Class outline:

1. Examples of real world applications (as motivation)
2. simpler example to gain intuition (since not always intuitive the first time)..
3. Back to applications and other examples (next couple classes)

Applications: very useful approach today - some examples - non exhaustive

- Computational Biology ✓
- pattern / speech recognition
- finding shortest path (later)
- cypher to Thomas Jefferson (will mention)

(a) Computational Biology: genome similarity:

- strings from alphabet $\{A, C, G, T\}$

  Example: ACGGAT
  CCGCTT

  (1 long case of DNA - 4 bases)

  human genome: ~6 Billion
  gene - distance lengths - could be 10,000 bases

- (ask) How "similar"? we'll (see why we care and how we do it)

- Why? If we understand function of one string (mouse genome), can teach us about function of other string (human genome) which know less about.

- How? (ask)
  - number changes from one to other
  - small; allowed to insert blank, change character
  - longest common subsequence

  We'll look for Longest common subsequence (must appear same order, not necessarily consecutive)

Example: A C G G A T

C C G C T T

(English: C A T
C H A T)

Question: longest common subsequence length? (ask)

Answer: 3 CGT.

Efficiency: How many possible subsequences? (ask)
small example we can scribble - but

Let's say string of length $n = 500 \ldots$

A C C C G G T C G A G T G ...

G T C G T T C G G A A T T ...

Brute force: consider all possible subsequences of first string and check if also subsequence of second string.

$n = 500$: 2^{500} Not feasible!

actually needs to multiply by length of 2^n string, since check each subsequence in second

To see: pick first character or do not...
pick 2nd character or do not...
pick 3rd character or do not...

$\underbrace{2 \cdot 2 \cdot 2 \cdots 2}_{n \text{ times}} = 2^n$

Therefore: need efficient algorithm!

Intuition: We'll save solutions to smaller subproblems; we'll come back to this...

→ will divide and conquer work? →

No, not in a simple way:
could for this example

A C G G A T
C C G C T T

But generally, not straightforward, eg this won't work:

C G T G A C
C G G T T T → Finds CG, not CGT

→ we'll come back to Dynamic Programming solution

simple example (to build intuition):

Fibonacci!

(Everyone has heard of? I'm going to teach it a little differently)

First write naive recursive algorithm:

Fib(n)

1. if n==0 return 1 (1 or 0 depending on source)
2. if n==1 return 1
3. else return Fib(n-1) + Fib(n-2)

Good algorithm? (ask) Does the job but...

No! Very wasteful

Ways to see:

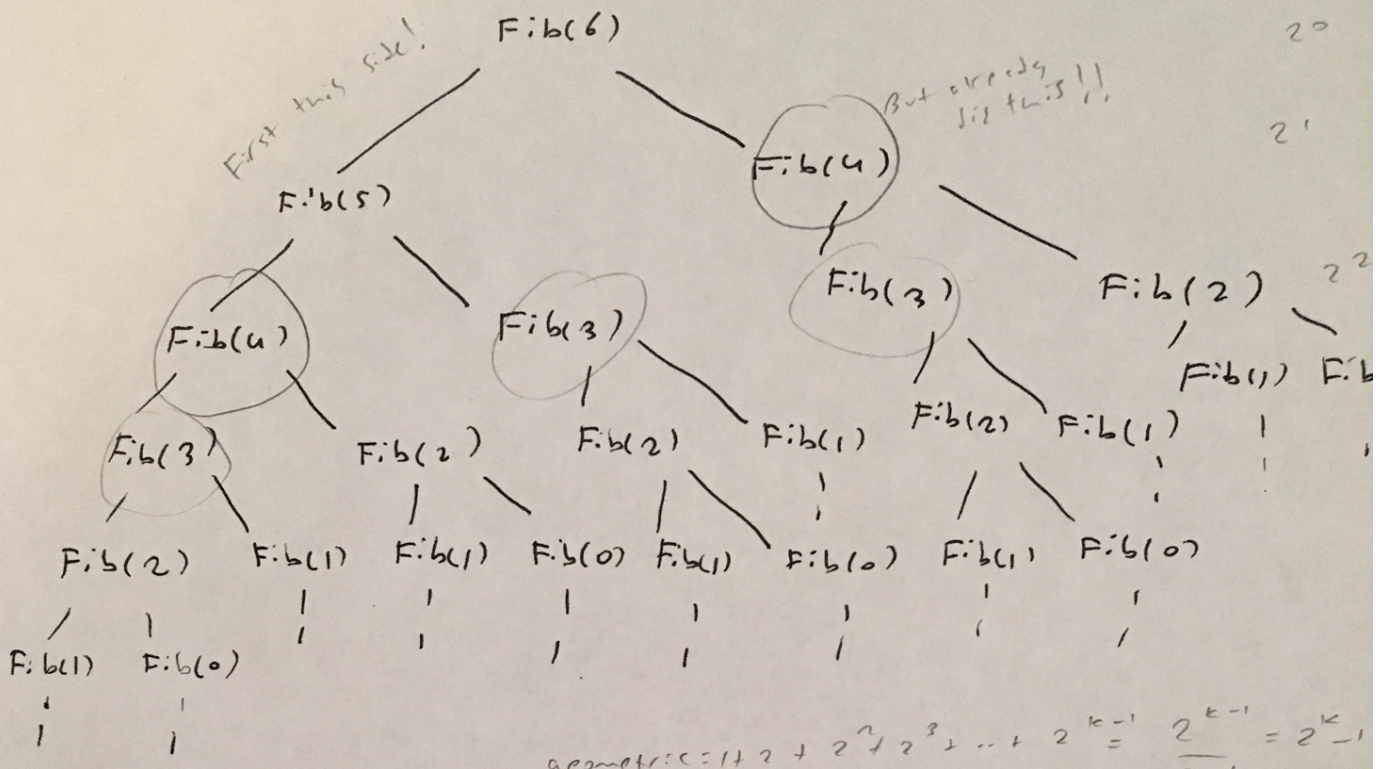
1. To compute Fib(25) you compute Fib(24) and Fib(23). To compute Fib(24) you compute Fib(23) again!, and Fib(22). That's a lot of re-computing and a lot of work! We'll look at a recursion tree in a moment.

Fib(25) = Fib(24) + Fib(23)

Fib(24) = Fib(23) + Fib(22)

2. Remember from divide & conquer how to write out recursion:
   - $T(n) = T(n-1) + T(n-2) + \text{const}$ (to combine, adding)
     $\geq 2T(n-2)$
   - each time subtract 2 and multiply by 2 ($\frac{n}{2}$ times)
     $T(n) \geq 2^{\frac{n}{2}}$ (run time at least $2^{\frac{n}{2}}$)

3. To get more intuition, recursion tree for n=6: (ask to do)



→ Fib(4) computes twice!

→ Fib(3) three times!

(imagine if we started from Fib(50)!)

Bad: keep repeating computations. (if I knew answer to Fib(4), wouldn't have to keep repeating whole subtree!)

4. See online: order of computing:

www.cs.usfca.edu/visualization/DPFib.html

www.usfca.edu/ngalles/visualization/DPFib.ht

(example for 6)

missing ngalles

Summary so far:

Fib problem has two important properties:

1. overlapping subproblems (lots!)
2. solution to big problem can be constructed from solutions to subproblems

→ type of problem well solved by dynamic programming

Dynamic Programming Fibonacci:

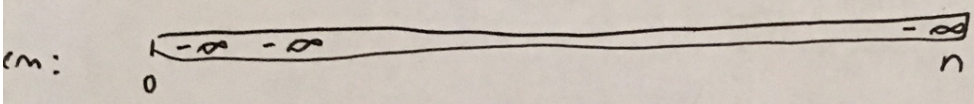
main idea: save in dictionary subproblems already solved so don't need to recompute (array or hash table).

called memoization: from memo pad or memory

memoized Dynamic Programming algorithm:

a. initialization:

Let mem be a new array with values initialized to $-\infty$



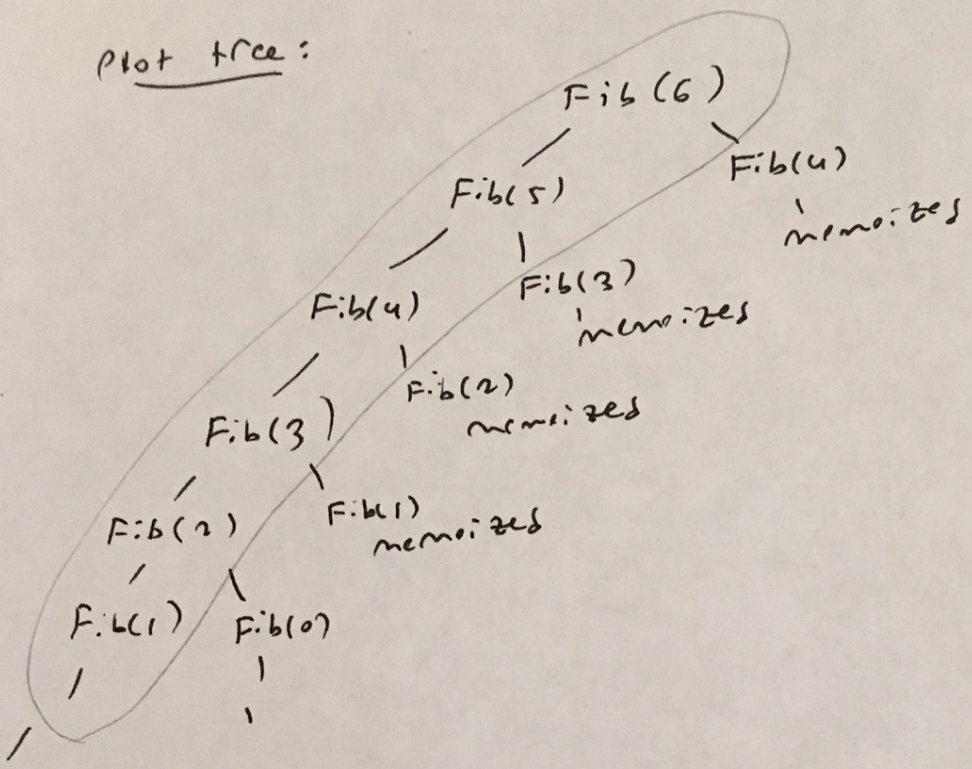
mem is dictionary each time compute Fib number, first check if already in dictionary

b. Fib(n) // memoized dynamic programming version

```
1. If mem[n] ≥ 0                 } if already previously computed and in memo pad
2.     return mem[n]
3. if n==0 return 1              } otherwise compute and save value
4. if n==1 return 1
5. else f = Fib(n-1) + Fib(n-2)
6.     mem[n] = f                } save value in memo pad
7.     return f
```

How fast? (ask)

Plot tree:



- run time proportional to n (= 6 here) $\Theta(n)$
- 2nd time encountered: just use memoized result.

  (cuts off whole recursion subtrees!)

  # subproblems n
  time per subproblem: constant

- see online: www.cs.usfca.edu/visualization/DPFib.html
- see online: Burt Rosenberg:
  www.cs.miami.edu/home/oselia/teaching/fib.html

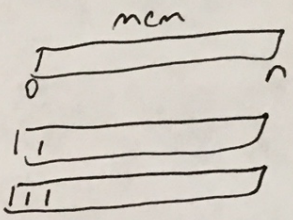
Summary: recursion + memoization

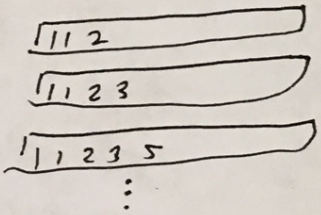
Another Fibonacci Dynamic Programming algorithm: Bottom up:

(point to tree; recursive starts at Fib(n) and works down; here start from bottom-more like you first learned Fib!)

Fib(n) // bottom up dynamic programming

1. Let mem[0..n] be a new array
2. mem[0] = 1
3. mem[1] = 1
4. For i = 2 to n
5. mem[i] = mem[i-1] + mem[i-2]
6. Return mem[n]





(could also just keep around last two in table)

same as before

Question: Same or different from previous memoized algorithm?

Answer: Easy to see (one For loop) time is $\Theta(n)$

Equivalent solution to memoization (same things happen in same order)

(can see online: www.cs.usfca.edu/visualization/DPFib.html)

Going back to Longest common Subsequence

problem: (DNA similarity)

(start to get intuition; will continue this and another problem next time)

Example: C C G C T T x_i
A C G G A T y_j

Longest common subsequence: C G T

LCS (6, 6) = 3 // length of longest common subsequence

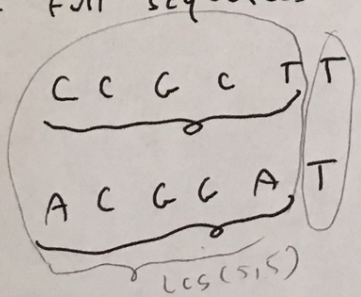
x_i y_j

C C G C T T
A C G G A T

How to formulate recursion:

→ as in Fib, we'll try and start from largest sequence, and formulate recursion from smaller subproblems:

→ Look at full sequences $i = j = 6$:



ideas for formulating recursively?

In this example: $x_6 = y_6 = T$:

solution is that of smaller subproblem length 5, plus 1 (for the T T)

$x_i = x_j$:

C C G C T T
A C G G A T

$i = j = 6$

$LCS(6,6) = LCS(5,5) + 1 = 3$

$LCS(5,5)$: C C G C T / A C G G A → $= 2$ CG; $+1$ T → CGT

Longest common subsequence has to include (T,T);
otherwise we could always append (T,T),
finding longer sequence - contradiction.

$x_i \neq y_j$: a little more tricky:

Let's say we have:

C C G C T C
A C G G A T

$i = j = 6$

this can't both be in LCS; what to do?

$LCS[6,6] = \max(LCS[5,6], LCS[6,5]) = 3$

$LCS[5,6]$: C C G C T / A C G G A T → $= 3$ CGT

$LCS[6,5]$: C C G C T C / A C G G A → $= 2$ CG

Summary:

$$LCS[i,j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ LCS[i-1,j-1]+1 & i,j>0 \quad x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & i,j>0 \quad x_i \neq y_j \end{cases}$$

Dynamic Programming 2

Reminder previous class:

Longest common subsequence:

C C G C T C

A C G G A T

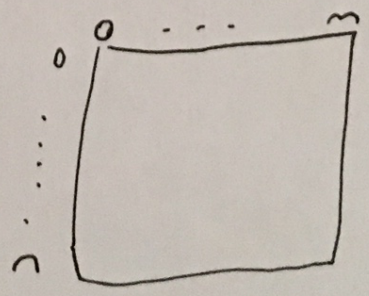
answer: CGT

Recursive solution:

$$LCS[i,j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ LCS[i-1, j-1]+1 & i,j>0,\ x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & i,j>0 \quad x_i \neq y_j \end{cases}$$

Dynamic programming solution:

→ Define table $C[0..m, 0..n]$



$n = x.length$

length of first subsequence

$m = y.length$

length of second subsequence

→ Either memoize solutions to subproblems not yet computed, or compute solutions to sub-problems bottom up.

→ Run time: $\Theta(mn)$

mn subproblems (at most)

constant computation each

→ we'll write out full bottom up solution (similar to animation we saw)
www.cs.usfca.edu/~galles/visualization/DPFib.html

Note: main properties that allow DP solution: (ask)

- overlapping subproblems.
- solution to big problem constructed from solutions to smaller subproblems.

Bottom-up from book:

```
LCS-LENGTH(X, Y)
1.  m = X.length
2.  n = Y.length
3.  let b[1..m, 1..n] and c[0..m, 0..n] be new tables.
4.  for i = 1 to m                          ]
5.      c[i, 0] = 1                         ] initialize
6.  for j = 0 to n                          ]
7.      c[0, j] = 0                         ]
8.  For i = 1 to m
9.      For j = 1 to n
10.         if x_i == y_i:
11.             c[i, j] = c[i-1, j-1] + 1
                b[i, j] = "↖"
12.
13.         elseif c[i-1, j] ≥ c[i, j-1]
14.             c[i, j] = c[i-1, j]
                b[i, j] = "↑"
15.
16.         else c[i, j] = c[i, j-1]
17.             b[i, j] = "←"
18.     return c and b
```

key to reconstructing (right answer of characters)

saves value of length

run time: $\Theta(mn)$

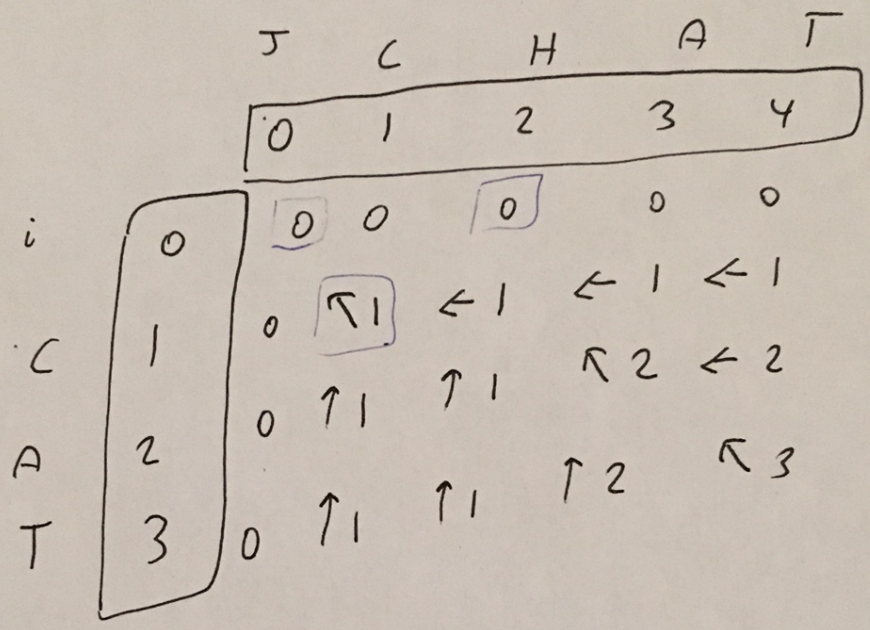
Example:

Filled: left to right, from top

CHAT x_i →

→

CAT y_j :



Printing:

"↖" : print(i-1, j-1) // recursive follow arrow
x_i // print out x_i

"↑" : print(i-1, j) // recursive

"←" : print(i, j-1) // recursive

C A T

(Todo yourself: memoized version + see also animations.

→ We've structured as large subproblem composed of smaller subproblems.

→ Is our recursive solution efficient?? (ask)

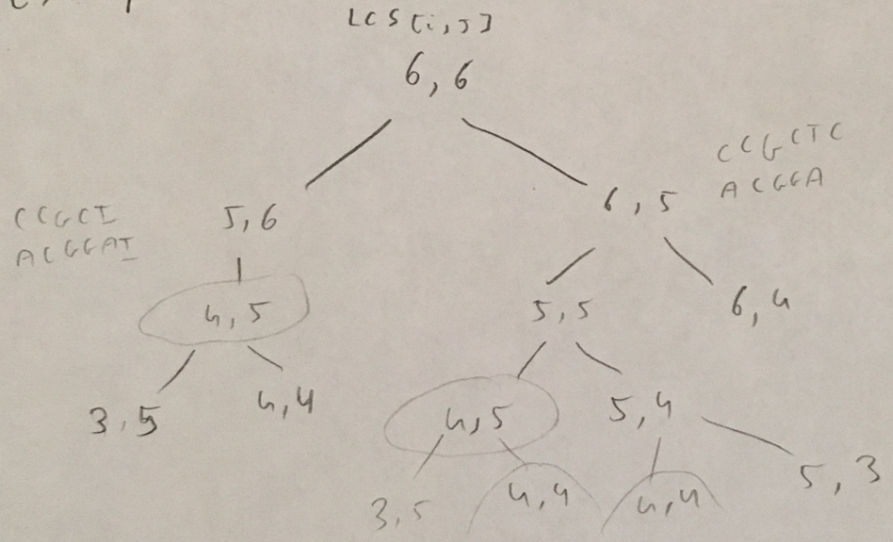
Answer: No. Only if memoized previous solutions (or build bottom up, saving answers to smaller subproblems). → Just like in Fib!

→ Are there overlapping subproblems? (ask)

Answer: yes.

CCGCTC
ACGGAT

Example:



(we'll write out Dynamic Programming algorithm next class)

indices start from -1 here

CAT
CHAT

→ show animation:
www.cs.usfca.edu/ngalles/visualization/DPLCS.html

gets stuck in regular DN