**Dynamic Programming**

* 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.
* **Overlapping Subproblems and Optimal Substructure** are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.
* 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.
* If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

/\* a simple recursive program for Fibonacci numbers \*/

int fib(int n)

{

if (n <= 1)

return n;

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

}

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)

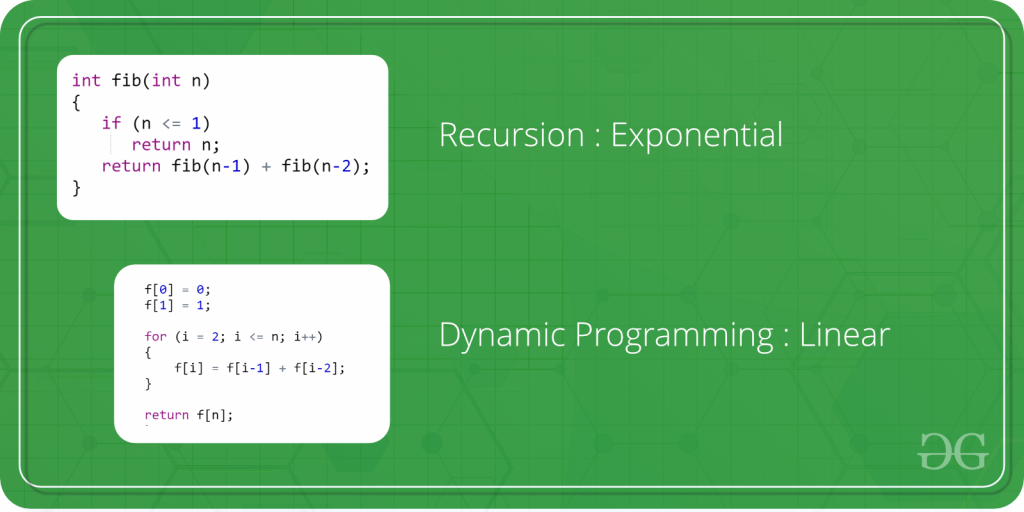
We can see that the function fib(3) is being called 2 times.

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)

* 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.
* For 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.



**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.

/\* C++ program for Memoized version for nth Fibonacci number \*/

#include <bits/stdc++.h>

using namespace std;

#define NIL -1

#define MAX 100

int lookup[MAX];

/\* Function to initialize NIL values in lookup table \*/

void \_initialize()

{

int i;

for (i = 0; i < MAX; i++)

lookup[i] = NIL;

}

/\* function for nth Fibonacci number \*/

int fib(int n)

{

if (lookup[n] == NIL) {

if (n <= 1)

lookup[n] = n;

else

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

}

return lookup[n];

}

// Driver code

int main()

{

int n = 40;

\_initialize();

cout << "Fibonacci number is " << fib(n);

return 0;

}

**Output**

Fibonacci number is 102334155

**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.

/\* C program for Tabulated version \*/

#include <stdio.h>

int fib(int n)

{

int f[n + 1];

int i;

f[0] = 0;

f[1] = 1;

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

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

return f[n];

}

int main()

{

int n = 9;

printf("Fibonacci number is %d ", fib(n));

return 0;

}

**Output**

Fibonacci number is 34

Both Tabulated and Memoized store the solutions of subproblems.

In Memoized version, the table is filled on demand while in the Tabulated version, starting from the first entry, all entries are filled one by one.

To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating the 40th Fibonacci number:

[Recursive solution](https://ide.geeksforgeeks.org/vHt6ly)   
**https://ide.geeksforgeeks.org/vHt6ly**

Memoized solution

**https://ide.geeksforgeeks.org/Z94jYR**

[Tabulated solution](https://ide.geeksforgeeks.org/12C5bP)  
**https://ide.geeksforgeeks.org/12C5bP**

Time taken by the Recursion method is much more than the two Dynamic Programming techniques mentioned above – Memorization and Tabulation!