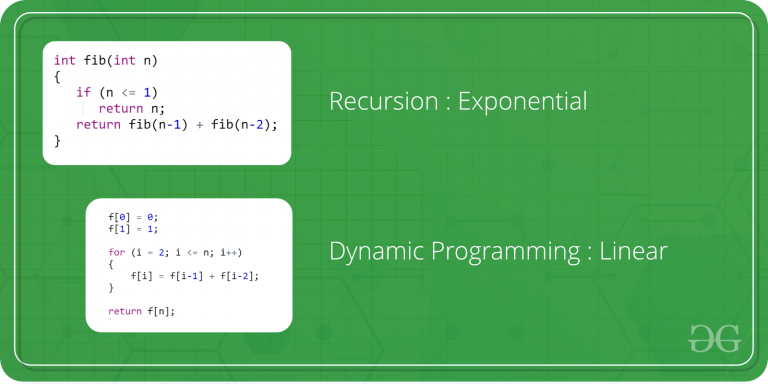
**Dynamic Programming**

Dynamic Programming is mainly an optimization over plain [recursion](https://www.geeksforgeeks.org/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](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/), we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



# How to solve a Dynamic Programming Problem ?

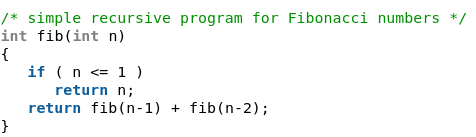
Dynamic Programming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than exponential brute method and can be easily proved for their correctness. Before we study how to think Dynamically for a problem, we need to learn:

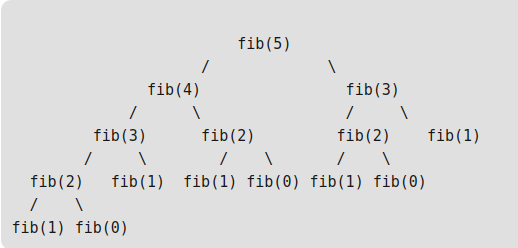
1. [Overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/)
2. [Optimal Substructure Property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)

# Overlapping Subproblems Property in Dynamic Programming

# <https://www.youtube.com/watch?v=mmjDZGSr7EA>

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of 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 which are solved again and again.

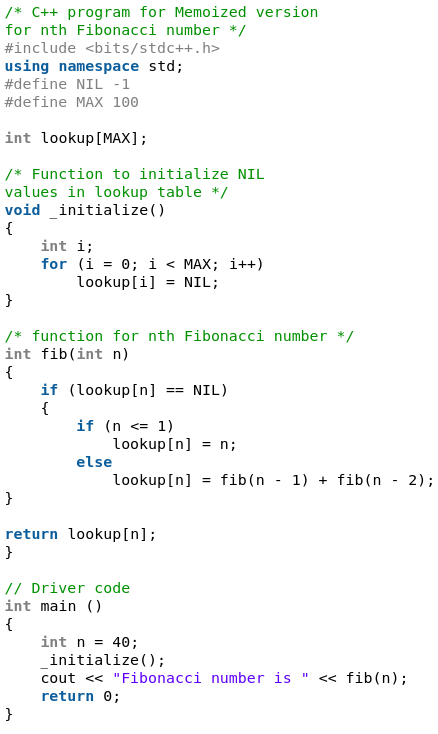




We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:  
a) Memoization (Top 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 it 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. <https://www.youtube.com/watch?time_continue=11&v=Taa9JDeakyU&feature=emb_logo>

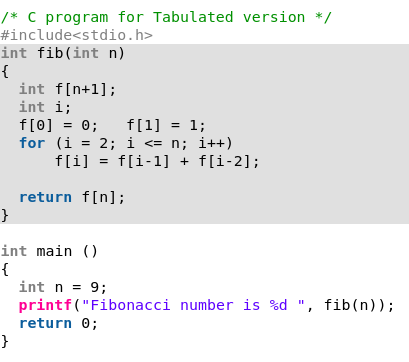
Following is the memoized version for nth Fibonacci Number.



b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from 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.

<https://www.youtube.com/watch?time_continue=17&v=OMkKWtSAF0c&feature=emb_logo>

Following is the tabulated version for nth Fibonacci Number.



Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. For example, [Memoized solution](https://www.ics.uci.edu/~eppstein/161/960229.html)of the [LCS problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)doesn’t necessarily fill all entries.

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

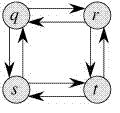
# Optimal Substructure Property in Dynamic Programming

<https://www.youtube.com/watch?time_continue=66&v=JWTqsNvtwP4&feature=emb_logo>

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:  
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like [Floyd–Warshall](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) and [Bellman–Ford](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the [CLRS book](http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf). There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.



**Steps to solve a DP**

1) Identify if it is a DP problem

2) Decide a state expression with

least parameters

3) Formulate state relationship

4) Do tabulation (or add memoization)

**Step 1 : How to classify a problem as a Dynamic Programming Problem?**

* Typically, all the problems that require to maximize or minimize certain quantity or counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using Dynamic Programming.
* All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP.

**Step 2 : Deciding the state**  
DP problems are all about state and their transition. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make. So, let’s see what do we mean by the term “state”.

**State:** A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.

For example: In our famous [Knapsack problem](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/), we define our state by two parameters **index** and **weight** i.e DP[index][weight]. Here DP[index][weight] tells us the maximum profit it can make by taking items from range 0 to index having the capacity of sack to be weight. Therefore, here the parameters index and weight together can uniquely identify a subproblem for the knapsack problem.

So, our first step will be deciding a state for the problem after identifying that the problem is a DP problem.

As we know DP is all about using calculated results to formulate the final result.  
So, our next step will be to find a relation between previous states to reach the current state.

**Step 3 : Formulating a relation among the states**  
This part is the hardest part of for solving a DP problem and requires a lot of intuition, observation and practice. Let’s understand it by considering a sample problem

**Given 3 numbers {1, 3, 5}, we need to tell**

**the total number of ways we can form a number 'N'**

**using the sum of the given three numbers.**

(allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Let’s think dynamically about this problem. So, first of all, we decide a state for the given problem. We will take a parameter n to decide state as it can uniquely identify any subproblem. So, our state dp will look like state(n). Here, state(n) means the total number of arrangements to form n by using {1, 3, 5} as elements. Now, we need to compute state(n).

**How to do it?**  
So here the intuition comes into action. As we can only use 1, 3 or 5 to form a given number. Let us assume that we know the result for n = 1,2,3,4,5,6 ; being termilogistic let us say we know the result for the  
state (n = 1), state (n = 2), state (n = 3) ……… state (n = 6)

Now, we wish to know the result of the state (n = 7). See, we can only add 1, 3 and 5. Now we can get a sum total of 7 by the following 3 ways:

**1) Adding 1 to all possible combinations of state (n = 6)**  
Eg : [ (1+1+1+1+1+1) + 1]  
[ (1+1+1+3) + 1]  
[ (1+1+3+1) + 1]  
[ (1+3+1+1) + 1]  
[ (3+1+1+1) + 1]  
[ (3+3) + 1]  
[ (1+5) + 1]  
[ (5+1) + 1]

**2) Adding 3 to all possible combinations of state (n = 4);**

Eg : [(1+1+1+1) + 3]  
[(1+3) + 3]  
[(3+1) + 3]

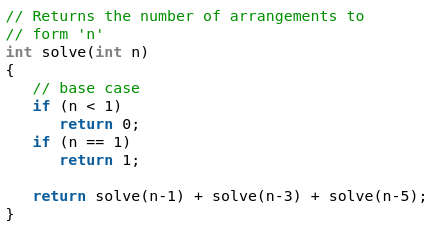
**3) Adding 5 to all possible combinations of state(n = 2)**  
Eg : [ (1+1) + 5]

Now, think carefully and satisfy yourself that the above three cases are covering all possible ways to form a sum total of 7;

Therefore, we can say that result for  
state(7) = state (6) + state (4) + state (2)  
or  
state(7) = state (7-1) + state (7-3) + state (7-5)

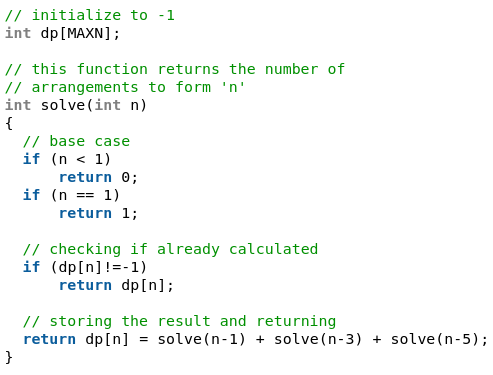
In general,  
**state(n) = state(n-1) + state(n-3) + state(n-5)**

So, our code will look like:



The above code seems exponential as it is calculating the same state again and again. So, we just need to add a memoization.

**Step 4 : Adding memoization or tabulation for the state**  
This is the easiest part of a dynamic programming solution. We just need to store the state answer so that next time that state is required, we can directly use it from our memory. Adding memoization to the above code..



You may check the below problems first and try solving them using the above described steps:-

* <http://www.spoj.com/problems/COINS/>
* <http://www.spoj.com/problems/ACODE/>
* <https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/>
* <https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/>
* <https://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/>
* <https://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/>

# Tabulation vs Memoization

Let’s describe a state for our DP problem to be dp[x] with dp[0] as base state and dp[n] as our destination state. So,  we need to find the value of destination state i.e dp[n].  
If we start our transition from our base state i.e dp[0] and follow our state transition relation to reach our destination state dp[n], we call it Bottom Up approach as it is quite clear that we started our transition from the bottom base state and reached the top most desired state.

**Now, Why do we call it tabulation method?**

To know this let’s first write some code to calculate the factorial of a number using bottom up approach. Once, again as our general procedure to solve a DP we first define a state. In this case, we define a state as dp[x], where dp[x] is to find the factorial of x.

Now, it is quite obvious that dp[x+1] = dp[x] \* (x+1)

// Tabulated version to find factorial x.

int dp[MAXN];

// base case

int dp[0] = 1;

for (int i = 1; i< =n; i++)

{

dp[i] = dp[i-1] \* i;

}

The above code clearly follows the bottom-up approach as it starts its transition from the bottom-most base case dp[0] and reaches its destination state dp[n]. Here, we may notice that the dp table is being populated sequentially and we are directly accessing the calculated states from the table itself and hence, we call it tabulation method.

**Memoization Method – Top Down Dynamic Programming**

Once, again let’s describe it in terms of state transition. If we need to find the value for some state say dp[n] and instead of starting from the base state that i.e dp[0] we ask our answer from the states that can reach the destination state dp[n] following the state transition relation, then it is the top-down fashion of DP.

Here, we start our journey from the top most destination state and compute its answer by taking in count the values of states that can reach the destination state, till we reach the bottom most base state.

Once again, let’s write the code for the factorial problem in the top-down fashion

// Memoized version to find factorial x.

// To speed up we store the values

// of calculated states

// initialized to -1

int dp[MAXN]

// return fact x!

int solve(int x)

{

if (x==0)

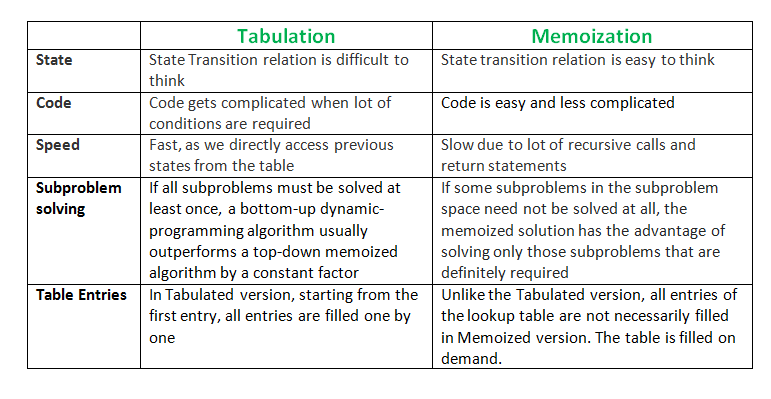
return 1;

if (dp[x]!=-1)

return dp[x];

return (dp[x] = x \* solve(x-1));

}

As we can see we are storing the most recent cache up to a limit so that if next time we got a call from the same state we simply return it from the memory. So, this is why we call it memoization as we are storing the most recent state values. In this case the memory layout is linear that’s why it may seem that the memory is being filled in a sequential manner like the tabulation method, but you may consider any other top down DP having 2D memory layout like [Min Cost Path](https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/), here the memory is not filled in a sequential manner.