# Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

**Let's understand this approach through an example.**

**Consider an example of the Fibonacci series. The following series is the Fibonacci series:**

**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…**

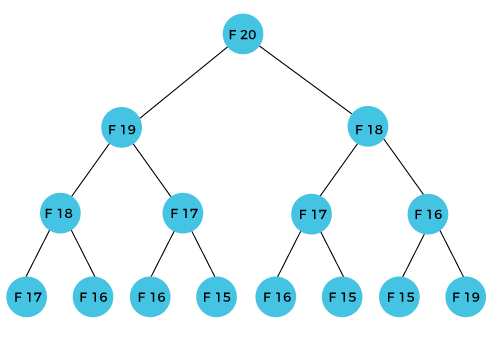
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

**F(n) = F(n-1) + F(n-2),**

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

### How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

### How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

* It breaks down the complex problem into simpler subproblems.
* It finds the optimal solution to these sub-problems.
* It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
* It reuses them so that same sub-problem is calculated more than once.
* Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

## Approaches of dynamic programming

There are two approaches to dynamic programming:

* Top-down approach
* Bottom-up approach

### Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

**Advantages**

* It is very easy to understand and implement.
* It solves the subproblems only when it is required.
* It is easy to debug.

**Disadvantages**

It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.

It occupies more memory that degrades the overall performance.

**Let's understand dynamic programming through an example.**

1. **int** fib(**int** n)
2. {
3. **if**(n<0)
4. error;
5. **if**(n==0)
6. **return** 0;
7. **if**(n==1)
8. **return** 1;
9. sum = fib(n-1) + fib(n-2);
10. }

In the above code, we have used the recursive approach to find out the Fibonacci series. When the value of 'n' increases, the function calls will also increase, and computations will also increase. In this case, the time complexity increases exponentially, and it becomes 2n.

One solution to this problem is to use the dynamic programming approach. Rather than generating the recursive tree again and again, we can reuse the previously calculated value. If we use the dynamic programming approach, then the time complexity would be O(n).

When we apply the dynamic programming approach in the implementation of the Fibonacci series, then the code would look like:

1. **static** **int** count = 0;
2. **int** fib(**int** n)
3. {
4. **if**(memo[n]!= NULL)
5. **return** memo[n];
6. count++;
7. **if**(n<0)
8. error;
9. **if**(n==0)
10. **return** 0;
11. **if**(n==1)
12. **return** 1;
13. sum = fib(n-1) + fib(n-2);
14. memo[n] = sum;
15. }

In the above code, we have used the memorization technique in which we store the results in an array to reuse the values. This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.

### Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

There are two ways of applying dynamic programming:

* **Top-Down**
* **Bottom-Up**

The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

**Key points**

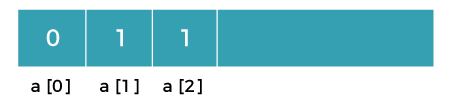
* We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
* We use for loop to iterate over the sub-problems.
* The bottom-up approach is also known as the tabulation or table filling method.

**Let's understand through an example.**

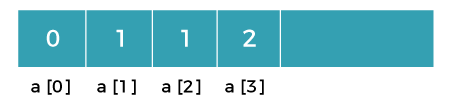
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



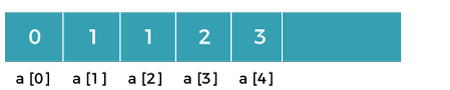
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:



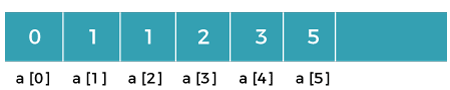
The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



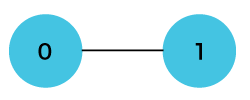
The code for implementing the Fibonacci series using the bottom-up approach is given below:

1. **int** fib(**int** n)
2. {
3. **int** A[];
4. A[0] = 0, A[1] = 1;
5. **for**( i=2; i<=n; i++)
6. {
7. A[i] = A[i-1] + A[i-2]
8. }
9. **return** A[n];
10. }

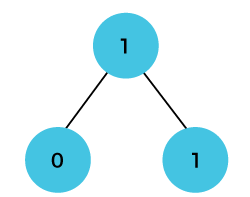
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

**Let's understand through the diagrammatic representation.**

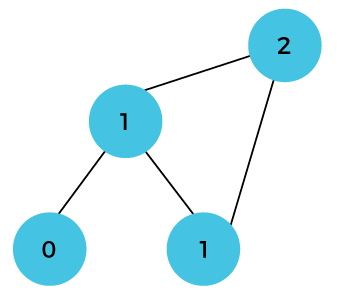
Initially, the first two values, i.e., 0 and 1 can be represented as:



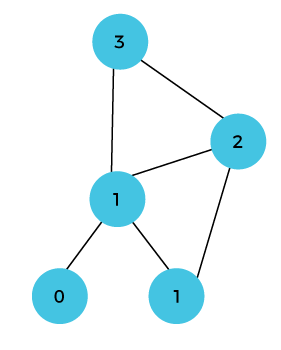
When i=2 then the values 0 and 1 are added shown as below:



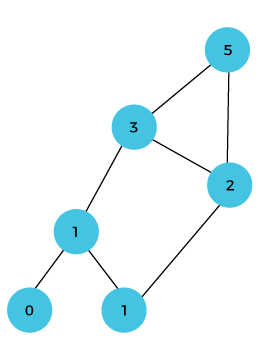
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top.

**Bit Masking**

**Consider the below problems statement. There are 100 different types of caps each having a unique id from 1 to 100. Also, there are ‘n’ persons each having a collection of a variable number of caps. One day all of these persons decide to go in a party wearing a cap but to look unique they decided that none of them will wear the same type of cap. So, count the total number of arrangements or ways such that none of them is wearing the same type of cap. Constraints: 1 <= n <= 10 Example:**

**The first line contains the value of n, next n lines contain collections**

**of all the n persons.**

**Input:**

**3**

**5 100 1 // Collection of the first person.**

**2 // Collection of the second person.**

**5 100 // Collection of the third person.**

**Output:**

**4**

**Explanation: All valid possible ways are (5, 2, 100), (100, 2, 5),**

**(1, 2, 5) and (1, 2, 100)**

**Since, number of ways could be large, so output modulo 1000000007 We strongly recommend you to minimize your browser and try this yourself first. A Simple Solution is to try all possible combinations. Start by picking the first element from the first set, marking it as visited and recur for remaining sets. It is basically a Backtracking based solution. A better solution is to use Bitmasking and DP. Let us first introduce Bitmasking. What is Bitmasking? Suppose we have a collection of elements which are numbered from 1 to N. If we want to represent a subset of this set then it can be encoded by a sequence of N bits (we usually call this sequence a “mask”). In our chosen subset the i-th element belongs to it if and only if the i-th bit of the mask is set i.e., it equals to 1. For example, the mask 10000101 means that the subset of the set [1… 8] consists of elements 1, 3 and 8. We know that for a set of N elements there are total 2N subsets thus 2N masks are possible, one representing each subset. Each mask is, in fact, an integer number written in binary notation. Our main methodology is to assign a value to each mask (and, therefore, to each subset) and thus calculate the values for new masks using values of the already computed masks. Usually our main target is to calculate value/solution for the complete set i.e., for mask 11111111. Normally, to find the value for a subset X we remove an element in every possible way and use values for obtained subsets X’1, X’2… ,X’k to compute the value/solution for X. This means that the values for X’i must have been computed already, so we need to establish an ordering in which masks will be considered. It’s easy to see that the natural ordering will do: go over masks in increasing order of corresponding numbers. Also, We sometimes, start with the empty subset X and we add elements in every possible way and use the values of obtained subsets X’1, X’2… ,X’k to compute the value/solution for X. We mostly use the following notations/operations on masks: bit(i, mask) – the i-th bit of mask count(mask) – the number of non-zero bits in the mask first(mask) – the number of the lowest non-zero bit in the mask set(i, mask) – set the ith bit in the mask check(i, mask) – check the ith bit in the mask How is this problem solved using Bitmasking + DP? The idea is to use the fact that there are upto 10 persons. So we can use an integer variable as a bitmask to store which person is wearing a cap and which is not.**

**Let i be the current cap number (caps from 1 to i-1 are already**

**processed). Let integer variable mask indicates that the persons w**

**earing and not wearing caps. If i'th bit is set in mask, then**

**i'th person is wearing a cap, else not.**

**// consider the case when ith cap is not included**

**// in the arrangement**

**countWays(mask, i) = countWays(mask, i+1) +**

**// when ith cap is included in the arrangement**

**// so, assign this cap to all possible persons**

**// one by one and recur for remaining persons.**

**∑ countWays(mask | (1 << j), i+1)**

**for every person j that can wear cap i**

**Note that the expression "mask | (1 << j)" sets j'th bit in mask.**

**And a person can wear cap i if it is there in the person's cap list**

**provided as input.**

**If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. So we use Dynamic Programming. A table dp[][] is used such that in every entry dp[i][j], i is mask and j is cap number. Since we want to access all persons that can wear a given cap, we use an array of vectors, capList[101]. A value capList[i] indicates the list of persons that can wear cap i. Below is the implementation of above idea.**

* **C++**
* **Java**
* **Python**

| **// Java program to find number of ways to wear hats**    **import java.io.BufferedReader;**  **import java.io.InputStreamReader;**  **import java.util.Vector;**    **class Test**  **{**  **static final int MOD = 1000000007;**    **// for input**  **static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));**    **// capList[i]'th vector contains the list of persons having a cap with id i**  **// id is between 1 to 100 so we declared an array of 101 vectors as indexing**  **// starts from 0.**  **static Vector<Integer> capList[] = new Vector[101];**      **// dp[2^10][101] .. in dp[i][j], i denotes the mask i.e., it tells that**  **// how many and which persons are wearing cap. j denotes the first j caps**  **// used. So, dp[i][j] tells the number ways we assign j caps to mask i**  **// such that none of them wears the same cap**  **static int dp[][] = new int[1025][101];**    **// This is used for base case, it has all the N bits set**  **// so, it tells whether all N persons are wearing a cap.**  **static int allmask;**    **// Mask is the set of persons, i is cap-id (OR the**  **// number of caps processed starting from first cap).**  **static long countWaysUtil(int mask, int i)**  **{**  **// If all persons are wearing a cap so we**  **// are done and this is one way so return 1**  **if (mask == allmask) return 1;**    **// If not everyone is wearing a cap and also there are no more**  **// caps left to process, so there is no way, thus return 0;**  **if (i > 100) return 0;**    **// If we already have solved this subproblem, return the answer.**  **if (dp[mask][i] != -1) return dp[mask][i];**    **// Ways, when we don't include this cap in our arrangement**  **// or solution set.**  **long ways = countWaysUtil(mask, i+1);**    **// size is the total number of persons having cap with id i.**  **int size = capList[i].size();**    **// So, assign one by one ith cap to all the possible persons**  **// and recur for remaining caps.**  **for (int j = 0; j < size; j++)**  **{**  **// if person capList[i][j] is already wearing a cap so continue as**  **// we cannot assign him this cap**  **if ((mask & (1 << capList[i].get(j))) != 0) continue;**    **// Else assign him this cap and recur for remaining caps with**  **// new updated mask vector**  **else ways += countWaysUtil(mask | (1 << capList[i].get(j)), i+1);**  **ways %= MOD;**  **}**    **// Save the result and return it.**  **return dp[mask][i] = (int) ways;**  **}**    **// Reads n lines from standard input for current test case**  **static void countWays(int n) throws Exception**  **{**  **//----------- READ INPUT --------------------------**  **String str;**  **String split[];**  **int x;**    **for (int i=0; i<n; i++)**  **{**  **str = br.readLine();**  **split = str.split(" ");**    **// while there are words in the split[]**  **for (int j = 0; j < split.length; j++) {**  **// add the ith person in the list of cap if with id x**  **x = Integer.parseInt(split[j]);**  **capList[x].add(i);**  **}**    **}**  **//----------------------------------------------------**    **// All mask is used to check of all persons**  **// are included or not, set all n bits as 1**  **allmask = (1 << n) - 1;**    **// Initialize all entries in dp as -1**  **for (int[] is : dp) {**  **for (int i = 0; i < is.length; i++) {**  **is[i] = -1;**  **}**  **}**    **// Call recursive function count ways**  **System.out.println(countWaysUtil(0, 1));**  **}**    **// Driver method**  **public static void main(String args[]) throws Exception**  **{**  **int n; // number of persons in every test case**    **// initializing vector array**  **for (int i = 0; i < capList.length; i++)**  **capList[i] = new Vector<>();**      **n = Integer.parseInt(br.readLine());**  **countWays(n);**  **}**  **}** |
| --- |