SIG Algorithm Challenges

Week 6: Dynamic Programming

Dane's Contact Email: dziema2@uic.edu
Github: Dane8373

Slides for this week available NOW at
https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week6

Join our Slack channel, #sig_algorithm_chal on the UIC ACM slack (uicacm.slack.com)

Mock Technical Interviews Available!

Sign up here: http://bit.ly/SIG_AC_TECH

Dynamic Programming Introduction

Dynamic Programming is similar to recursion, in the sense that you take a large problem and break it into smaller subproblems and solve those

- The main difference between Dynamic Programming and regular recursion is that the same sub problem occurs multiple times
    - We will see an example of this with the fibonacci numbers

- These repeated sub problems often lead to exponential runtimes, despite the fact that only a polynomial number of unique subproblems exist

- In order to avoid these exponential run times, Dynamic Programming stores the results of sub problems it has already solved
    - Essentially, it may take some time to solve a subproblem the first time we see, but every time after that we will solve it instantly

Dynamic Programming Motivation: Fibonacci

Reminder: the Fibonacci numbers are a series of numbers where the nth number in the series is found by adding the (n-1)th term to the (n-2)th term.
E.g. 0,1, 1, 2, 3, 5,8,13

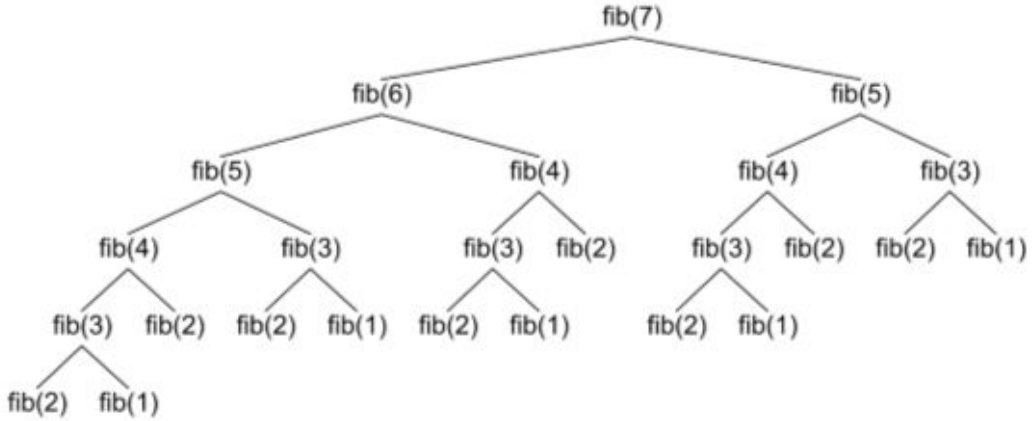
A naive recursive function to calculate the nth fibonacci number is seen below

F(n):

If (n == 0) return 0

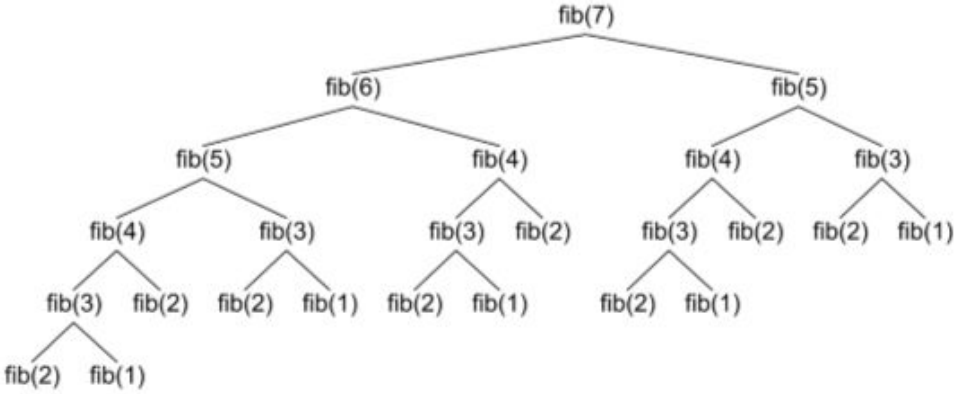
if (n==1) return 1

Return f(n-1) + f(n-2)



An example recursion tree for f(7) is to the right

Dynamic Programming Motivation: Fibonacci



As you can see, this tree gets rather large for even N=7, and subproblems appear many times (f(3) is not a base case, and it appears 5 times for instance). Our naive recursive algorithm will lead to exponential runtime, and f(50) would not be able to be calculated using any standard computer.

However, by augmenting the with dynamic programming, f(50) can be solved extremely quickly

Two Styles of Dynamic Programming

There are two different Dynamic Programming paradigms, Top-down and bottom-up

- Top-Down: To solve the problem of size n, we start by solving all the subproblems that it depends on and store the answers along the way until we have what we need to solve the nth one.
  - Fibonacci Example: To solve F(7), we would first try to solve F(6) and F(5) since F(7) depends on those. To solve F(6), we would solve F(5) and F(4) etc. Once we solve a subproblem once, we store the result.
  - For example: once we solve F(5) to calculate F(6), then we don't need to calculate it again when we use F(5) to get F(7), we just immediately return the value we already calculated

- Bottom Up: To solve problem n, we start by solving the subproblem of size 0, then size 1… all the way up to size 7
  - Example: To calculate F(7), we would first calculate F(0) and F(1), then use those to get F(2), then use F(2) and F(1) to get F(3) etc.

- Often times, newer programmers prefer Bottom up, however it can be difficult to get the bottom up solution without knowing the top down one. It is important to practice both.

Pseudocode for top down Fibonacci

```
Static Container Answers //something used to store the results of the subproblems

F(n):

    If (n == 0)  return 0

    if (n==1) return 1

    If (Answers contains n) return Answers[n]

    Answers[n] = f(n-1) + f(n-2)

    Return Answers[n]
```

Pseudocode for Bottom Up Fibonacci

```
F(n):

    If (n == 0)  return 0

    if (n==1) return 1

    Fib1 = 0, Fib2 = 1, Next = 0

    For (i = 2; i<=n; i++):

        Next = Fib1+Fib2

        Fib1 = Fib2

        Fib2 = next

    Return Next;
```

Identifying Dynamic Programming Problems

- Dynamic programming problems must be able to be broken up into smaller subproblems, and the subproblems must repeat
    - If the subproblems do not repeat, just use a regular divide and conquer approach (dynamic programming will not help, and will just use extra memory)
    - If the problem cannot be subdivided, dynamic programming can't be used.

- Dynamic Programming problems are most often optimization problems
    - E.g. "Find the shortest x", "Find the best y"
    - If your first few attempts at a greedy algorithm for an optimization problem fail, consider trying dynamic programming
    - There are many archetypal dynamic programming optimization problems (Coin change problem, Substring matching problem for example) that others are based off. Know these problems and practice them, and you'll be able to identify and solve their variations in real life.

- Often times, dynamic programming problems can be solved by other means, however dynamic programming makes them faster
    - If you are given a runtime requirement that is significantly lower than the brute force method, consider Dynamic programming

General Tips for Using Dynamic Programming

- Construct a Recurrence relation first
  - The Fibonacci numbers give you their recurrence: F(n) = F(n-1) + F(n-2), however most other problems you have to construct the recurrence yourself
  - The recurrence often times makes the top down approach clear, and knowing the top down approach can help get you a bottom up approach

- Don’t be afraid of the Top Down approach
  - Most students are more comfortable with for loops than recursion, so they shy away from the top down approach
  - It can be hard to build a bottom up approach without knowing the top down approach
  - Interviewers often ask you to show both

- Know both time and space complexities
  - Dynamic programming generally takes a recursive problems with a slow run time, and makes it much faster at the cost of more memory, you need to know both time and memory complexity
  - General guideline for top down approaches: If you have k unique subproblems and each subproblem takes O(f(n)) to solve, then the run time of the total program is (O(k*F(n))), and it takes O(k) extra memory
  - Because of the extra memory requirement, don’t apply dynamic programming if you don’t have to! If the brute force algorithm takes O(n^3) and you write a dynamic programming solution that is O(n^3) with O(n) space, that is bad!

Sample Dynamic Programming Problem

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

```
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

Problem Approach

First, before doing dynamic programming I would try a greedy approach

Idea: Look at pairs of numbers, take the largest of the pair, move on to the next pair

Problem: Consider [2, 5, 100, 10000]. We would first take the 5, but then we would see the 100 and want to take that, and would have to go back and correct our answer. Same with the 10000.

Greedy will not work here because of the restriction of no adjacent numbers, need dynamic programming

First step: Decide how to divide the subproblems

Best Division: Consider only the first k element of the array, call the solution to this problem opt(k)

For opt(k), we can either choose to take the kth element (and thus not take the (k-1)th element) or not take it

This leads to the following recurrence: opt(k) = Max(opt(k-1), Array[k] + opt(k-2))

Pseudocode for This Problem (Top Down)

```
Static Container Answers //something used to store the results of the subproblems

Robber(n):

    If (n == 0)  return 0

    if (n==1) return Array[n]

    If (Answers contains n) return Answers[n]

    Answers[n] = Max (Array[n] + Robber(n-2), Robber(n-1))

    Return Answers[n]
```

Pseudocode for This Problem (Bottom Up)

```
F(n):

    If (n == 0)  return 0

    if (n==1) return Array[n]

    oldOldMax = Array[0], OldMax = Max(oldOldMax, Array[1]), curr = 0

    For (i = 2; i<n; i++):

        curr = Max(OldOldMax + Array[i], oldMax)

        oldOldMax = oldMax

        oldMax = curr

    Return curr
```

Java Code for this problem (Top Down Part 1)

Driver code

```
    public int rob(int[] nums) {
        //initialize our memoization
        answers = new int[nums.length];
        for (int i=0; i<nums.length; i++) {
            answers[i] = -1;
        }
        return helper(nums, nums.length-1);
    }
}
```

Java Code for this problem (Top Down Part 2)

```java
int[] answers; //Stores the best value for a subarray ending at index i
public int helper(int[] nums, int end) {
    //base case, 0 if there are > 0 houses left
    if (end < 0) {
        return 0;
    }
    //return the 1 house if there is only 1
    if (end == 0) {
        return nums[0];
    }
    //if we have already sovled this sub problem, return the answer we found
    if (answers[end] != -1) {
        return answers[end];
    }
    //either we take the house at index end, or we do not
    //max value we can get without taking this house is just the max of the index before
    int dontTake = helper(nums, end-1);
    //max value we can get is the max value from 2 indexes ago + the current house
    //(2 indexes ago because we can't take adjacent houses)
    int take = nums[end] + helper(nums,end-2);
    //pick the max
    answers[end]= Math.max(dontTake,take);
    return answers[end];
}
```

Java Code for this problem (Bottom up)

```java
public int rob(int[] nums) {
    //bunch of base cases since the algorithm only works for
    //arrays of size > 2
    if (nums.length == 0) {
        return 0;
    }
    else if (nums.length == 1) {
        return nums[0];
    }
    int oldMax = Math.max(nums[0],nums[1]); //best we can get without taking the current house
    if (nums.length == 2) {
        return oldMax;
    }
    int oldOldMax = nums[0]; //best we can get from 2 indexes ago
    int curr = 0; //amount we can get with taking current house
    for (int i=2; i<nums.length; i++) {
        //current max = max of 2 indexes ago + current house
        curr = oldOldMax + nums[i];
        //take the max of either taking this house, or the max without it
        curr = Math.max(curr,oldMax);
        oldOldMax = oldMax;
        oldMax = curr;
    }
    return curr;
}
```

Dynamic Programming Problems to Work On

Links to Problems 1-4 can be found on my Github account (Dane8373)

Direct Link: https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week6

Note: These problems have roughly similar difficulty

1) A knight (from the game of chess) is placed on a standard telephone number pad. Count the number of different phone numbers this knight can dial given K moves

2) You are given two string A and B, you can capitalize or remove any lowercase letter from A. Determine whether or not it is possible to make A and B equivalent.

3) Alex and Lee play a game with piles of stones. There are an even number of piles arranged in a row, and each pile has a positive integer number of stones `piles[i]`.

Alex and Lee take turns, with Alex starting first. Each turn, a player takes the entire pile of stones from either the beginning or the end of the row. This continues until there are no more piles left, at which point the person with the most stones wins.

Assuming Alex and Lee play optimally, return `True` if and only if Alex wins the game.

4) The coin change problem: Given a set of coins with various values, count the number of ways you can make change for some number N