COP 3503 11/10/26

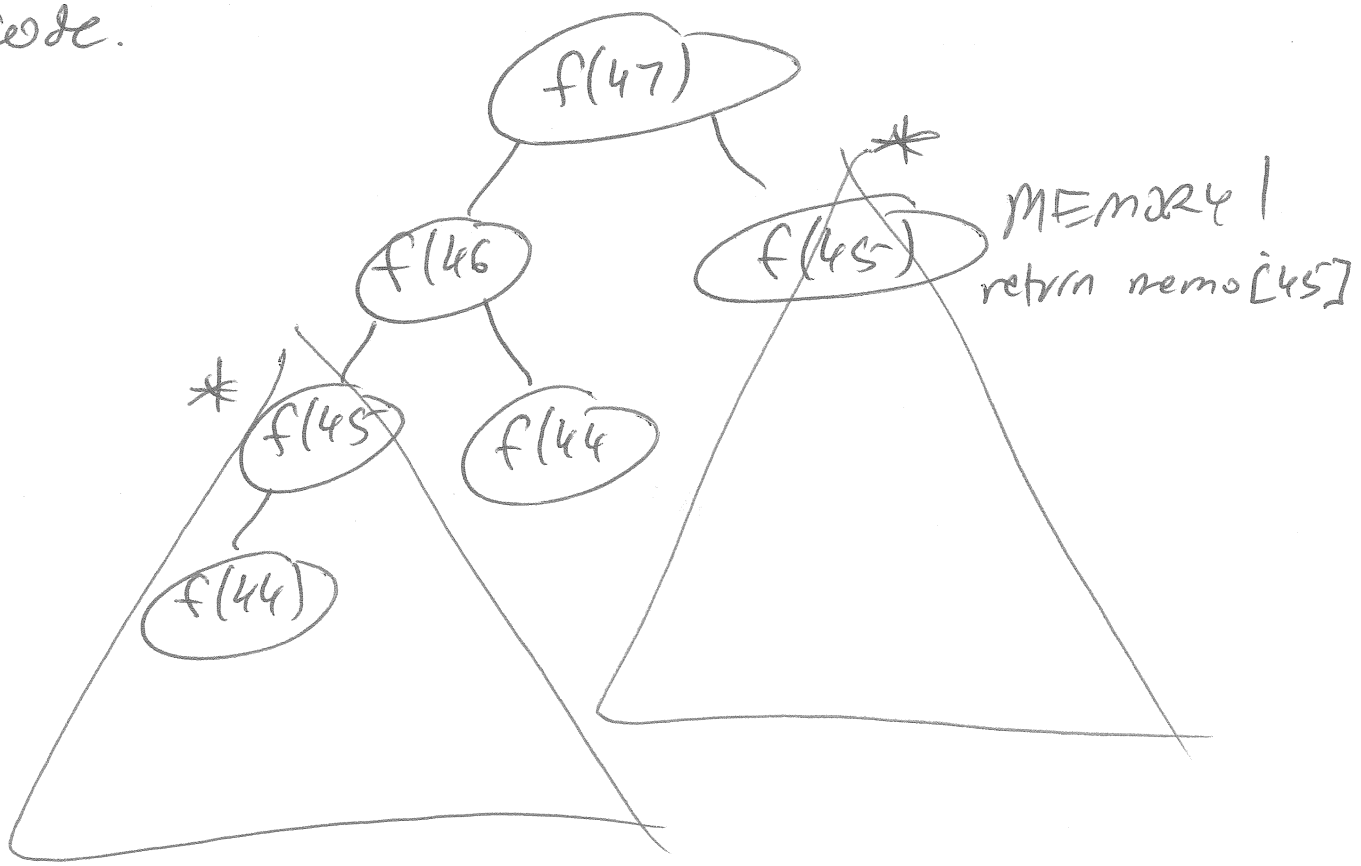
(0) Go over test

(1) DP - Idea, Fib, LCS, Edit Distance?

(2) Whiteboard Schedule

Dynamic Programming (DP)

If a ~~res~~ recursive solution is inefficient due to making identical recursive calls multiple times, we can use D.P. to make each unique recursive call once + speed up our code.



Memoization Steps to Speed Up Recursion

1. Create storage for all unique recursive calls (usually array or map)
2. Fill storage w/ -1 or sentinel value indicating unsolved. (Map do nothing → check if key is there)
3. Add base case to recursion
   if memo[ ][ ] != -1 or if map.contains key
   if we're done at, return answer
4. Before you return store answer in memo table/map.

Dynamic Programming

⇒ Remove recursion

1. Identify all necessary recursive calls
2. Figure out an order to "mate" the recursive calls so that you always already have the answers to the cases you need.
3. Implement this ordering storing results in a table/map

fibonacci

need access to smaller fibs

0	1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	1	1

(1) Seed table w/ beginning values (base cases)

(2) build answers in an order that work

```
f[0] = 0; f[1] = 1;
for (int i=2; i<=n; i++)
    f[i] = f[i-1] + f[i-2];
```

Longest Common Subsequence

Input : 2 strings/sequences

Output: length of the longest common subsequence

f(i,j) = lcs of 1st i char S_1 and 1st j chars S_2

		A	C	C	G	T	A	C	G	A	T
	0	0	0	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1	1	1	1	1
T	0	0	1	1	1	2	2	2	2	2	2
A	0	1	1	1	1	2	3	3	3	3	3
G	0	1	1	1	2	2	3	3	4	4	4
C	0	1	2	2	2	2	3	4	4	4	4

$$f(i,j) = 1 + f(i-1, j-1) \quad \text{if} \quad s_1[i-1] = s_2[j-1]$$

$$= \begin{cases} \max \begin{bmatrix} f(i-1, j) \\ f(i, j-1) \end{bmatrix} & \text{otherwise} \end{cases}$$

```
public static int lcs(char[] s, char[] t) {
    int[][] dp = new int[s.length+1][t.length+1];
    for (int i = 1; i <= s.length; i++) {
        for (int j = 1; j <= t.length; j++) {
            if (s[i-1] == t[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = Math.max(dp[i][j-1],
                                    dp[i-1][j]);
        }
    }
    return dp[s.length][t.length];
}
```

EDIT DISTANCE

Input s (start string)
t (target string)

Goal is to transform s into t using the minimum # of operations.

The operations (allowed) ARE

1) Delete a letter
2) Add a letter (insert)
3) Change a letter

~~HORUS~~

HORSE ↔ HOUSE

HORSE
↕
HOSE

LORSE
↓
LOSE

If letters match, greedily match them

S A C G T T A

t C C A T

Change it
~~delete it~~ delete it

We can think of adding a letter to end of s as deleting a letter from end of t

$f(i,j)$ = edit distance of the first i letters of s and 1st j letters of t

$$f(i,j) = 0 + f(i-1, j-1) \quad \text{if } s[i-1] = t[j-1]$$

$$= \min \begin{cases} 1 + f(i-1, j) & \text{del last char } s \\ 1 + f(i, j-1) & \text{del last char } t \\ 1 + f(i-1, j-1) & \Delta \text{ last char } s \text{ to last char } t. \end{cases}$$