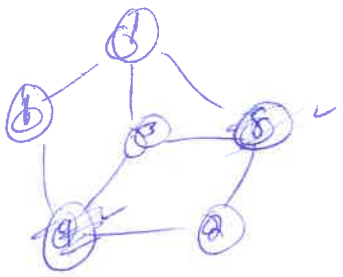
[[One more example ... just try to set up recursion...]]

Max-Weight-Indep-Set:

Input: $G = (V, E)$, "weights" $w(v) \geq 0$ $\forall v \in V$.

Output: An "indep set": $S \subseteq V$ s.t. no two verts in S have an edge

Goal: Maximize $w(S) := \sum_{v \in S} w(v)$.



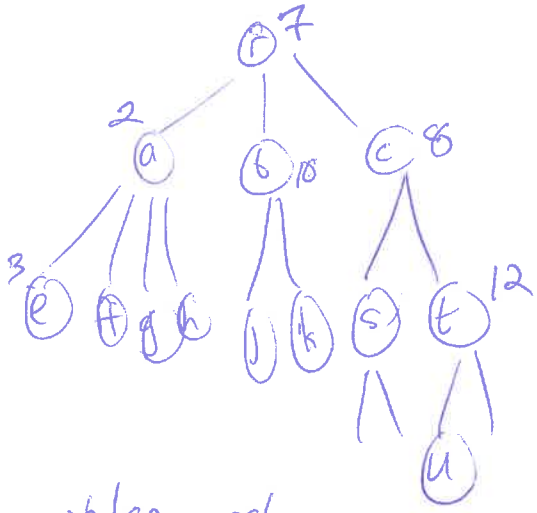
[[Think: places to put broadcasting devs. Get pts for placing at var. spots. Can't have two adj. Interference!]]

NP-hard! ☹ [[no subexp. time alg. prob.]]

☺ Efficiently solvable if G is a tree.

[[More generally if G has small "treewidth"...]]

Make G a rooted tree:



[[Trees + recursion = mWah!
A rooted tree is a node, with children who are rooted trees...]]

$M.W.I.S.(r) :=$ max wt. indep. set you can get in tree rooted at r...

// idea: can either "take" r, get 7, but then you can't take kids... but the grandkid probs are sep

or don't take, and solve kids...

$$MWIS(r) := \max \left\{ w(r) + \sum_{v \text{ a grandchild of } r} MWIS(v) ; \sum_{u \text{ a child of } r} MWIS(u) \right\}.$$

$$MWIS(\text{leaf } \ell) := w_\ell.$$

Memoize as before! $O(n)$ space, one slot for each vtx.

Bottom-up? [[A bit annoying, need to traverse from leaves up

Great case for using recursion and letting compiler handle it!!]]

[["post-order traversal"]] [[$O(n)$ time for subtle reason: each vtx only looked at by parent & grandparent]]