Dynamic Programming (DP)

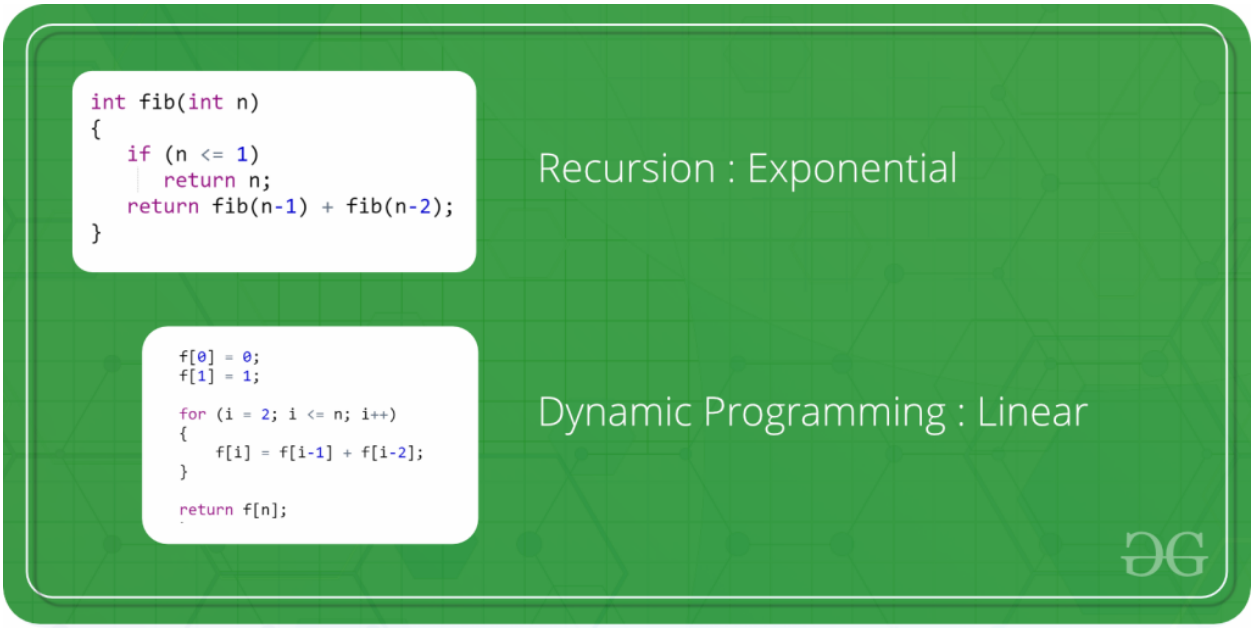
What is Dynamic Programming (DP) ?

> ***Those who cannot remember the past are condemned to repeat it.***

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Example:

if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



Types of approaches to solve Dynamic Programming Problems:

There are two main approaches to solving dynamic programming problems:

1. **Top-Down Dynamic Programming (Memoization):**
    a. In the top-down approach, we start with the original problem and break it down into smaller subproblems. We then recursively solve each subproblem, storing the result in a cache or table as we go. When we encounter a subproblem that we've already solved, we simply look up the answer in the cache instead of recomputing it. This approach is also known as memoization.
2. **Bottom-Up Dynamic Programming (Tabulation):**
    a. In the bottom-up approach, we start with the smallest subproblems and work our way up to the original problem. We store the results of each subproblem in a table or cache and use those results to solve larger subproblems. By the time we get to the original problem, we've already solved all the subproblems we need and can simply look up the answer in the table. This approach is also known as tabulation.

Let’s take an example of Fibonacci number to understand DP:

The Fibonacci numbers are a sequence of numbers in which each number is the sum of the two preceding numbers. The sequence starts with 0 and 1, and the next number in the sequence is the sum of the two previous numbers.

The first few numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … so on.

Let’s say we have to find the 5th Fibonacci Number. Now, if you try to solve this using recursion, the recurrence relation and the recursive code is given below,

```
// Recurrence Relation of Fibonacci Number is:
Fib(n) = Fib(n-1) + Fib(n-1);
```

```
// Recursive Code for finding nth Fibonacci Number.

/* Recursive Function to find nth term of fibonacci series. */
// Time Complexity :- O(2^n)
// Space Complexity :- O(n), recursive stack space.
int fibRecursive(int n)
    {
        // The first term of the Fibonacci series is 0,
```

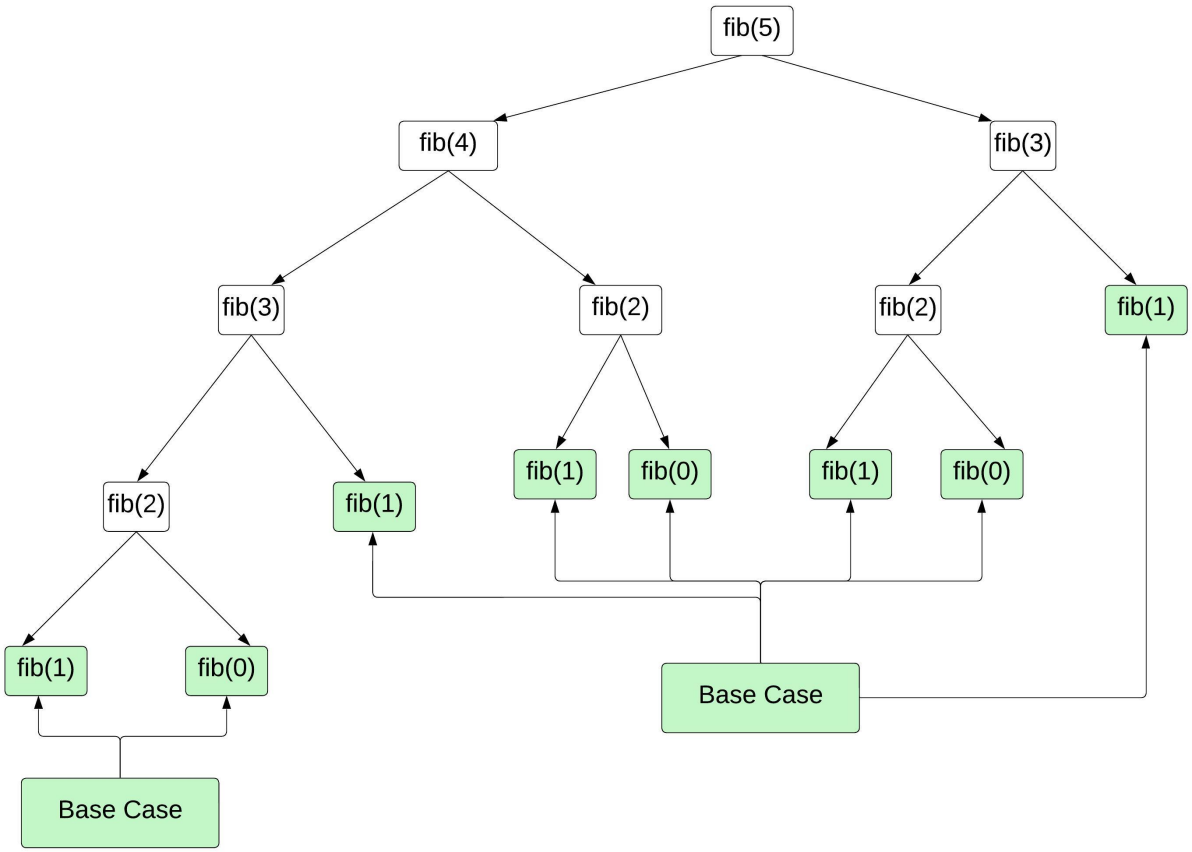
```
        // the second term of the Fibonacci series is 1
        // and the next term of the fibonacci series
        // is the summation of previous 2 terms.

        // Base Condition.
        if (n == 0)
        {
            // The first term of the Fibonacci series is 0.
            return 0;
        }
        if (n == 1)
        {
            // the second term of the Fibonacci series is 1.
            return 1;
        }

        // Recursive Case.
        // The next term of the fibonacci series is the summation of previous 2 terms.
        int nextTerm = fibRecursive(n - 1) + fibRecursive(n - 2);
        return nextTerm;
    }
```

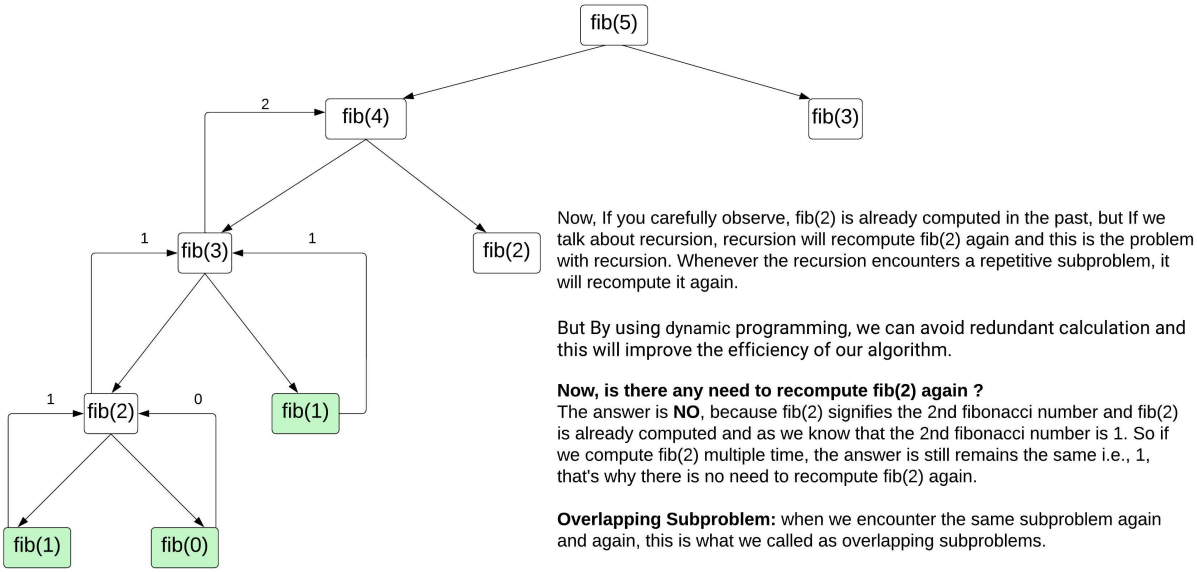
Now, if you draw the Recursive Tree for finding the 5th Fibonacci number, the Recursive Tree would look like this,

Recursive Tree



Now let’s see how can we optimize this using Dynamic Programming,

Recursive Tree



Now, If you carefully observe, fib(2) is already computed in the past, but If we talk about recursion, recursion will recompute fib(2) again and this is the problem with recursion. Whenever the recursion encounters a repetitive subproblem, it will recompute it again.

But By using dynamic programming, we can avoid redundant calculation and this will improve the efficiency of our algorithm.

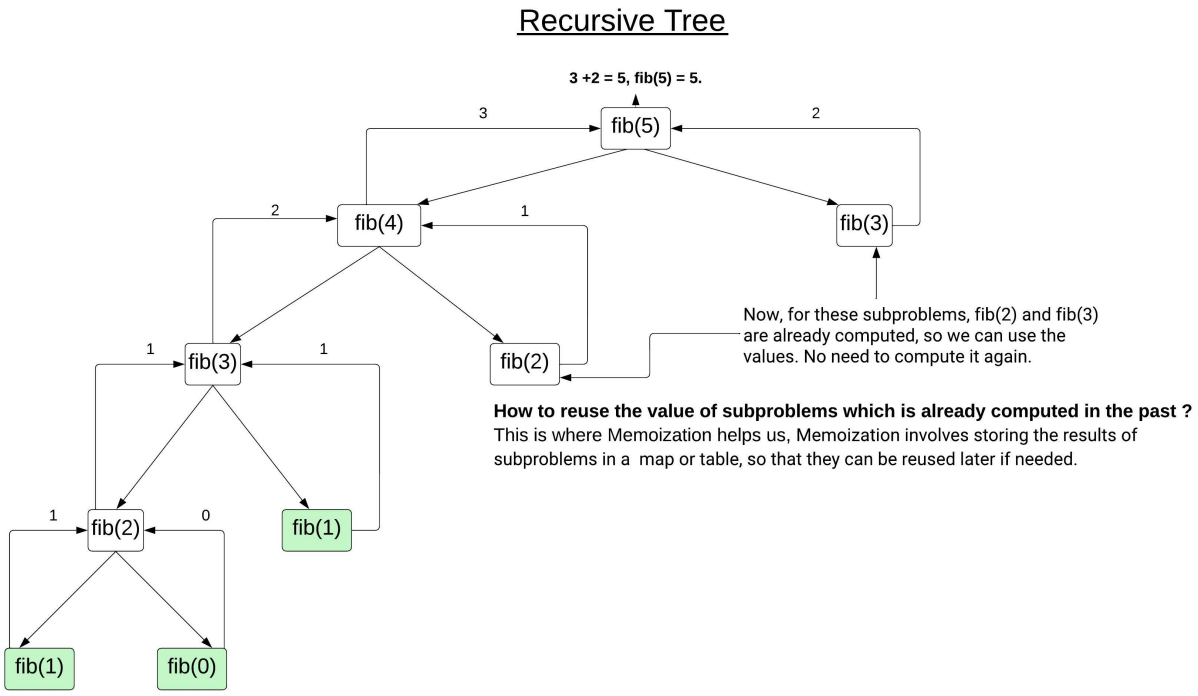
Now, is there any need to recompute fib(2) again ?
The answer is **NO**, because fib(2) signifies the 2nd fibonacci number and fib(2) is already computed and as we know that the 2nd fibonacci number is 1. So if we compute fib(2) multiple time, the answer is still remains the same i.e., 1, that's why there is no need to recompute fib(2) again.

Overlapping Subproblem: when we encounter the same subproblem again and again, this is what we called as overlapping subproblems.

Overlapping Subproblem: when we encounter the same subproblem again and again, this is what we called an overlapping subproblems.

Overlapping subproblems occur when a problem can be divided into smaller subproblems, and some of those subproblems are solved repeatedly during the computation of the larger problem. This means that the same subproblem is encountered multiple times, often with different inputs or parameters, which can result in redundant calculations and slower algorithms.

In dynamic programming, we can address overlapping subproblems by using techniques like memoization or tabulation. Memoization involves storing the results of subproblems in a lookup table or cache, so that they can be reused later if needed. Tabulation, on the other hand, involves building a table of solutions to subproblems in a bottom-up fashion, and using those solutions to solve the larger problem.



Now, you can clearly see, using the concepts of DP we save a lot of time by storing the results of previously solved subproblems and reusing them when needed.

Now, if you carefully observe, we have only solved 5 subproblems for finding 5th Fibonacci number. So the time complexity will reduced from exponential O(2^n) to linear O(n).

So, using Dynamic Programming, we can reduce the time complexity of finding the nth Fibonacci number from exponential O(2^n) to linear O(n).

What is Memoization ?

Now, if you carefully observe, we have only solved 5 subproblems for finding 5th Fibonacci number. This is because each subproblem depends only on the previous two subproblems, so we can reuse the results of those subproblems to compute the current subproblem. As a result, the total number of subproblems we need to solve is proportional to n, which means the time complexity is linear in terms of n.

So, I can say that, if we have to find the nth Fibonacci number, then we only have to solve n subproblem.

The idea behind memoization is to store the results of previously computed subproblems in a cache so that we can reuse them later instead of recomputing them.

For the Fibonacci sequence, we can create an array of size (n+1) and initialize the first two values to 0 and 1. We can then use a loop to fill in the rest of the array by adding the previous two values together. When we need to compute a Fibonacci number, we can first check if it is already stored in the array. If it is, we simply return the cached result. If it is not, we compute it using the cached results and store the result in the array for future use. This approach saves a lot of time by avoiding redundant computations and reducing the time complexity of the algorithm from exponential to linear.

Why the size of array is n+1 ?

The size of the array used for memoization is n+1 because we need to store the values of all subproblems from 0 to n. For example, if we want to compute the 5th Fibonacci number, we need to compute and store the values of all Fibonacci numbers from 0 to 5, which requires an array of size 6 (0, 1, 1, 2, 3, 5). Similarly, if we want to compute the nth Fibonacci number, we need to compute and store the values of all Fibonacci numbers from 0 to n, which requires an array of size (n+1).