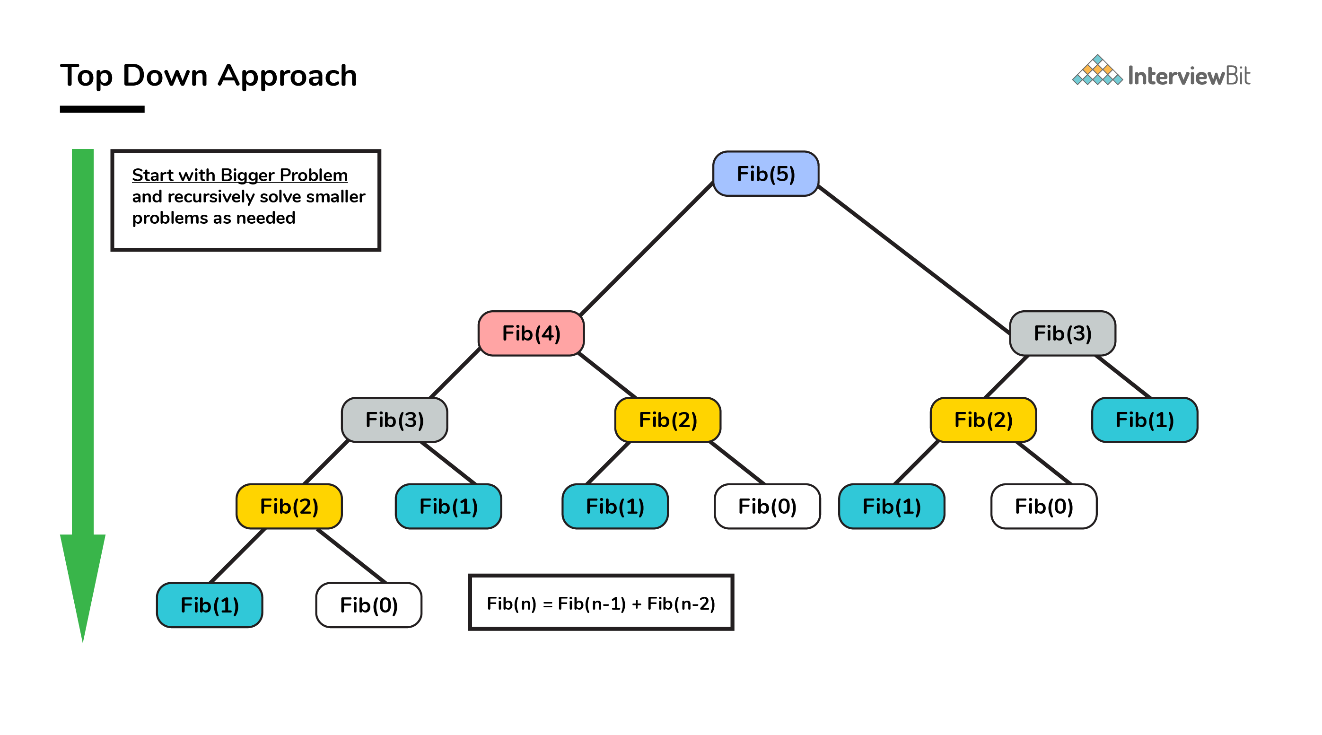
<https://www.interviewbit.com/courses/programming/topics/dynamic-programming>

## Dynamic Programming Methods

We shall continue with the example of finding the nth Fibonacci number in order to understand the DP methods available. We have the following two methods in DP technique. We can use any one of these techniques to solve a problem in optimised manner.

### Top Down Approach (Memoization)

* Top Down Approach is the method where we solve a bigger problem by recursively finding the solution to smaller sub-problems. 
* Whenever we solve a smaller subproblem, we remember (cache) its result so that we don’t solve it repeatedly if it’s called many times. Instead of solving repeatedly, we can just return the cached result.
* This method of remembering the solutions of already solved subproblems is called **Memoization**.

#### Pseudo Code and Analysis

**Without Memoization**

1. **Think of a recursive approach to solving the problem.** This part is simple. We already know Fib(n) = Fib(n - 1) + Fib(n - 2).
2. **Write a recursive code for the approach you just thought of.**

/\*\*

\* Pseudo code for finding Fib(n) without memoization

\* @Parameters: n : nth Fibonacci term needed

\*

\*/

int Fib(int n) {

if (n <= 1) return n; //Fib(0)=0; Fib(1)=1

return Fib(n - 1) + Fib(n - 2);

}

The time complexity of the above approach based on careful analysis on the property of recursion shows that it is essentially **exponential in terms of n** because some terms are evaluated again and again.

**With Memoization**

1. Save the results you get for every function run so that if Fib(n) is called again, you do not recompute the whole thing.
2. Instead of computing again and again, we save the value somewhere. This process of remembering the values of already run subproblem is called memoization.
3. Lets declare a global variable memo then.

/\*\*

\* Pseudo code for finding Fib(n) with memoization

\* @Parameters: n : nth Fibonacci term needed

\*

\*/

int memo[100] = {0};

int Fib(int n) {

if (n <= 1) return n;

// If we have processed this function before,

// return the result from the last time.

if (memo[n] != 0) return memo[n];

// Otherwise calculate the result and remember it.

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

return memo[n];

}

1. Let us now analyze the space and time complexity of this solution. We can try to improve this further if at all it is possible.
   * Lets look at the space complexity first.
     + We use an array of size n for remembering the results of subproblems. This contributes to a space complexity of O(n).
     + Since we are using recursion to solve this, we also end up using stack memory as part of recursion overhead which is also O(n). So, overall space complexity is O(n) + O(n) = 2 O(n) = O(n).
   * Lets now look at the time complexity.
     + Lets look at Fib(n).
     + When Fib(n - 1) is called, it makes a call to Fib(n - 2). So when the call comes back to the original call from Fib(n), Fib(n-2) would already be calculated. Hence the call to Fib(n - 2) will be **O(1)**.
     + Hence,
     + T(n) = T(n - 1) + c where c is a constant.
     + = T(n - 2) + 2c
     + = T(n - 3) + 3c
     + = T(n - k) + kc
     + = T(0) + n \* c = 1 + n \* c = O(n)

Thanks to Dynamic Programming, **we have successfully reduced a exponential problem to a linear problem**.

#### Implementation

/\* Java Program to find Nth Fibonacci Number using Memoization \*/

public class Fibonacci

{

final int NIL = -1;

int memo[] = new int[100];

/\* Function to initialize NIL values in memo \*/

void initializeMemo()

{

for (int i = 0; i < 100; i++)

memo[i] = NIL;

}

int fib(int n)

{

if (memo[n] == NIL)

{

if (n <= 1)

memo[n] = n;

else

memo[n] = fib(n-1) + fib(n-2);

}

return memo[n];

}

//Main Driver class

public static void main(String[] args)

{

Fibonacci fibonacci = new Fibonacci();

int n = 10;

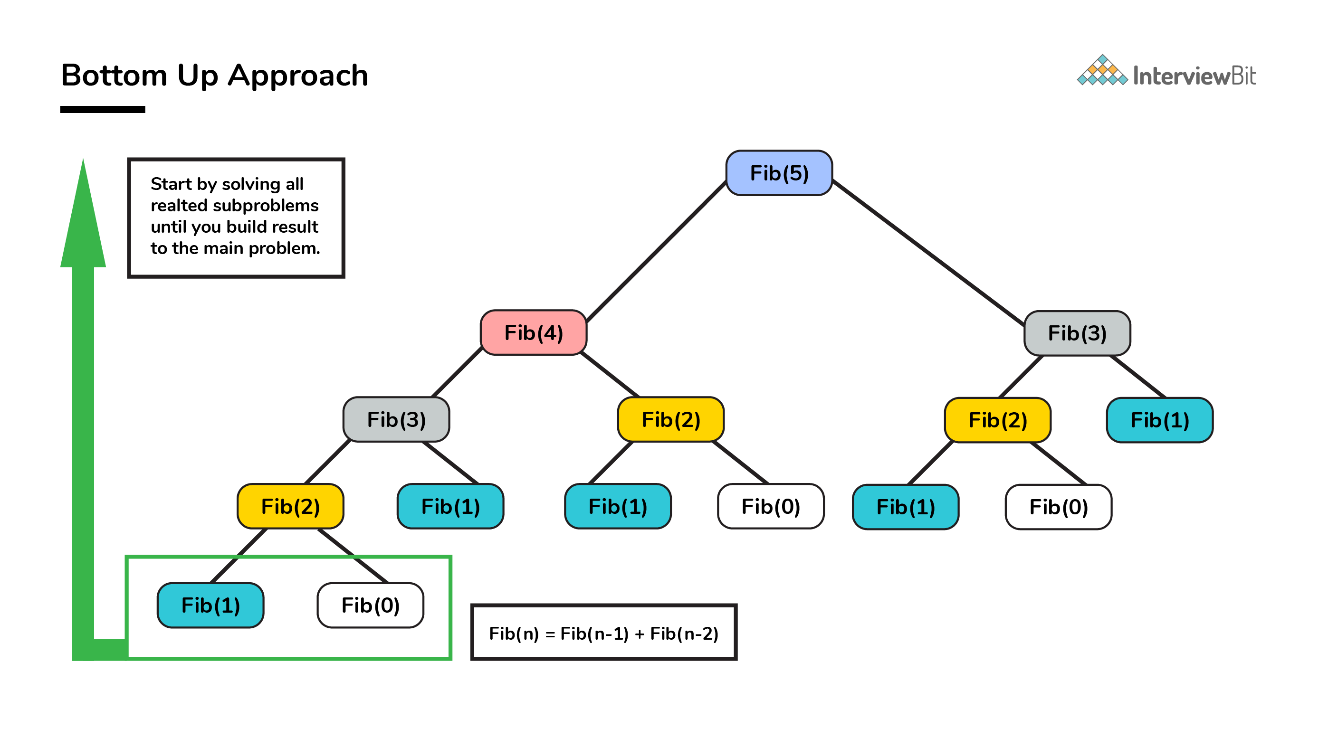
fibonacci.initializeMemo();

System.out.println("Fibonacci number : " + fibonacci.fib(n));

}

}

### Bottom Up Approach (Tabulation)

* As the name indicates, bottom up is the opposite of the top-down approach which avoids recursion.
* Here, we solve the problem “bottom-up” way i.e. by solving all the related subproblems first. This is typically done by populating into an n-dimensional table. 
* Depending on the results in the table, the solution to the original problem is then computed. This approach is therefore called as **“Tabulation”**.

#### Pseudo Code and Analysis

1. We already know Fib(n) = Fib(n - 1) + Fib(n - 2).
2. Based on the above relation, we calculate the results of smaller subproblems first and then build the table

/\*\*

\* Pseudo code for finding Fib(n) using tabulation

\* @Parameters: n : nth Fibonacci term needed

\* local variable dp[] table built to store results of smaller subproblems

\*/

int Fib(int n) {

if (n==0) return 0;

int dp[] = new int[n+1];

//define base cases

dp[0] = 0;

dp[1] = 1;

//Iteratively compute the results and store

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

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

//return the value corresponding to nth term

return dp[n];

}

1. Analyze the space and time requirements
   * Lets look at the space complexity first.
     + In this case too, we use an array of size n for remembering the results which contributes to a space complexity of O(n).
     + We can further reduce the space complexity from O(N) to O(1) by just using 2 variables. This is left as an assignment to the reader.
   * Lets now look at the time complexity.
     + Lets look at Fib(n).
     + Here, we solve each subproblem only once in iterative manner. So the time complexity of the algorithm is also O(N).
     + Again thanks to DP, **we arrived at solution in linear time complexity**.

#### Implementation

/\* Java Program to find Nth Fibonacci Number using Tabulation \*/

public class Fibonacci

{

int fib(int n){

int dp[] = new int[n+1];

//base cases

dp[0] = 0;

dp[1] = 1;

//calculating and storing the values

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

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

return dp[n];

}

public static void main(String[] args)

{

Fibonacci fibonacci = new Fibonacci();

int n = 10;

System.out.println("Fibonacci number : " + fibonacci.fib(n));

}

}