COP ~~3502~~ 4516 2/3/26

1) Review what DP means when it can be used.

2) Quickly look at "Classic Problems" taught in algorithms course.

3) Live solve: heritage (Kattis)

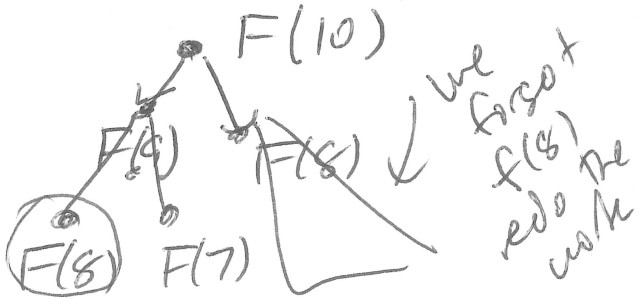
Dynamic Programming applies to problems w/ optimal substructure:

- Optimal answers to larger cases can be built upon optimal answers to smaller cases.

↳ Sounds like recursion

Big Difference -

DP is particularly useful IF the natural recursive breakdown of a problem results in redundant function calls



To avoid redundant recursive calls store the answer to each ~~re~~ unique recursive call and if you know the answer, skip the recursion!

STORAGE:
① Arrays
② MAP

① write recursively with global memo table (storing ans to old calls

② write iteratively, reordering the unique function calls "bottom up" so that you already know the answers for smaller cases that you need.

Standard Problems

Longest Common Subsequence

Knapsack

Longest Increasing Subsequence

fromCS2 * # ways to make change
* fewest # coins to make change

LCS (String s, String t)

s = A C C A B A C

t = C B A B C A C A B

LCS (s[0 ... n-1] , t [0 .. m-1])

= if (s[n-1] ≠ t[m-1]) → cut off last t

max (LCS (s[0 .. n-1], t[0 .. m-2]),
LCS (s[0 .. n-2], t[0 .. m-1]))

→ cut off last s.

else

1 + LCS (s[0 .. n-2], t[0 .. m-2])

let lcs(i, j) = lcs of first i chars of s and first j chars of t.

↓ array

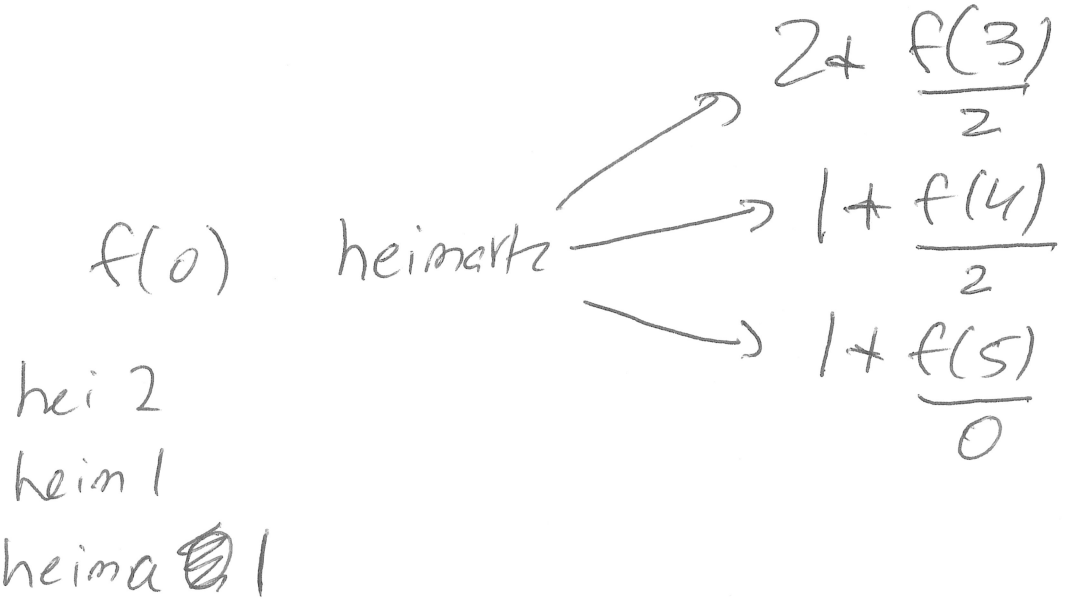
dp[n+1][m+1]

dp[i][j] stores

memo

```
lcs ( int i, int j ) {

    if (i==0 || j==0) return 0;

    if ( memo[i][j] != -1) // In mein set memo = -1
        return memo[i][j];

    if (s[i-1] == t[j-1])
        return memo[i][j] = 1 + lcs(i-1, j-1);

    return
        memo[i][j] = max( lcs(i-1, j), lcs(i, j-1));
```

heritese



hei 2
hein 1
heima 1