AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 04

CSC2211 Algorithms

Summer 2019-2020

TITLE

An Introduction to Dynamic Programming

PREREQUISITE

- Have a clear understanding of Divide and Conquer Approach
- Have a basic idea about Recursive Functions
- Have a basic idea about Recurrence Relation and Complexity Analysis

OBJECTIVE

- To understand Fibonacci Series and its implementation using both D&C and DP
- To know about 0-1 Knapsack and its implementation using DP
- To be able to find complexity from the code

THEORY

DYNAMIC PROGRAMMING

Dynamic programming (usually referred to as DP) is a very powerful technique to solve a class of problems. The idea is very simple, if you have solved a problem with the given input, then save the result for future reference, to avoid solving the same problem again. If the given problem can be broken up in to smaller sub problems and these smaller sub problems are in turn divided into still smaller ones, and in this process, if you observe some over lapping sub problems, then it is a big hint for DP. Also, the optimal solutions to the sub problems contribute to the optimal solution of the given problem (referred to as the Optimal Substructure Property).

Bottom Up:

Analyze the problem and see the order in which the sub problems are solved and start solving from the trivial sub problem, up towards the given problem. In this process, it is guaranteed that the sub problems are solved before solving the problem. This is referred to as Dynamic Programming.

The key steps in a dynamic programming solution are

- **Characterize** the optimality - formally state what properties an optimal solution exhibit
- **Recursively** define an optimal solution - analyze the problem in a top-down fashion to determine how sub problems relate to the original
- **Solve** the sub problems - start with a base case and solve the sub problems in a bottom-up manner to find the optimal value
- **Reconstruct** the optimal solution - (optionally) determine the solution that produces the optimal value

Thus, the process involves breaking down the original problem into sub problems that also exhibit optimal behavior. While the sub problems are not usually independent, we only need to solve each sub problem once and then store the values for future computations.

FIBONACCI NUMBER

One of the most beautiful occurrences of mathematics in nature is Fibonacci numbers. Named after Leonardo Fibonacci, this series dates to ancient history with origins in Indian mathematics.

The beauty of the series is enhanced by its simple definition, x(n)=x(n-1) + x(n-2)

So starting from x (0) = 0 and x(1) = 1, the series progresses infinitely as: 0,1,1,2,3,5,8,13...

Now the question is how do we go about finding the n^{th} Fibonacci number?

This problem can be solved recursively but the complexity becomes exponential in this method.

Repeating the exact same computation many times is a cry for help - We can save all this effort by computing only once and re-using every other time. We can achieve that by storing the results the first time we compute them and instead of doing the same computation again, we can just look it up from the stored memory. This technique of avoiding repeated calculation of results by storing them is called memoization and is used in Dynamic Programming.

The Pseudo-code to find the nth Fibonacci number without memorization is given below:

```
int Fib(int n)
{
    if(n<=1)
      return n;
   else
      return Fib(n-1)+ Fib(n-2);
}
```

The Pseudo-code to find the nth Fibonacci number by memorization is given below:

```
Fib(n)
{
    if (n == 0)
        return M[0];

    if (n == 1)
        return M[1];

    if (Fib(n-2) is not already calculated)
        call Fib(n-2);

    if(Fib(n-1) is already calculated)
        call Fib(n-1);

    //Store the ${n}^{th}$ Fibonacci no. in memory & use previous results.
    M[n] = M[n-1] + M[n-2]

    Return M[n];
}
```

LAB WORK

Problem 1: Find the time complexity of finding nth Fibonacci number from the code. For the following three cases: Recursive without memoization, Recursive with memoization (DP), memoization iterative version.

Problem 2: Implement the code for 0-1 Knapsack Problem. Find the time complexity of the Algorithm from code.