**Overlapping Subproblems Property in Dynamic Programming | DP-1**

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.

In this post, we will discuss the first property (Overlapping Subproblems) in detail. The second property of Dynamic programming is discussed in the next post i.e. [Set 2](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/).  
1)**Overlapping Subproblems**  
2) **Optimal Substructure**

**1) Overlapping Subproblems:**   
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, [Binary Search](https://www.geeksforgeeks.org/binary-search/) doesn’t have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

Recursion tree for execution of *fib(5)*

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:   
a) Memoization (Top Down)   
b) Tabulation (Bottom Up)

**a) Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Following is the memoized version for the nth Fibonacci Number.

**b) Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.

Following is the tabulated version for nth Fibonacci Number.

**public** **class** Fibonacci {

**int** fib(**int** n)

    {

**int** f[] = **new** **int**[n + 1];

        f[0] = 0;

        f[1] = 1;

**for** (**int** i = 2; i <= n; i++)

            f[i] = f[i - 1] + f[i - 2];

**return** f[n];

    }

**public** **static** **void** main(String[] args)

    {

        Fibonacci f = **new** Fibonacci();

**int** n = 9;

        System.out.println("Fibonacci number is"

                           + " " + f.fib(n));

    }

}