Week 10!

CMSC389O: The Coding Interview

Today

- Dynamic Programming
- Sample Questions
- Group Problem Solving!

Reminders: Recursion

Recursion: How to approach

1. Think about what the sub-problem is: How many sub-problems does f(n) depend on?
2. Solve for a “base case.” That is, if you need to compute f(n), first compute it for f(0) or f(1) This is usually just a hard-coded value
3. Solve for f(2)
4. Understand how to solve for f(3) using f(2) (or previous solutions) That is, understand the exact process of translating the solutions for sub-problems into the real solution
5. Generalize for f(n)

Recursion: Complexity

- Time
  - O(n) if you call the recursive problem n times
  - O(logn) if you split the input by a certain amount
  - Exponential depending on how many times the function calls itself
    - Ex. Fibonacci (2^n)
- Space
  - Think about heap memory AND stack memory

Recursion: Complexity

- Time
  - O(n) if you call the recursive problem n times
  - O(logn) if you split the input by a certain amount
  - Exponential depending on how many times the function calls itself
    - Ex. Fibonacci (2^n)
- Space
  - Think about heap memory AND stack memory

But Exponential Is Bad??!?

- Yes! Exponential is bad. But…

- For the coding interview, interviewers may accept an exponential solution.
- Example:
  - If you can solve the problem brute force, and know you can do it better in a DP version, say this to your interviewer. Ask what they would like you to do, because you know that there are two ways, but doing both would take too long.
  - Generally, they will tell you what they want.

But knowing how to do better than exponential is pretty cool.

- You can impress your interviewer if you know how to apply Dynamic Programming concepts to a problem

Recursion: How to Think

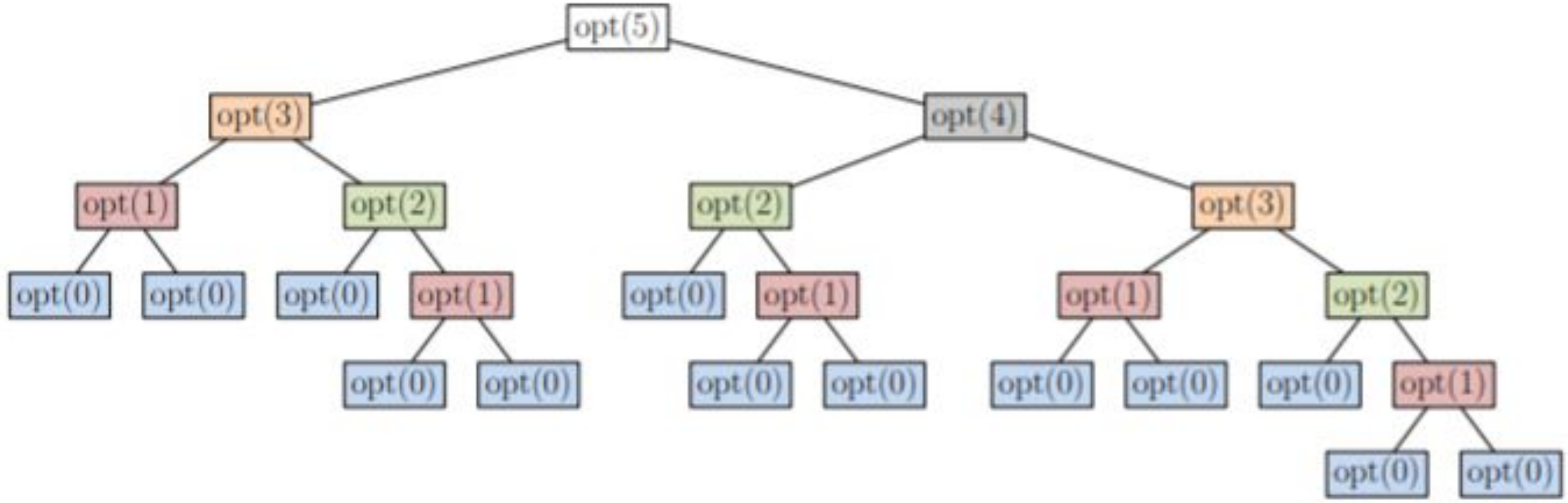
Recursion: How to Think

- Think about which “path” to take!
- If you hear the word “permute,” etc, then that means recursive solution

- If you are taking many of the same paths, you should DP it!

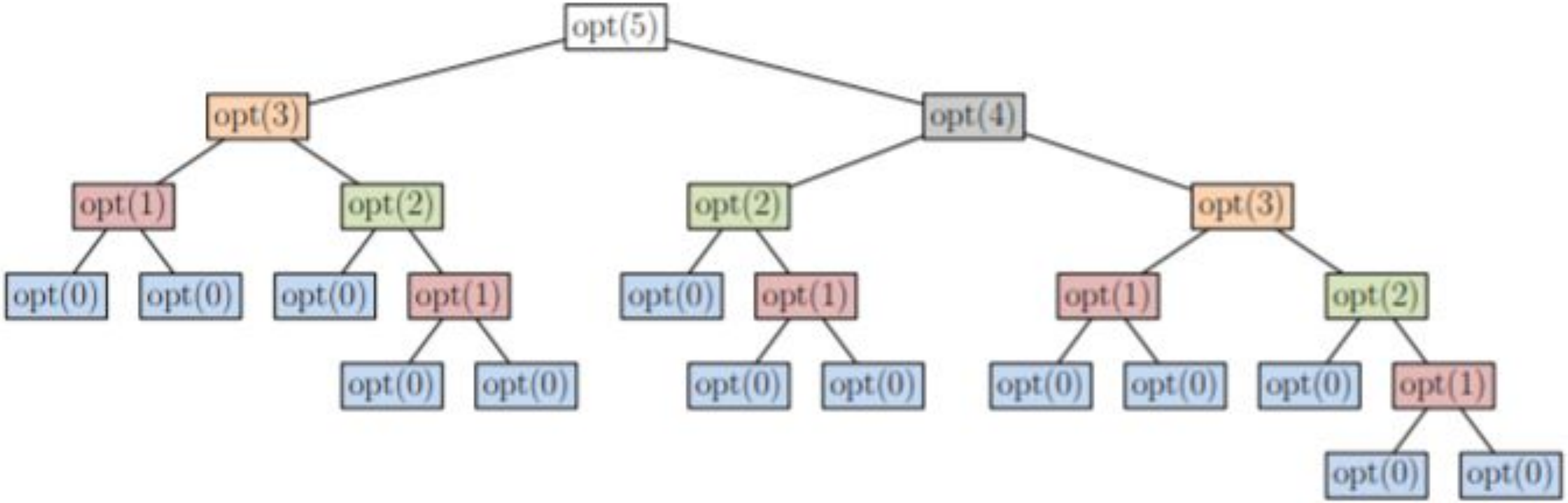
Fibonacci!

- Last week, we discussed the Fibonacci Sequence.
- F(n) = F(n - 1) + F(n - 2)



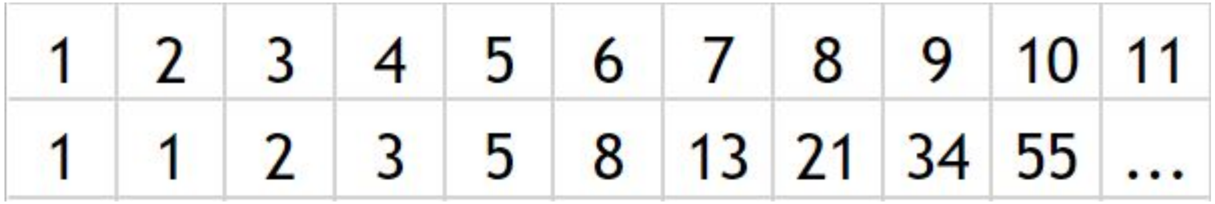
Fibonacci!

- You can see that we take opt(4) first. That already computes opt(3), opt(2), and opt(1). We can just reuse these when we compute opt(3) next!!!



Fibonacci

- Save the computed values in an array!



1	2	3	4	5	6	7	8	9	10	11
1	1	2	3	5	8	13	21	34	55	...

- Now, when we want to access previous fibonacci numbers, we only have to access this array, which takes O(1) time!

Dynamic Programming

Two Ways to DP

- Top-Down
  - This approach applies recursion directly to solve the problem
  - Due to the overlapping nature of the subproblems, the same recursive call is made many times
  - You can use *memoization*, which is what we did with the previous fibonacci problem.
  - YOU ASSUME YOU HAVE ALREADY COMPUTED ALL SUBPROBLEMS
  - Starts from the root
- Bottom-Up
  - Problem is still formulated recursively
  - Solution is built iteratively by combining the solutions to subproblems
  - *Tabulation* - The results are stored in a table.
  - If doing fibonacci, you choose to calculate the numbers in this order: fib(2),fib(3),fib(4)...
  - You can also think of it as filling up a table (a form of caching).

Dynamic Programming Formulation

- Find out how you determine the best path
- "I will take this path first, and calculate that value. If that value was better than the value from the other path, then I will take that value."
- This might not make sense… Example!!!

Knapsack Problem

- The Knapsack Problem is best described by the following:

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Try making one decision (or “go down one path”).
Calculate if that decision is better than not doing that choice.
Continue from there.

Knapsack Problem

- All weights are non-negative.
- We want to maximize the total value of the knapsack, given that the entire weight must be less than the constraint
- Calculate this using a Top-Down solution.

- M[0] = 0 (the sum of 0 items)
- M[weight_i] = max(value_i + M[total_weight - weight_i]

- The complexity is O(nW), which is pseudo-polynomial

Knapsack Problem: Code!

```
knapsack(int values[], int weights[], int num, int capacity) {
    // values[] and weights[] start at index 1
    Int memo[ ][ ];
    for j = 0 to capacity do:
        memo[ 0, j ] := 0
    for i = 1 to num do:
        for j = 0 to capacity do:
            if ( weights[ i ]  >  j ) then:
                memo[ i, j ] := memo[ i–1, j ]              //  “  := “ means assignment in pseudo code
            else:
                memo[ i, j ] := max(   memo[ i–1, j ]   ,   ( memo[ i–1, j–w[ i ] ] ) + values[ i ]   )
                                       // ignore it            // take it and add its value
```

Example of a “Knapsack” Type Problem

- Longest Common Subsequence
- Given two sequences of letters, we want to find the longest common subsequence between them.

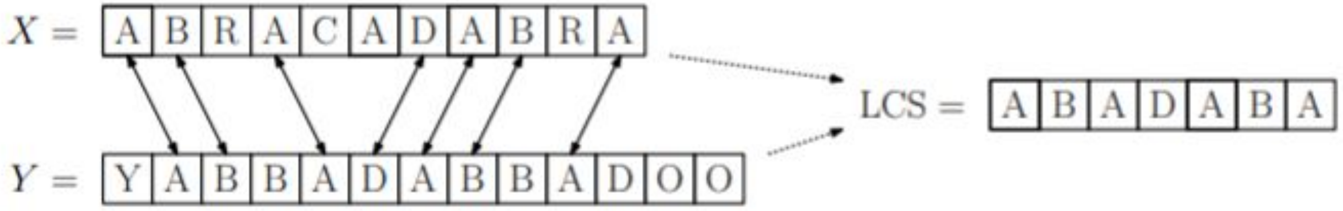


Fig. 1: An example of the LCS of two strings X and Y.

LCS

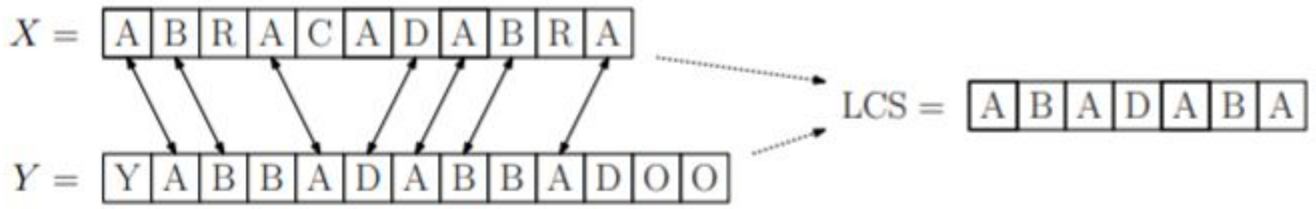


Fig. 1: An example of the LCS of two strings X and Y.

Try the brute force first. What would that be?

Let’s break this down into smaller pieces like DP fashion.

We’ll be thinking Top-Down, for now.

First move???

- What’s our first move in our DP formulation?

- HINT: What is the “bottom case” for this problem?

First move???

- What’s our first move in our DP formulation?

- HINT: What is the “bottom case” for this problem?

  The empty string case!!!

- If either sequence is empty, then the LCS is empty.
  - LCS(i, 0) = LCS(j, 0) = 0

Second move???

- What direction are we going? What would be easiest?

Second move???

- What direction are we going? What would be easiest?

We go backwards, because we want to calculate the longest common subsequence. So, if they both end with things that are similar, then we KNOW that that must be in the LCS. By matching one of the last characters to a previous character, it is only limiting our options.

Second move???

- What direction are we going? What would be easiest?

We go backwards, because we want to calculate the longest common subsequence. So, if they both end with things that are similar, then we KNOW that that must be in the LCS. By matching one of the last characters to a previous character, it is only limiting our options.

- What is our second move, in the best case scenario?

Second move???

- What direction are we going? What would be easiest?

We go backwards, because we want to calculate the longest common subsequence. So, if they both end with things that are similar, then we KNOW that that must be in the LCS. By matching one of the last characters to a previous character, it is only limiting our options.

- What is our second move, in the best case scenario?
- The last characters match! Let’s take them. We know we have to include them in our DP formulation because of the previous statement.

LCS

1. If either sequence is empty, then the LCS is empty.
    a. LCS(i, 0) = LCS(j, 0) = 0
2. Let's suppose that the last characters match
    a. Since both end with the same letter, the LCS MUST contain it.
    b. By matching one of the last characters to a previous character, it is only limiting our options.
    c. Let's TAKE these last two as a pair. Let's calculate the rest of the strings as if we didn't have those characters to begin with.

$$\text{if } (x_i = y_j) \text{ then } \operatorname{lcs}(i, j) = \operatorname{lcs}(i-1, j-1) + 1$$

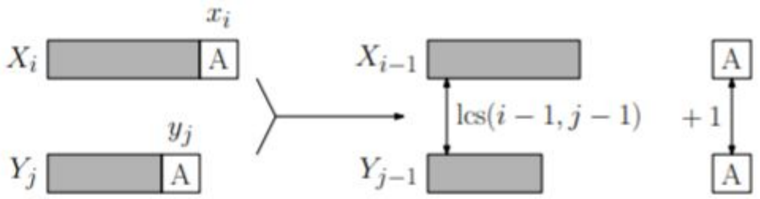


Fig. 2: LCS of two strings whose last characters are equal.

Third case???

- What’s the worst case that could happen?

Third case???

- What’s the worst case that could happen?
- We don’t have a match :(

But what does this mean???

Third case???

- What’s the worst case that could happen?
- We don’t have a match :(

But what does this mean???

- We have to try leaving one string to match in the future, AND ALSO try leaving the other string to match in the future.
- What do we do with this information? What do we want to calculate?

LCS

3. The last characters don't match…
    a. Well, it's either that the first string has another matching character, or the second string…
    b. Let's try both!!!
    c. If they don't match, then LCS(i, j) = max(LCS(i–1, j) , LCS(i, j–1))

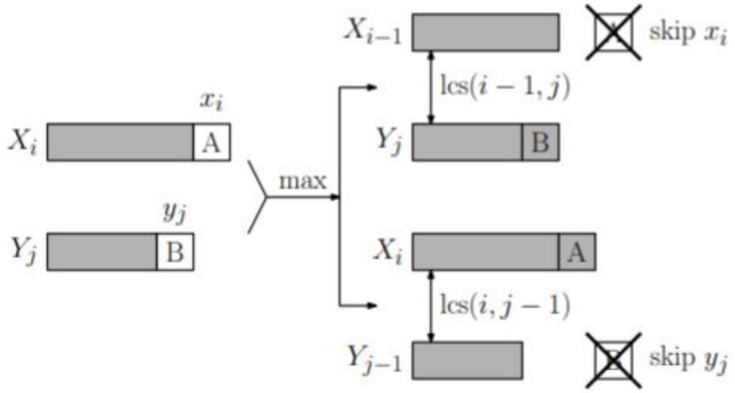


Fig. 3: The possibe cases in the DP formulation of LCS.

All Together Now!

Let’s Try a DP problem together...

A message containing letters from A-Z is being encoded to numbers using the following mapping:

‘A’ → 1
‘B’ → 2
…
‘Z’ → 26

Let’s Try a DP problem together...

Given an encoded message containing digits, determine the **total number of ways** to decode it.

Example:

111	→	“aaa”
111	→	“ka”
111	→	“ak”

How to approach it?

- What’s the first thing we should think of?
  - It should be “This looks easier backwards.”
  - Why is that?

How to approach it?

- What’s the first thing we should think of?
  - It should be “This looks easier backwards.”
  - Why is that?
- What’s the DP way to think about this?

How to approach it?

- What’s the first thing we should think of?
    - It should be “This looks easier backwards.”
    - Why is that?
- What’s the DP way to think about this?
    - We can either take the number we’re at, or we leave it to see what the number before that was.
    - When we get to the empty string, we just decoded the string. Add one to the global counter.

How to approach it?

- What's the first thing we should think of?
    - It should be "This looks easier backwards."
    - Why is that?
- What's the DP way to think about this?
    - We can either take the number we're at, or we leave it to see what the number before that was.
    - When we get to the empty string, we just decoded the string. Add one to DW[i]
    - If it's not valid, then we return and never add that string sequence.
    - Notice how we don't actually know what the memoized values are. We assume they are already calculated, because by the time we need them, they already are!
- Heuristic:
    - DW[0] = 0
    - DW[1] = 1
    - DW[i] = ***max(*** DW[i–1] + 1 , DW[i–2] + 1 ***)***

The Code...

```
public class Solution {
    public int numDecodings(String s) {
        if(s == null || s.length() == 0)  return 0;
        int n = s.length();
        int[] dp = new int[n+1];
        dp[0] = 1;
        dp[1] = s.charAt(0) != '0' ? 1 : 0;
        for(int i = 2; i <= n; i++) {
            int first = Integer.valueOf(s.substring(i-1, i));
            int second = Integer.valueOf(s.substring(i-2, i));
            If (first >= 1 && first <= 9)                dp[i] += dp[i-1];
            If (second >= 10 && second <= 26)      dp[i] += dp[i-2];
        }
        return dp[n];
} }
```

For more DP problems...

- Go to my favorite professor’s website from fall 2017.
- http://www.cs.umd.edu/class/fall2017/cmsc451-0101/lectures.shtml
- Look online!
- During the summer, in preparation for this fall’s class, we’ll be compiling a list of problems from every single category of questions we’ve taught. It’ll be posted to github!

Poll: What should we do next class?

- Dynamic Programming ICI
- Greedy Algorithms & Other Special Algorithms in Computer Science
- Hashing
- Cracking the Coding Interview Book Run Through
- General ICI (any question from any topic)
- Finals Practice
- Guest Speaker (Tom Goldstein)

Reminders

- Fill out feedback form at [ter.ps/fq0](ter.ps/fq0) !