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.Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

====================================================================

**Basic Concepts:**

1. [Overlapping Subproblems Property](https://www.geeksforgeeks.org/dynamic-programming-set-1/)
2. [Optimal Substructure Property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
3. [How to solve a Dynamic Programming Problem ?](https://www.geeksforgeeks.org/solve-dynamic-programming-problem/)
4. [Tabulation vs Memoizatation](https://www.geeksforgeeks.org/tabulation-vs-memoizatation/)

=====================================================================

1. [Overlapping Subproblems Property](https://www.geeksforgeeks.org/dynamic-programming-set-1/)

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 recomputed.

EX- during teaching

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

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.

1. [How to solve a Dynamic Programming Problem ?](https://www.geeksforgeeks.org/solve-dynamic-programming-problem/)

**D**ynamic **P**rogramming (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

**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 lots 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 for 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:

|  |
| --- |
| // Returns the number of arrangements to  // form 'n'  int solve(int n)  {  // base case  if (n < 0)  return 0;  if (n == 0)  return 1;  return solve(n-1) + solve(n-3) + solve(n-5);  } |

1. [Tabulation vs Memoizatation](https://www.geeksforgeeks.org/tabulation-vs-memoizatation/)

# Tabulation vs Memoizatation

Prerequisite – [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming-set-1/), [How to solve Dynamic Programming problems?](https://www.geeksforgeeks.org/solve-dynamic-programming-problem/)  
There are following two different ways to store the values so that the values of a problem can be reused. Here, will discuss two patterns of solving DP problem:

* 1. **Tabulation:** Bottom Up
  2. **Memoization:** Top Down

Before getting to the definitions of the above two terms consider the below statements:

* **Version 1**: I will study the theory of Dynamic Programming from GeeksforGeeks, then I will practice some problems on classic DP and hence I will master Dynamic Programming.
* **Version 2**: To Master Dynamic Programming, I would have to practice Dynamic problems and to practice  
  problems – Firstly, I would have to study some theory of Dynamic Programming from GeeksforGeeks

Both the above versions say the same thing, just the difference lies in the way of conveying the message and that’s exactly what Bottom Up and Top Down DP do. Version 1 can be related to as Bottom Up DP and Version-2 can be related as Top Down Dp.

**Tabulation Method – Bottom Up Dynamic Programming**

As the name itself suggests starting from the bottom and cumulating answers to the top. Let’s discuss in terms of state transition.

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] tis 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 it 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 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 for 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.  
