UQCS Competitive Programming Group
Week 4
Dynamic Programming

Matthew Low

UQ Computing Society

Welcome to CPG!

UQCS **Competitive Programming Group**.

Algorithms, data structures and, most of all, problem solving.

We will try to be beginner-friendly, but basic programming knowledge is expected. COMP3506 (or any algorithms course) would also be helpful for tackling these problems.

Last week!

We learnt about graphs and basic traversals (breadth-first and depth-first search).

This week!

Dynamic programming!

What is dynamic programming?

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.[1]

[1] https://en.wikipedia.org/wiki/Dynamic_programming

What is dynamic programming?

In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

Why dynamic programming

Dynamic programming is typically used when you want to solve the same subproblem again and again. Essentially, if we can reduce the large problem to subproblems, it makes sense to solve it with dynamic programming.

Fibonacci numbers

Consider the Fibonacci sequence

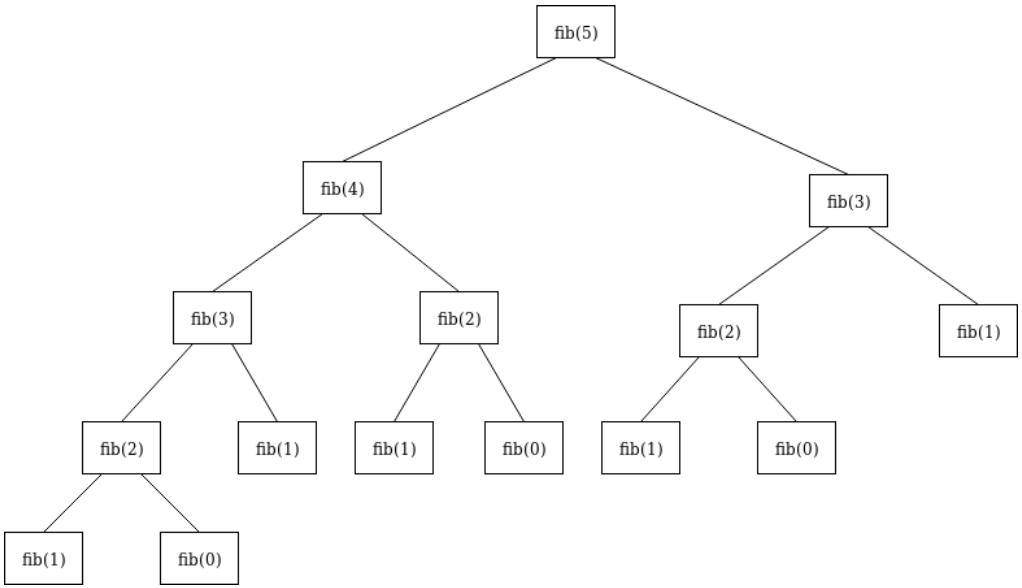
$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

If you write a recursive program to compute the Fibonacci sequence, then you would have something like this:

```
def fib(self, N: int) -> int:
    if N <= 1:
        return N
    return fib(N-1) + fib(N-2)
```

Fibonacci numbers

But looking at the recursion tree, this can be quite slow:



Look at the amount of times we call `fib(1)`! So... why not just store the results of each `fib(i)` call (break it down into subproblems)? This is called **memoization**.

Memoised Fibonacci numbers

```
def fib(self, N: int) -> int:
    if N <= 1:
        return N
    return memoize(N)

def memoize(self, N: int) -> {}:
    cache = {0: 0, 1: 1}

    for i in range(1, N):
        cache[i+1] = cache[i] + cache[i-1]

    return cache[N]
```

Now some problems:

Easy: Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

`https://leetcode.com/problems/climbing-stairs/`

Easy: Maximum Subarray

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

https://leetcode.com/problems/maximum-subarray/

Medium: Coin Change

You are given coins of different denominations and a total amount of money `amount`. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

`https://leetcode.com/problems/coin-change/`

Hard: Edit Distance

Given two words `word1` and `word2`, find the minimum number of operations required to convert `word1` to `word2`.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

`https://leetcode.com/problems/edit-distance/`