Competitive Programming and Mathematics Society

Introduction to Dynamic Programming

CPMSoc

Welcome

CPMSOC

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- Mathematics workshops will run every odd-numbered week (3, 5, 7, ...)
- Programming ones will run every even-numbered week (4, 6, 8, ...)
- We have lots of other events too (e.g. Integration Bee, Board Games next week)
- Slides will be uploaded on our website (unswcpmsoc.com)
- Hoodie sales will be released very soon!!!

Attendance form :D

Workshop Overview

- Introduction
- Example
  - Fibonacci
  - Optimising Fibonacci
- Types of Dynamic Programming
- Further Examples to Try

Introduction

- **What is Dynamic Programming (DP)?**
  Put simply, it's a method for solving complex problems by breaking them down into simpler subproblems.

  Let's have a look at an example!

Fibonacci Sequence

The Fibonacci sequence goes like this 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 . . .

- We can see that every number in the series is the sum of the previous two numbers except the first two numbers.
- The first number is 0 and the second number is 1. They are fixed. The third number is 1 which is the sum of the previous two numbers 0 and 1.

Now lets take the function fib(n) which will return the nth number of the Fibonacci sequence. A simple formula for it will look something like this:

- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2) if n > 1

Let's put this into a function so we can calculate any Fibonacci number!

Fibonacci Continued...

Lets start by breaking the problem down into simpler subproblems. We'll start with the 'base cases' which we know already as:

- fib(0) = 0
- fib(1) = 1

And similarly, we can break the problem of finding the *n*th Fibonacci number down into simpler subproblems with the help of the formula

- fib(n) = fib(n-1) + fib(n-2) if n > 1

Fibonacci Continued...

Thus, a recursive algorithm will look something like this:

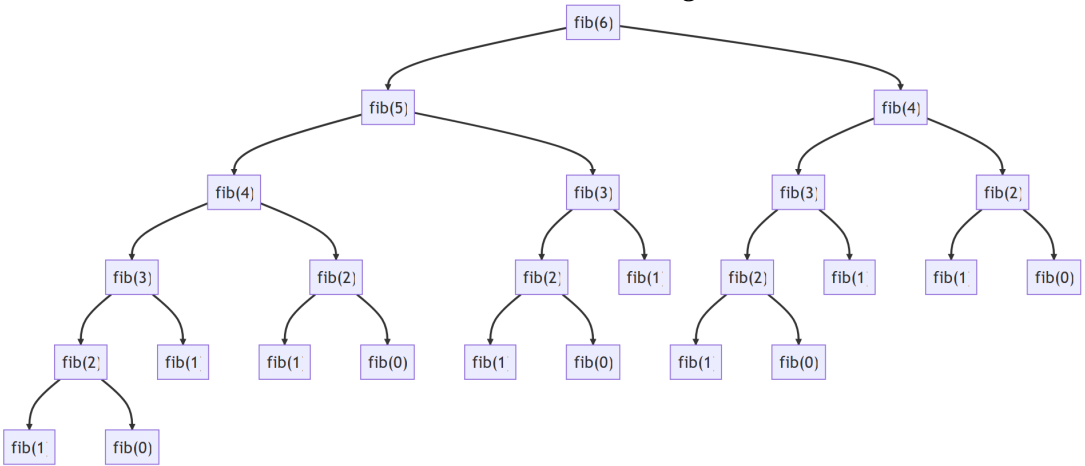
```
def Fib(n) {
  // Base Cases
  if (n == 0) {
    return 0;
  }

  if (n == 1) {
    return 1;
  }
  // Recursive Cases
  return Fib(n - 1) + Fib(n - 2);
}
```

And when we call fib(6), we would get the output: 8 (working as intended)
However, it has some serious issues!

Question Time: What's the issue and what can we do about it?

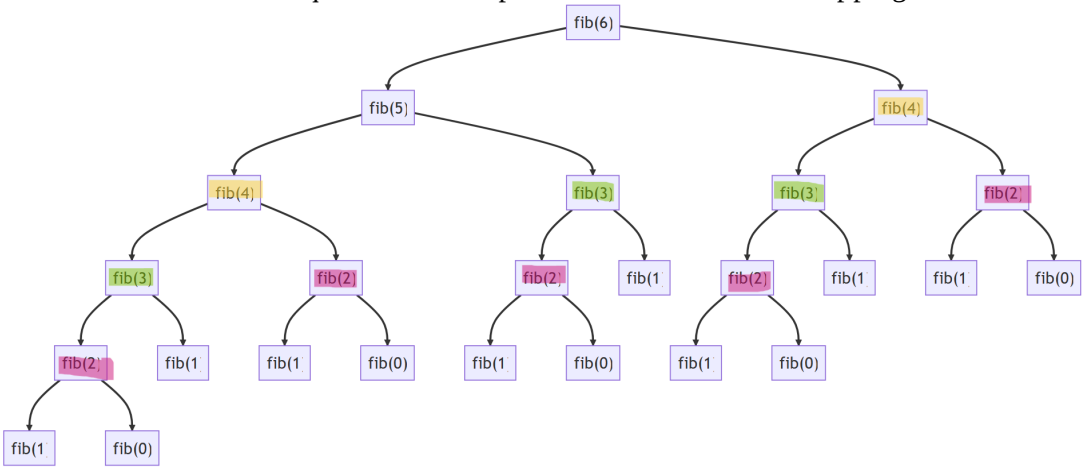
The Recursion Tree

Let's start analysing what's happening when we call the function fib(6). The recursive nature of our function will call functions in the following manner -



Recursion Tree Continued...

It is clear that there are quite a few subproblems which are overlapping -



Recursion Tree Continued...

- fib(4) is being called 2 times
- fib(3) is being called 3 times
- fib(2) is being called 4 times

In general, we want to make sure that overlapping subproblems are solved (calculated) using our function only once in dynamic programming - calculating them is an expensive task!

In order to achieve this goal, we will save the solutions to overlapping subproblems in a data structure. This is called "memoization" or "caching"!

The DP Approach

We can store the results of solved subproblems inside a data structure like an array, and the function will check if a subproblem has already been solved before or not.
If it has been solved before, we won't solve it again, we just used the already calculated value.
Lets take an array memo[] and populate it with a value that can't ever occur, say... -1.

Now for fib(n) we will check if memo[n] is equal to -1 or not. If it isn't then we can continue solving it and if it is we can just return the already calculated value.

Lets take a look at our new code!

Improved Fibonacci

```
def Fib(n) {
  if (n == 0) {
    return 0;
  }

  if (n == 1) {
    return 1;
  }

  if (memo[n] != -1) {
    return memo[n];
  }

  memo[n] = Fib(n - 1) + Fib(n - 2);
  return memo[n];
}
```

Comparing Complexities

Talk with the people around you and discuss what the effect of Memoization has done to the complexity of the program!

Some things to take note are:

- What was the time complexity of the original program?
- What is the new time complexity and how much has it improved?

Comparing Complexities Continued...

In the recursive approach for every value, two function are called. For example, fib(6) calls fib(5) and fib(4). Fib(5) calls fib(4) and fib(3), etc.
Hence, the time complexity is $O(2^n)$ which is very slow!

Now, in the dynamic programming approach however, we are only solving each subproblem once e.g. fib(0), fib(1), fib(2), etc. And since we have n subproblems the complexity comes out to be just $O(n)$. A huge improvement!

Structure of a DP

So from that example, we can now extract some general steps required to solve DP problems **Steps for Solving DP Problems**

1. Define subproblems
2. Formulate a recurrence that relates subproblems
3. Recognise and solve the original problem
4. Define the base casess

Each step is very important!

Types of Dynamic Programming

CPMSOC

Recursive DP

- Our Fibonacci example is an example of this type of DP
- Called "top-down" DP
  - You start from the 'top', and do function calls 'down'
- Very similar to regular recursion format
- The DP function will record its previously calculated answer
- This is stored in your DP cache

Iterative DP

- Prefix sum is an example of this type of DP
- Called "bottom-up" DP
  - You start from the 'bottom' case, and use that to build 'up' other cases
- Instead of recursive function calls, use a for loop
- Generate answers in order
- All answers should be stored in DP cache

Maximum Non-Adjacent Subarray Sum

CPMSOC

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- Find the maximum subset sum in an array, where you cannot select any 2 adjacent elements
  - For example, given the array [4, 1, 1, 4], find the maximum sum of elements following the constraints
  - Answer: 8, picking the two 4s
- Brute force approach
  - Horrible time complexity, but also painful to code up
- DP (recursive)
- DP (iterative)

Coin Change Problem

- Given a few types of coins, make a certain amount of change with the least number of coins
  - For example, given 1c, 3c and 4c coins, make 17 cents with as few coins as possible
  - Answer: 4x 4c coins and 1x 1c coin.
- Brute force approach
  - Still a poor time complexity, but a bit easier to code - decent for smaller input sizes
- DP (recursive)
- DP (iterative)

Coin Combinations 1

- Given a few types of coins, how many permutations are there to make a certain amount of change.
  - For example, given 2c and 6c coins, make 10 cents.
  - Answer: 4 (2+2+6, 2+6+2, 6+2+2, 2+2+2+2+2)
- Brute force approach
  - Still a poor time complexity, but a bit easier to code - decent for smaller input sizes
- DP (recursive)
- DP (iterative)
- Coin Combinations 2
  - Same problem but with combinations (so order doesn't matter)
  - Answer to previous example: 2 (2x2c + 1x6c, 5x2c)

Elevator Rides

- Given n people, with the ith person having a weight of w_i, who want to ride an elevator with a weight capacity of x, find the minimum number of trips to take everyone to the top.
  - For example, there are people with weight 60 kg, 70kg and 80 kg, and the elevator has a capacity of 130 kg
  - Answer: the minimum number of trips is 2. One trip has 60+70=130 kg, the second has 80 kg
- Brute force approach
  - Poor time complexity, difficult to code
- DP (recursive)
- DP (iterative)

Attendance form :D

Feedback form :D

CPMSOC

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

Further events

Please join us for:

- Maths workshop next week
- Programming workshop in two weeks
- Integration Bee next Wednesday
- Board Games next Thursday
- Hoodie sales will be released very soon!!!