

# Striver DP Series : Dynamic Programming Problems

 [takeuforward.org/dynamic-programming/striver-dp-series-dynamic-programming-problems](https://takeuforward.org/dynamic-programming/striver-dp-series-dynamic-programming-problems)

February 10, 2022



Striver DP Series : Dynamic Programming Problems

Dynamic Programming can be described as storing answers to various sub-problems to be used later whenever required to solve the main problem.

The two common dynamic programming approaches are:

- Memoization: Known as the “top-down” dynamic programming, usually the problem is solved in the direction of the main problem to the base cases.
- Tabulation: Known as the “bottom-up ” dynamic programming, usually the problem is solved in the direction of solving the base cases to the main problem.

This post contains some hand-picked questions by Striver to learn or master Dynamic Programming. The post contains popular dynamic programming problems along with a detailed tutorials (both text and video). You can also practice the problem on the given link before jumping straight to the solution.

[Share on Whatsapp](#)

## ▼ Part 1: Introduction to DP

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Dynamic Programming Introduction</a>	<a href="#">Link</a>	<a href="#">Link</a>

## ▼ Part 2: 1D DP

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Climbing Stars</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Frog Jump(DP-3)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Frog Jump with k distances(DP-4)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Maximum sum of non-adjacent elements (DP 5)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">House Robber (DP 6)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 3: 2D/3D DP and DP on Grids

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Ninja's Training (DP 7)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Grid Unique Paths : DP on Grids (DP8)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Grid Unique Paths 2 (DP 9)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum path sum in Grid (DP 10)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum path sum in Triangular Grid (DP 11)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum/Maximum Falling Path Sum (DP-12)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">3-d DP : Ninja and his friends (DP-13)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 4: DP on Subsequences

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Subset sum equal to target (DP- 14)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Partition Equal Subset Sum (DP- 15)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Partition Set Into 2 Subsets With Min Absolute Sum Diff (DP- 16)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Count Subsets with Sum K (DP – 17)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Count Partitions with Given Difference (DP – 18)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">0/1 Knapsack (DP – 19)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum Coins (DP – 20)</a>	<a href="#">Link</a>	<a href="#">Link</a>

<a href="#">Target Sum (DP – 21)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Coin Change 2 (DP – 22)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Unbounded Knapsack (DP – 23)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Rod Cutting Problem  (DP – 24)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 5: DP on Strings

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Longest Common Subsequence  (DP – 25)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Print Longest Common Subsequence  (DP – 26)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Longest Common Substring  (DP – 27)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Longest Palindromic Subsequence  (DP-28)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum insertions to make string palindrome   DP-29</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum Insertions/Deletions to Convert String  (DP-30)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Shortest Common Supersequence  (DP – 31)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Distinct Subsequences  (DP-32)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Edit Distance  (DP-33)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Wildcard Matching  (DP-34)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 6: DP on Stocks

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Best Time to Buy and Sell Stock  (DP-35)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Buy and Sell Stock – II  (DP-36)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Buy and Sell Stocks III  (DP-37)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Buy and Stock Sell IV  (DP-38)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Buy and Sell Stocks With Cooldown  (DP-39)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Buy and Sell Stocks With Transaction Fee  (DP-40)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 7: DP on LIS

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Longest Increasing Subsequence  (DP-41)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Printing Longest Increasing Subsequence (DP-42)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Longest Increasing Subsequence  (DP-43)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Largest Divisible Subset (DP-44)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Longest String Chain (DP-45)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Longest Bitonic Subsequence  (DP-46)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Number of Longest Increasing Subsequences (DP-47)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 8: MCM DP | Partition DP

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Matrix Chain Multiplication (DP-48)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Matrix Chain Multiplication   Bottom-Up (DP-49)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Minimum Cost to Cut the Stick (DP-50)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Burst Balloons (DP-51)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Evaluate Boolean Expression to True (DP-52)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Palindrome Partitioning – II (DP-53)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Partition Array for Maximum Sum (DP-54)</a>	<a href="#">Link</a>	<a href="#">Link</a>

#### ▼ Part 9: DP on Squares

Find both C++/Java codes of all problem in the articles in the first column.

Topic	Video Solution	Practice Link
<a href="#">Maximum Rectangle Area with all 1's (DP-55)</a>	<a href="#">Link</a>	<a href="#">Link</a>
<a href="#">Count Square Submatrices with All Ones (DP-56)</a>	<a href="#">Link</a>	<a href="#">Link</a>

[Share on Whatsapp](#)

# Dynamic Programming Introduction

 [takeuforward.org/data-structure/dynamic-programming-introduction](https://takeuforward.org/data-structure/dynamic-programming-introduction)

January 9, 2022

## Problem Statement: Introduction To Dynamic Programming

In this article, we will be going to understand the concept of dynamic programming.

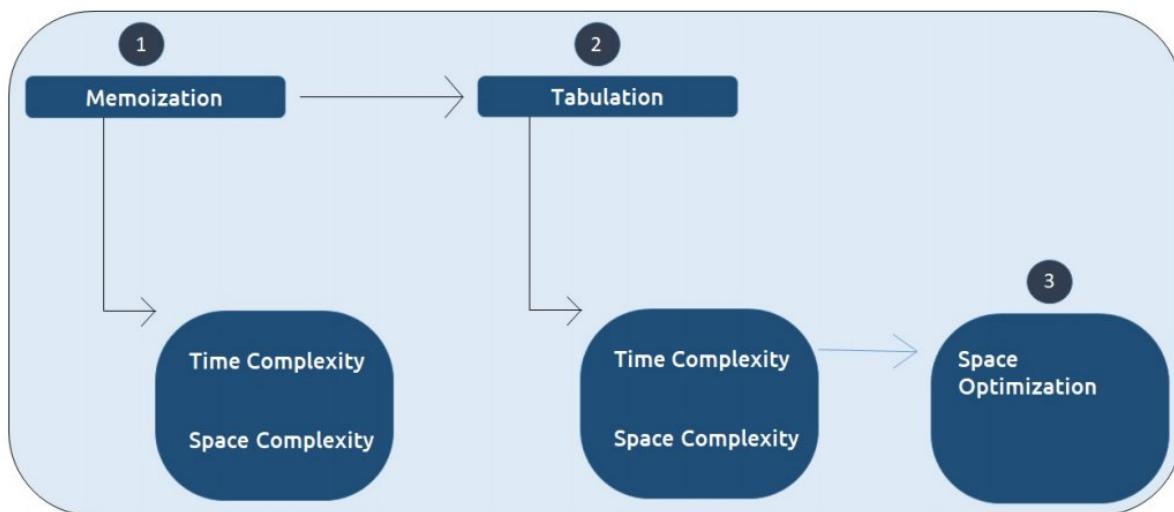
Dynamic Programming can be described as storing answers to various sub-problems to be used later whenever required to solve the main problem.

The two common dynamic programming approaches are:

- Memoization: Known as the “top-down” dynamic programming, usually the problem is solved in the direction of the main problem to the base cases.
- Tabulation: Known as the “bottom-up ” dynamic programming, usually the problem is solved in the direction of solving the base cases to the main problem

**Note:** The base case does not always mean smaller input. It depends upon the implementation of the algorithm.

The flow of this article will be as follows:



We will be using the example of Fibonacci numbers here. The following series is called the Fibonacci series:

**0,1,1,2,3,5,8,13,21,...**

We need to find the  $n^{\text{th}}$  Fibonacci number, where  $n$  is based on a 0-based index.

Every  $i^{\text{th}}$  number of the series is equal to the sum of  $(i-1)^{\text{th}}$  and  $(i-2)^{\text{th}}$  number where the first and second number is given as 0 and 1 respectively.

**Disclaimer:** *Don't jump directly to the solution, try it out yourself first.*

## Pre-req: Recursion

### Solution :

#### Part – 1: Memoizaton

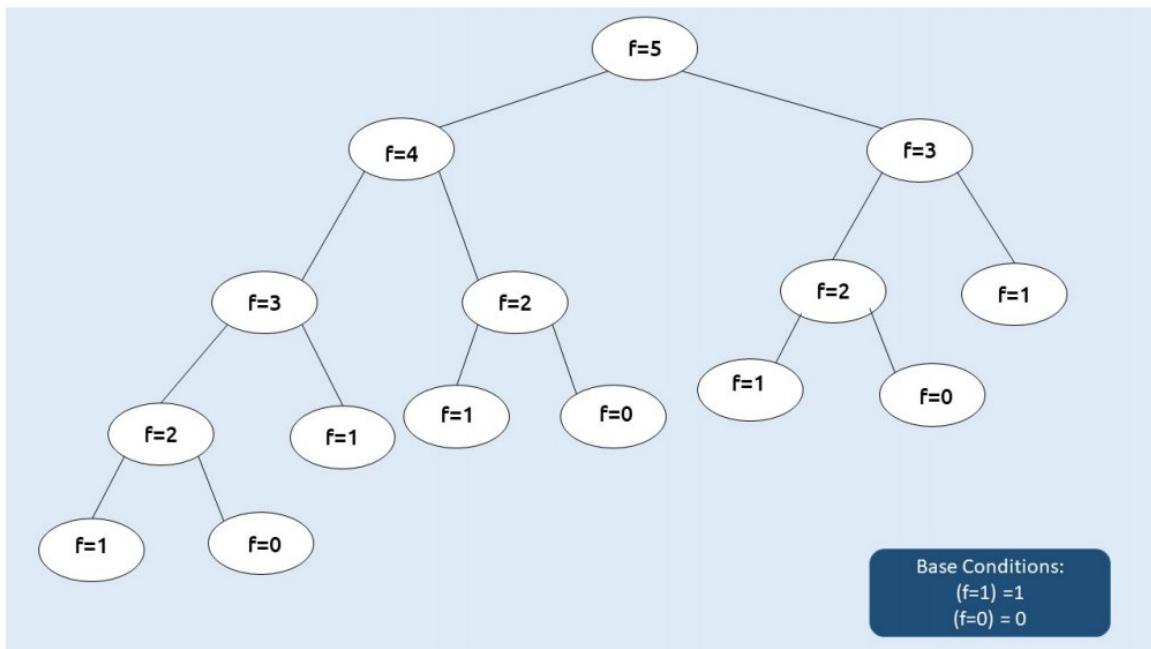
As every number is equal to the sum of the previous two terms, the recurrence relation can be written as:

The basic pseudo-code for the problem will be given as:

If we draw the recursive tree for n=5, it will be:

$$f(n) = f(n-1) + f(n-2)$$

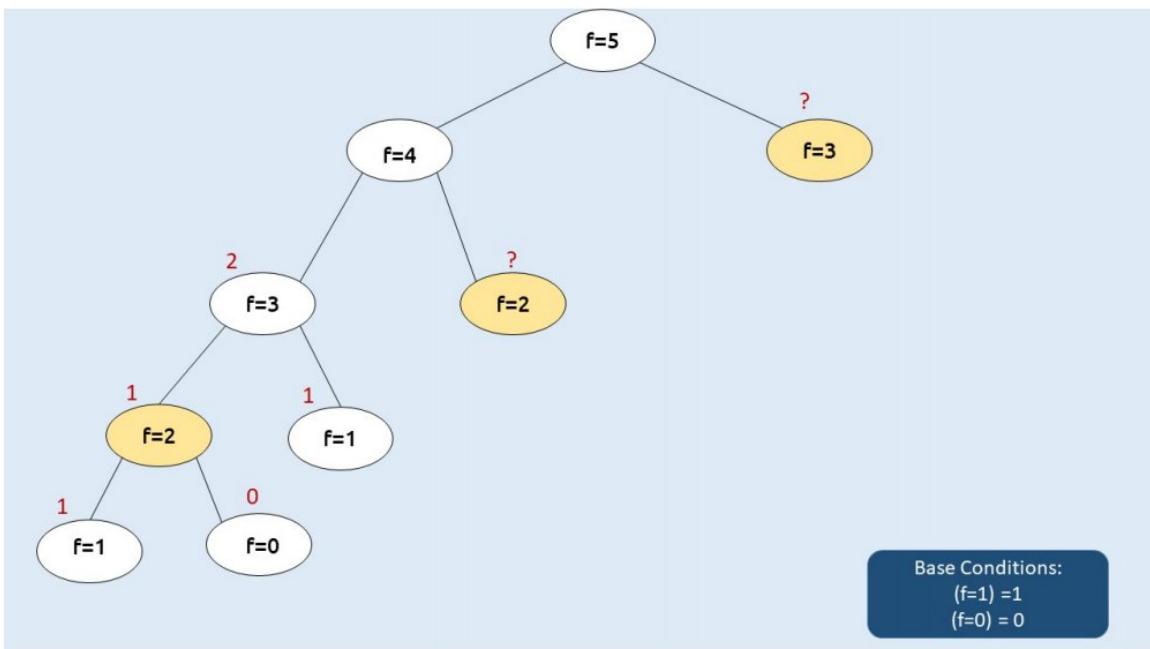
```
f(n) {  
    if( n<= 1) return n  
  
    return f(n-1) + f(n-2)  
}
```



If there are two recursive calls inside a function, the program will run the first call, finish its execution and then run the second call. Due to this reason, each and every call in the recursive tree will be executed. This gives the recursive code its exponential time

complexity.

Can we improve on this? The answer is **Yes!**



We want to compute  $f(2)$  as the second call from  $f(4)$ , but in the recursive tree we had already computed  $f(2)$  once (in the first recursive call of  $f(3)$ ). Similar is the case with  $f(3)$ , therefore if we somehow store these values, the first time we calculated it then we can simply find its value in constant time whenever we need it. This technique is called **Memoization**. Here the cases marked in yellow are called overlapping sub-problems and we need to solve them only once during the code execution.

### Steps to memoize a recursive solution:

Any recursive solution to a problem can be memoized using these three steps:

1. Create a  $dp[n+1]$  array initialized to -1.
2. Whenever we want to find the answer of a particular value (say  $n$ ), we first check whether the answer is already calculated using the  $dp$  array (i.e  $dp[n] \neq -1$ ). If yes, simply return the value from the  $dp$  array.
3. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[n]$  to the solution we get.

### Dry Run:

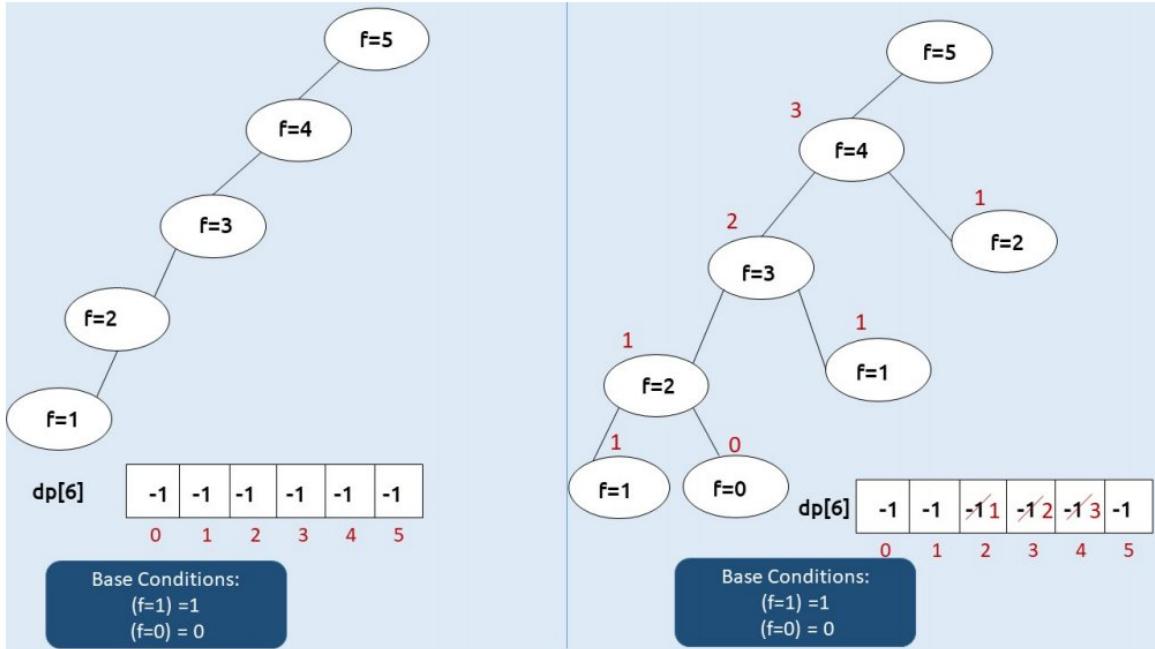
Declare a  $dp[]$  array of size  $n+1$  and initialize it to -1.

Now, following the recursive code we see that at  $n=5$ , the value of  $dp[5]$  is equal to -1 therefore we need to compute its value

dp[6]	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

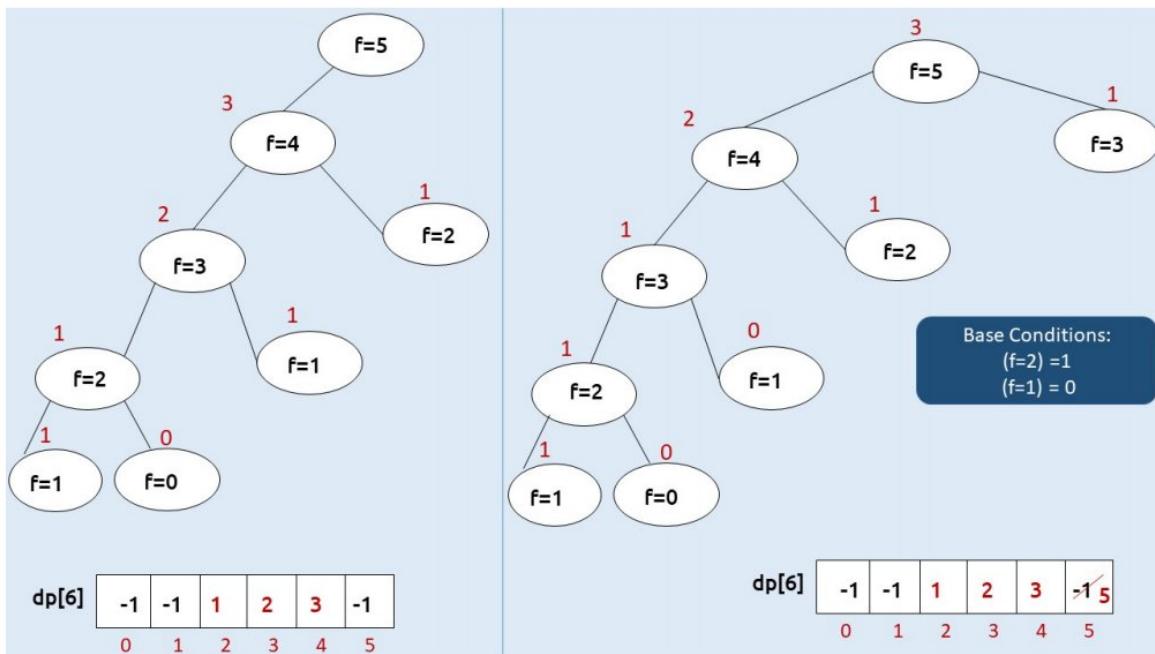
and go to the inner recursive calls. Similar is the case at  $n=4$ ,  $n=3$ , and  $n=2$ . For  $n=2$ , we hit our two base conditions as the inner recursive calls.

As we traverse back after solving  $n=2$ , we update its dp array value to 1 (the answer we got). Similarly, for the second recursive call for  $n=3$ , we again hit a base condition and we get an answer of  $f(n=3)$  as 2, we again update the dp array.



Then for the second recursive call  $f(n=4)$ , we see that  $dp[2]$  is not equal to -1, which means that we have already solved this subproblem and we simply return the value at  $dp[2]$  as our answer. Hence we get the answer of  $f(n=4)$  as  $3(2+1)$ .

Similarly, for the second recursive call  $f(n=5)$ , we get  $dp[3]$  as 2. Then we compute for  $f(n=5)$  as  $5(2+3)$ .



### **Code:**

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int f(int n, vector<int>& dp){
    if(n<=1) return n;

    if(dp[n]!= -1) return dp[n];
    return dp[n]= f(n-1,dp) + f(n-2,dp);
}

int main() {

    int n=5;
    vector<int> dp(n+1,-1);
    cout<<f(n,dp);
    return 0;
}
```

**Output:** 5

### **Time Complexity: O(N)**

Reason: The overlapping subproblems will return the answer in constant time O(1). Therefore the total number of new subproblems we solve is ‘n’. Hence total time complexity is O(N).

### **Space Complexity: O(N)**

Reason: We are using a recursion stack space(O(N)) and an array (again O(N)). Therefore total space complexity will be  $O(N) + O(N) \approx O(N)$

### **Part -2: Tabulation**

Tabulation is a ‘bottom-up’ approach where we start from the base case and reach the final answer that we want.

### **Steps to convert Recursive Solution to Tabulation one.**

- Declare a dp[] array of size n+1.
- First initialize the base condition values, i.e i=0 and i=1 of the dp array as 0 and 1 respectively.
- Set an iterative loop which traverses the array( from index 2 to n) and for every index set its value as  $dp[i] = dp[i-1] + dp[i-2]$ .

### **Code:**

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int main() {

    int n=5;
    vector<int> dp(n+1, -1);

    dp[0]= 0;
    dp[1]= 1;

    for(int i=2; i<=n; i++){
        dp[i] = dp[i-1]+ dp[i-2];
    }
    cout<<dp[n];
    return 0;
}
```

**Output:** 5

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(N)**

Reason: We are using an external array of size ‘n+1’.

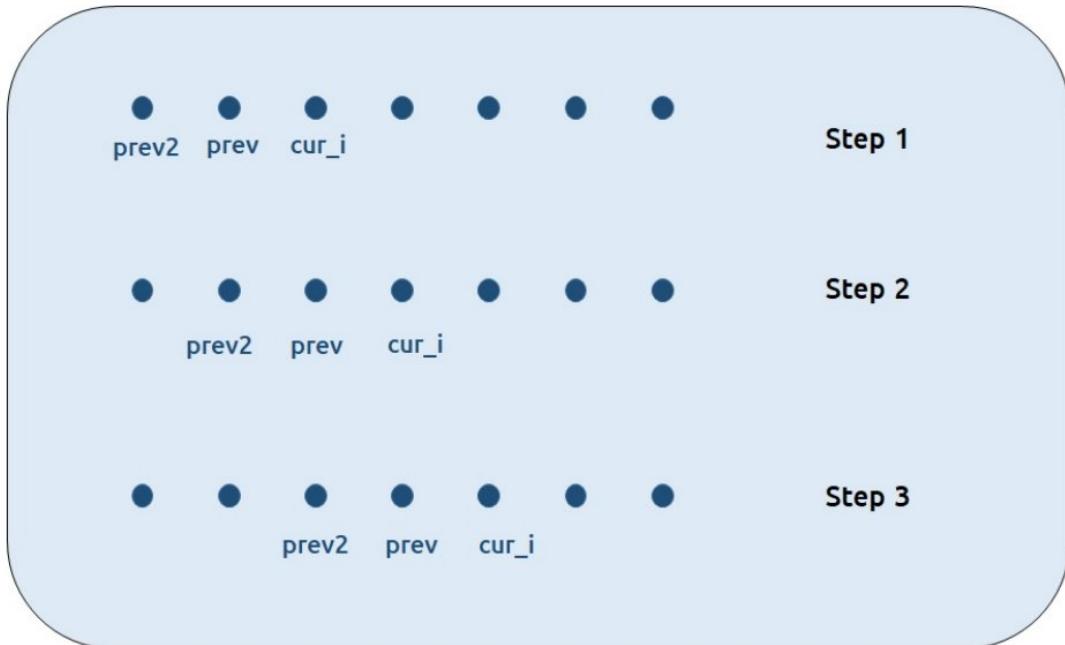
**Part 3: Space Optimization**

If we closely look the relation,

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

we see that for any i, we do need only the last two values in the array. So is there a need to maintain a whole array for it?

The answer is ‘No’. Let us call  $dp[i-1]$  as prev and  $dp[i-2]$  as prev2. Now understand the following illustration.



- Each iteration's `cur_i` and `prev` becomes the next iteration's `prev` and `prev2` respectively.
- Therefore after calculating `cur_i`, if we update `prev` and `prev2` according to the next step, we will always get the answer.
- After the iterative loop has ended we can simply return `prev` as our answer.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int main() {

    int n=5;

    int prev2 = 0;
    int prev = 1;

    for(int i=2; i<=n; i++){
        int cur_i = prev2+ prev;
        prev2 = prev;
        prev= cur_i;
    }
    cout<<prev;
    return 0;
}
```

**Output:** 5

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(1)**

# Dynamic Programming : Climbing Stairs

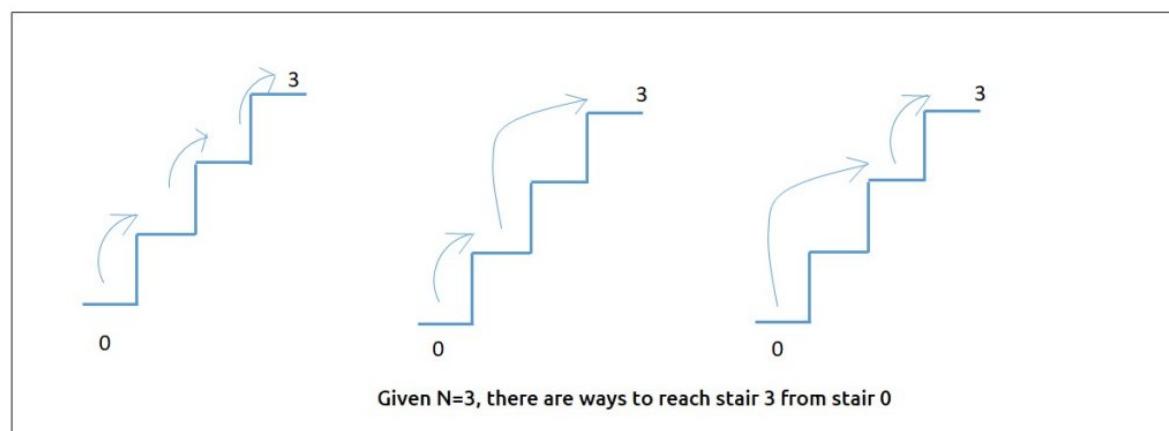
 [takeuforward.org/data-structure/dynamic-programming-climbing-stairs](https://takeuforward.org/data-structure/dynamic-programming-climbing-stairs)

January 9, 2022

## How to write 1-D Recurrence relation / Climbing Stairs

In this article, we will learn to write 1-D Recurrence relations using the problem “Climbing Stairs”

**Problem Statement:** Given a number of stairs. Starting from the 0th stair we need to climb to the “Nth” stair. At a time we can climb either one or two steps. We need to return the total number of distinct ways to reach from 0th to Nth stair.



**Pre-req:** Recursion, [Dynamic Programming Introduction](#)

**Solution :**

### How to Identify a DP problem?

When we see a problem, it is very important to identify it as a dynamic programming problem. Generally (but not limited to) if the problem statement asks for the following:

- Count the total number of ways
- Given multiple ways of doing a task, which way will give the minimum or the maximum output.

We can try to apply recursion. Once we get the recursive solution, we can go ahead to convert it to a dynamic programming one.

### Steps To Solve The Problem After Identification

Once the problem has been identified, the following three steps comes handy in solving the problem:

- Try to represent the problem in terms of indexes.

- Try all possible choices/ways at every index according to the problem statement.
- If the question states
  - Count all the ways – return sum of all choices/ways.
  - Find maximum/minimum- return the choice/way with maximum/minimum output.

### Using these steps to solve the problem “Climbing Stairs”

**Step 1:** We will assume n stairs as indexes from 0 to N.

**Step 2:** At a single time, we have 2 choices: Jump one step or jump two steps. We will try both of these options at every index.

**Step 3:** As the problem statement asks to count the total number of distinct ways, we will return the sum of all the choices in our recursive function.

The base case will be when we want to go to the 0th stair, then we have only one option.

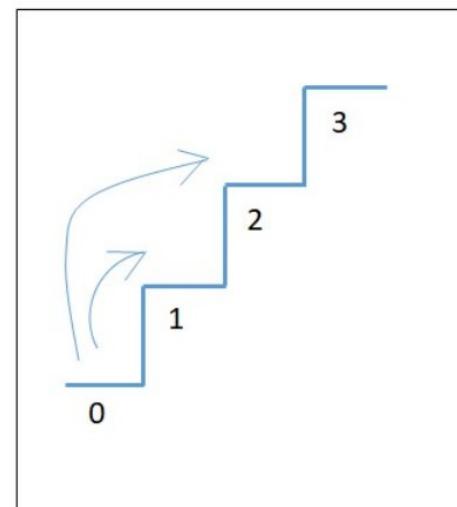
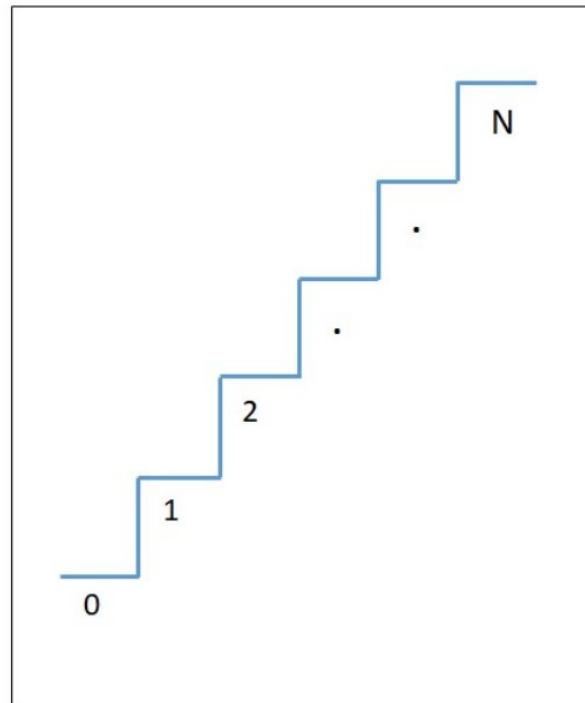
The basic pseudo-code for the problem will be given as:

There will be one more edge-case when  $n=1$ , if we call  $f(n-2)$  we will reach stair numbered -1 which is not defined, therefore we add an extra test case to return 1 (the only way) when  $n=1$ .

If we clearly observe the pseudo-code, we see that it almost matches the problem “**fibonacci numbers**” discussed in [Dynamic Programming Introduction](#)! So the readers can follow that article to understand the approach used for the dynamic programming solution after the recursive solution.

### Steps for the Tabulation approach.

- Declare a  $dp[]$  array of size  $n+1$ .
- First initialize the base condition values, i.e  $i=0$  and  $i=1$  of the  $dp$  array as 1.



- Set an iterative loop which traverses the array( from index 2 to n) and for every index set its value as  $dp[i-1] + dp[i-2]$ .

### Code:

```
C++ Code
#include <bits/stdc++.h>

using namespace std;

int main() {

    int n=3;
    vector<int> dp(n+1, -1);

    dp[0]= 1;
    dp[1]= 1;

    for(int i=2; i<=n; i++){
        dp[i] = dp[i-1]+ dp[i-2];
    }
    cout<<dp[n];
    return 0;
}
```

### Time Complexity: O(N)

Reason: We are running a simple iterative loop

### Space Complexity: O(N)

Reason: We are using an external array of size ‘n+1’.

### Part 3: Space Optimization

If we closely look the relation,

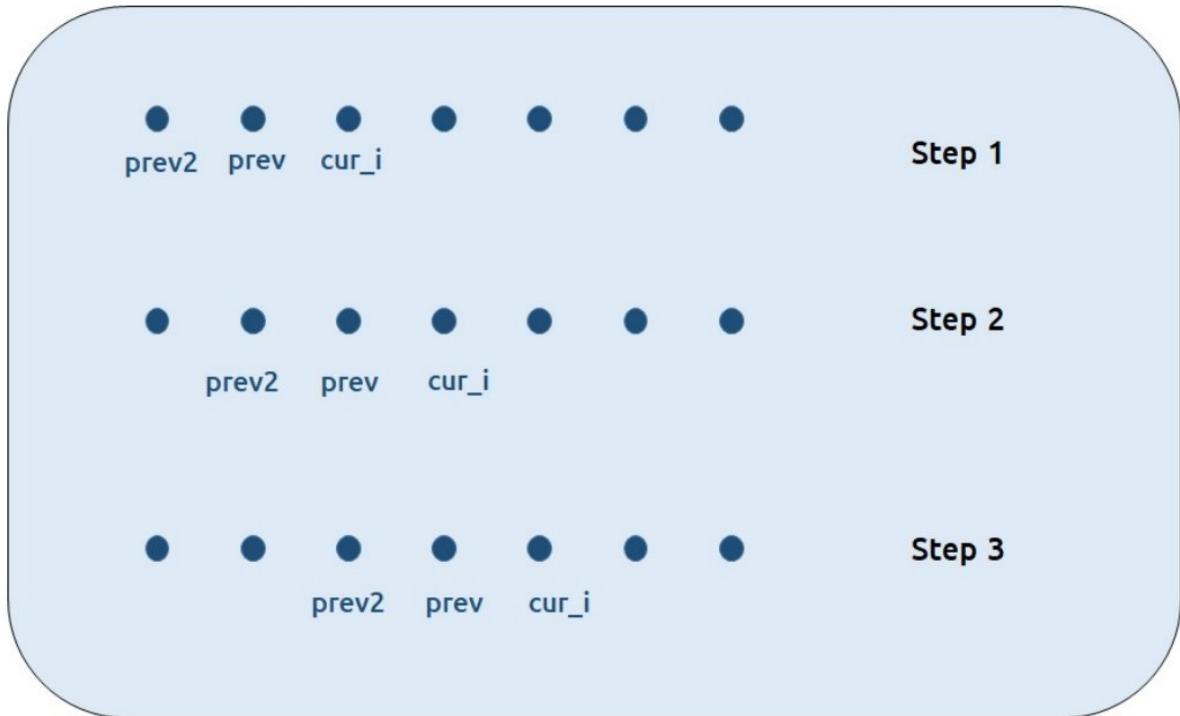
$$dp[i] = dp[i-1] + dp[i-2]$$

we see that for any i, we do need only the last two values in the array. So is there a need to maintain a whole array for it?

The answer is ‘No’. Let us call  $dp[i-1]$  as prev and  $dp[i-2]$  as prev2. Now understand the following illustration.

```
f(ind) {
    if( ind == 0) return 1
    return f(ind-1) + f(ind-2)
}
```

```
f(ind) {
    if( ind == 0) return 1
    if( ind == 1) return 1
    return f(ind-1) + f(ind-2)
}
```



- Each iteration's `cur_i` and `prev` becomes the next iteration's `prev` and `prev2` respectively.
- Therefore after calculating `cur_i`, if we update `prev` and `prev2` according to the next step, we will always get the answer.
- After the iterative loop has ended we can simply return `prev` as our answer.

### Code:

```
C++ Code
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n=3;

    int prev2 = 1;
    int prev = 1;

    for(int i=2; i<=n; i++){
        int cur_i = prev2+ prev;
        prev2 = prev;
        prev= cur_i;
    }
    cout<<prev;
    return 0;
}
```

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(1)**

Reason: We are not using any extra space.

# Dynamic Programming : Frog Jump (DP 3)

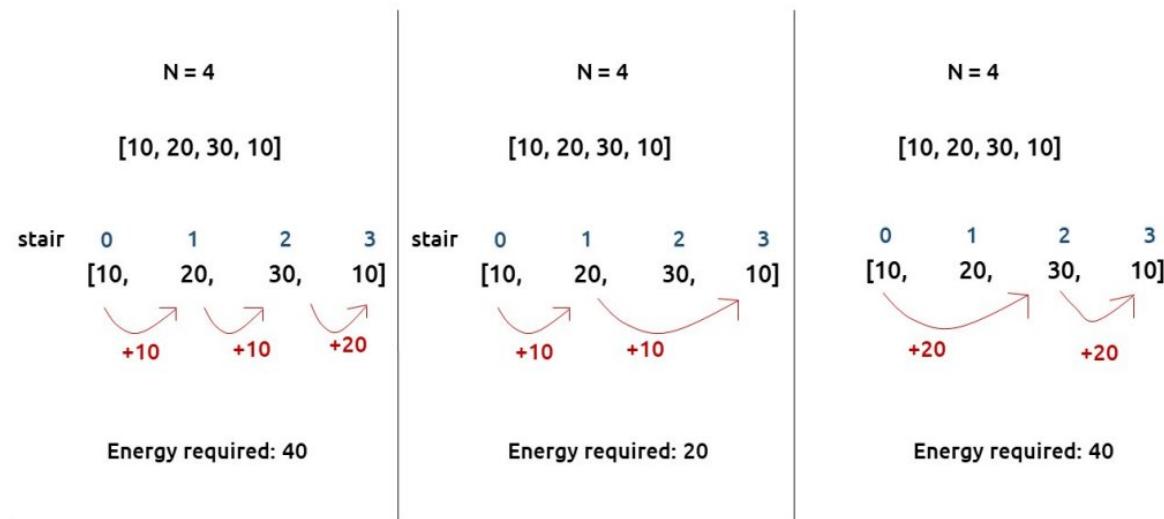
 [takeuforward.org/data-structure/dynamic-programming-frog-jump-dp-3](https://takeuforward.org/data-structure/dynamic-programming-frog-jump-dp-3)

January 10, 2022

## Problem Statement:

Given a number of stairs and a frog, the frog wants to climb from the 0th stair to the (N-1)th stair. At a time the frog can climb either one or two steps. A height[N] array is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is  $\text{abs}(\text{height}[i] - \text{height}[j])$ , where  $\text{abs}()$  means the absolute difference. We need to return the minimum energy that can be used by the frog to jump from stair 0 to stair N-1.

### Examples:



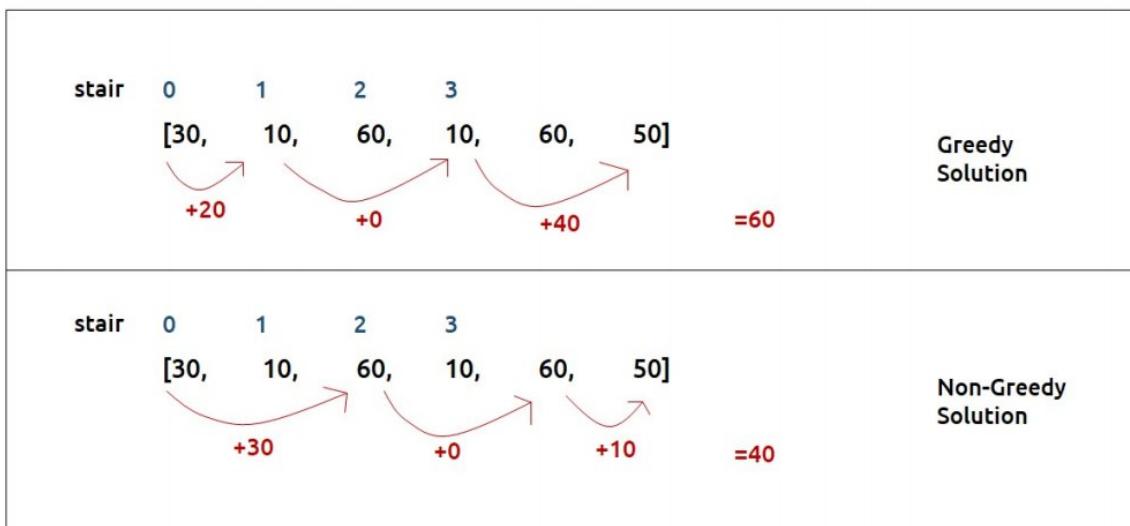
## Pre-req: Recursion, Dynamic Programming Introduction

## Solution :

As the problem statement states to find the minimum energy required, two approaches should come to our mind, greedy and dynamic programming.

First, we will see why a greedy approach will not work?

The total energy required by the frog depends upon the path taken by the frog. If the frog just takes the cheapest path in every stage it can happen that it eventually takes a costlier path after a certain number of jumps. The following example will help to understand this.



Therefore a greedy solution will not work and we need to try all possible paths to find the answer.

### Steps to form the recursive solution

We will recap the steps discussed in the [previous article](#) to form the recursive solution.

#### Step 1: Express the problem in terms of indexes

- This can be easily done as there are array indexes [0,1,2,..., n-1].
- We can say that  $f(n-1)$  signifies the minimum amount of energy required to move from stair 0 to stair  $n-1$ .
- Therefore  $f(0)$  simply should give us the answer as 0(base case).

#### Step 2: Try all the choices to reach the goal.

The frog can jump either by one step or by two steps. We will calculate the cost of the jump from the height array. The rest of the cost will be returned by the recursive calls that we make

Our pseudocode till this step will be:

```

f(ind,height[ ]) {

    if( ind == 0) return 0

    jumpOne= f(ind-1, height)+  

            abs(height[ind]- height [ind-1])
    if (ind>1)
        jumpTwo= f(ind-2, height)+  

            abs(height[ind]- height [ind-2])

}

```

### **Step 3:** Take the minimum of all the choices

As the problem statement asks to find the minimum total energy, we will return the minimum of two choices of step2.

Also at  $\text{ind}=1$ , we can't try the second choice so we will only make one recursive call.

The base case will be when we want to go to the 0th stair, then we have only one option.

Our final pseudo-code will be:

```

f(ind,height[ ]) {

    if( ind == 0) return 0

    jumpOne= f(ind-1, height)+  

            abs(height[ind]- height [ind-1])
    if (ind>1)
        jumpTwo= f(ind-2, height)+  

            abs(height[ind]- height [ind-2])

    return min(jumpOne,jumpTwo)

}

```

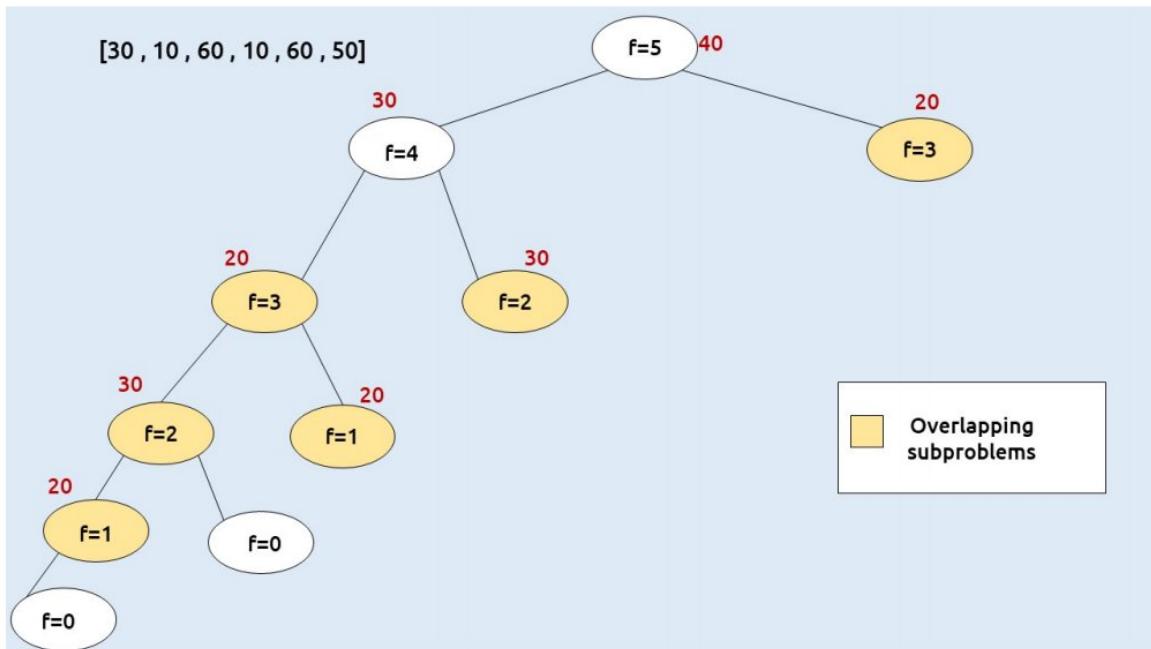
Once we form the recursive solution, we can use the approach told in [Dynamic Programming Introduction](#) to convert it into a dynamic programming one.

### **Memoization approach**

#### **Steps to convert Recursive code to memoization solution:**

- Create a  $dp[n]$  array initialized to -1.
- Whenever we want to find the answer of a particular value (say  $n$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[n] \neq -1$  ). If yes, simply return the value from the dp array.
- If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[n]$  to the solution we get.

### Recursion tree diagram:



**Note:** To watch a detailed dry run of this approach, please watch the video attached below

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int solve(int ind, vector<int>& height, vector<int>& dp){
    if(ind==0) return 0;
    if(dp[ind]!=-1) return dp[ind];
    int jumpTwo = INT_MAX;
    int jumpOne= solve(ind-1, height,dp)+ abs(height[ind]-height[ind-1]);
    if(ind>1)
        jumpTwo = solve(ind-2, height,dp)+ abs(height[ind]-height[ind-2]);

    return dp[ind]=min(jumpOne, jumpTwo);
}

int main() {

    vector<int> height{30,10,60 , 10 , 60 , 50};
    int n=height.size();
    vector<int> dp(n,-1);
    cout<<solve(n-1,height,dp);
}

```

**Output:** 40

### Time Complexity: O(N)

Reason: The overlapping subproblems will return the answer in constant time O(1). Therefore the total number of new subproblems we solve is 'n'. Hence total time complexity is O(N).

### Space Complexity: O(N)

Reason: We are using a recursion stack space(O(N)) and an array (again O(N)). Therefore total space complexity will be  $O(N) + O(N) \approx O(N)$

### Tabulation approach

- Declare a dp[] array of size n.
- First initialize the base condition values, i.e dp[0] as 0.
- Set an iterative loop which traverses the array( from index 1 to n-1) and for every index calculate jumpOne and jumpTwo and set  $dp[i] = \min(jumpOne, jumpTwo)$ .

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int main() {

    vector<int> height{30,10,60,10,60,50};
    int n=height.size();
    vector<int> dp(n,-1);
    dp[0]=0;
    for(int ind=1;ind<n;ind++){
        int jumpTwo = INT_MAX;
        int jumpOne= dp[ind-1] + abs(height[ind]-height[ind-1]);
        if(ind>1)
            jumpTwo = dp[ind-2] + abs(height[ind]-height[ind-2]);

        dp[ind]=min(jumpOne, jumpTwo);
    }
    cout<<dp[n-1];
}

```

**Output:** 40

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(N)**

Reason: We are using an external array of size ‘n+1’.

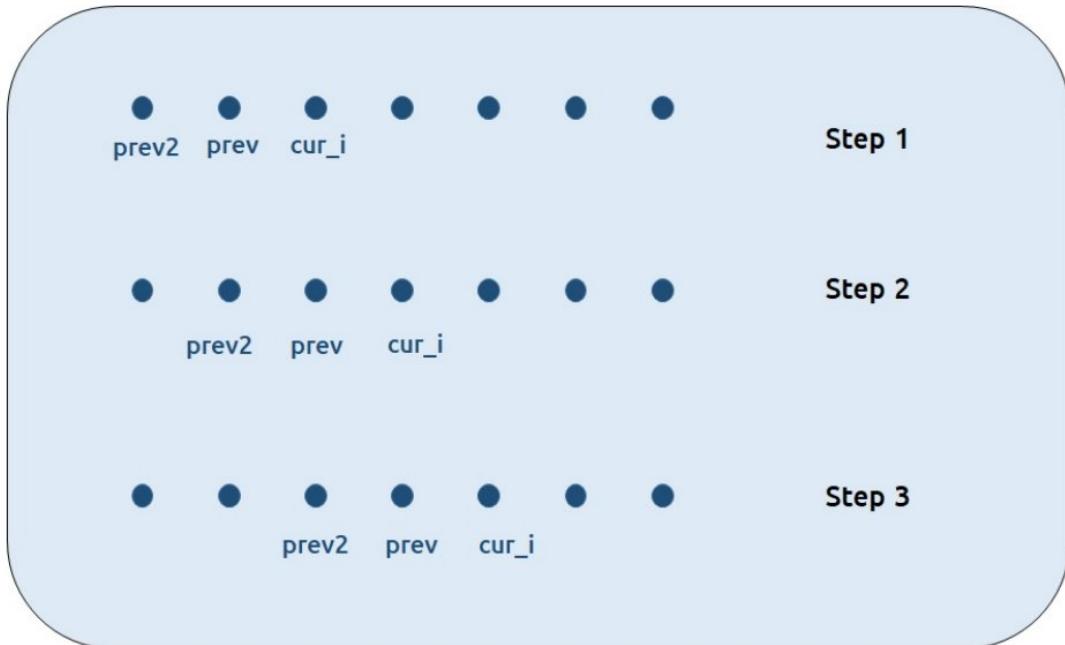
**Part 3: Space Optimization**

If we closely look the values required at every iteration,

**dp[i] , dp[i-1] and dp[i-2]**

we see that for any i, we do need only the last two values in the array. So is there a need to maintain a whole array for it?

The answer is ‘No’. Let us call dp[i-1] as prev and dp[i-2] as prev2. Now understand the following illustration.



- Each iteration's `cur_i` and `prev` becomes the next iteration's `prev` and `prev2` respectively.
- Therefore after calculating `cur_i`, if we update `prev` and `prev2` according to the next step, we will always get the answer.
- After the iterative loop has ended we can simply return `prev` as our answer.

#### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int main() {

    vector<int> height{30,10,60,10,60,50};
    int n=height.size();
    int prev=0;
    int prev2=0;
    for(int i=1;i<n;i++){

        int jumpTwo = INT_MAX;
        int jumpOne= prev + abs(height[i]-height[i-1]);
        if(i>1)
            jumpTwo = prev2 + abs(height[i]-height[i-2]);

        int cur_i=min(jumpOne, jumpTwo);
        prev2=prev;
        prev=cur_i;

    }
    cout<<prev;
}
```

**Output:** 40

**Time Complexity:** O(N)

Reason: We are running a simple iterative loop

**Space Complexity:** O(1)

Reason: We are not using any extra space.

# Dynamic Programming: Frog Jump with k Distances (DP 4)

 [takeuforward.org/data-structure/dynamic-programming-frog-jump-with-k-distances-dp-4/](https://takeuforward.org/data-structure/dynamic-programming-frog-jump-with-k-distances-dp-4/)

January 13, 2022

In this article we will learn about “Dynamic Programming: Frog Jump with k Distances (DP 4)”

**Problem Statement:** Frog Jump with K Distance/ Learn to write 1D DP

## Problem Statement:

This is a follow-up question to “Frog Jump” discussed in [the previous article](#). In the previous question, the frog was allowed to jump either one or two steps at a time. In this question, the frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

## Pre-req: [Frog Jump](#)

## Solution :

We will first see the modifications required in the pseudo-code. Once the recursive code is formed, we can go ahead with the memoization and tabulation.

Here is the pseudocode from the simple Frog Jump problem.

This was the case where we needed to try two options (move a single step and move two steps) in order to try out all the possible ways for the problem.

Now, we need to try K options in order to try out all possible ways.

These are the calls we need to make for K=2, K=3, K=4

```
f(ind,height[ ]) {  
    if( ind == 0) return 0  
  
    jumpOne= f(ind-1, height)+  
            abs(height[ind]- height [ind-1])  
    if (ind>1)  
        jumpTwo= f(ind-2, height)+  
                  abs(height[ind]- height [ind-2])  
  
    return min(jumpOne, jumpTwo)  
}
```

K = 2	K = 3	K = 4
Calls made:	Calls made:	Calls made:
f(ind-1)	f(ind-1)	f(ind-1)
f(ind-2)	f(ind-2)	f(ind-2)
	f(ind-3)	f(ind-3)
		f(ind-4)

If we generalize, we are making K calls, therefore, we can set a for loop to run from 1 to K and in each iteration we can make a function call, corresponding to a step. We will return the minimum step call after the loop.

The final pseudo-code will be:

```
f(ind,height[ ]) {
    if( ind == 0) return 0
    mmSteps = INT_MAX
    for(j=1 ; j<=K ; j++){
        if (ind-j>=0){
            jump = f(ind-j,height)+abs(height[ind]- height [ind-j])
            mmSteps = min(jump, mmSteps)
        }
    }
    return mmSteps
}
```

**Note:** We need to make sure that we are not passing negative index to the array, therefore an extra if condition is used.

Once we form the recursive solution, we can use the approach told in [Dynamic Programming Introduction](#) to convert it into a dynamic programming one.

### Memoization approach

## Steps to convert Recursive code to memoization solution:

- Create a dp[n] array initialized to -1.
- Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array(i.e dp[n] != -1 ). If yes, simply return the value from the dp array.
- If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[n] to the solution we get.

## Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int ind, vector<int>& height, vector<int>& dp, int k){
    if(ind==0) return 0;
    if(dp[ind]!=-1) return dp[ind];

    int mmSteps = INT_MAX;

    for(int j=1;j<=k;j++){
        if(ind-j<=0){
            int jump = solveUtil(ind-j, height, dp, k)+ abs(height[ind]- height[ind-j]);
            mmSteps= min(jump, mmSteps);
        }
    }
    return dp[ind]= mmSteps;
}

int solve(int n, vector<int>& height , int k){
    vector<int> dp(n,-1);
    return solveUtil(n-1, height, dp, k);
}

int main() {

    vector<int> height{30,10,60 , 10 , 60 , 50};
    int n=height.size();
    int k=2;
    vector<int> dp(n,-1);
    cout<<solve(n,height,k);
}
```

**Output:** 40

**Time Complexity:** O(N \*K)

**Reason:** The overlapping subproblems will return the answer in constant time. Therefore the total number of new subproblems we solve is ‘n’. At every new subproblem, we are running another loop for K times. Hence total time complexity is  $O(N * K)$ .

### **Space Complexity: $O(N)$**

**Reason:** We are using a recursion stack space( $O(N)$ ) and an array (again  $O(N)$ ). Therefore total space complexity will be  $O(N) + O(N) \approx O(N)$

### **Tabulation approach**

- Declare a  $dp[]$  array of size n.
- First initialize the base condition values, i.e  $dp[0]$  as 0.
- Set an iterative loop which traverses the array( from index 1 to  $n-1$ ) and for every index calculate  $jumpOne$  and  $jumpTwo$  and set  $dp[i] = \min(jumpOne, jumpTwo)$ .

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int solveUtil(int n, vector<int>& height, vector<int>& dp, int k){
    dp[0]=0;
    for(int i=1;i<n;i++){
        int mmSteps = INT_MAX;

        for(int j=1;j<=k;j++){
            if(i-j>=0){
                int jump = dp[i-j]+ abs(height[i]- height[i-j]);
                mmSteps= min(jump, mmSteps);
            }
        }
        dp[i]= mmSteps;
    }
    return dp[n-1];
}

int solve(int n, vector<int>& height , int k){
    vector<int> dp(n,-1);
    return solveUtil(n, height, dp, k);
}

int main() {

    vector<int> height{30,10,60 , 10 , 60 , 50};
    int n=height.size();
    int k=2;
    vector<int> dp(n,-1);
    cout<<solve(n,height,k);
}

```

**Output:** 40

**Time Complexity: O(N\*K)**

Reason: We are running two nested loops, where outer loops run from 1 to n-1 and the inner loop runs from 1 to K

**Space Complexity: O(N)**

Reason: We are using an external array of size ‘n’.

# Maximum sum of non-adjacent elements (DP 5)

 [takeuforward.org/data-structure/maximum-sum-of-non-adjacent-elements-dp-5](https://takeuforward.org/data-structure/maximum-sum-of-non-adjacent-elements-dp-5)

January 14, 2022

In this article we will solve the problem: **Maximum sum of non-adjacent elements (DP 5)**

## Problem Statement:

Given an array of 'N' positive integers, we need to return the maximum sum of the subsequence such that no two elements of the subsequence are adjacent elements in the array.

**Note:** A subsequence of an array is a list with elements of the array where some elements are deleted ( or not deleted at all) and the elements should be in the same order in the subsequence as in the array.

## Examples:

N = 3	N = 4	N = 9
[1, 2, 4]	[2, 1, 4, 9]	[1, 2, 3, 1, 3, 5, 8, 1, 9]
Output: 5	Output: 11	Output: 24
[1, 2, 4]	[2, 1, 4, 9]	[1, 2, 3, 1, 3, 5, 8, 1, 9]

## Pre-req: Recursion, Dynamic Programming Introduction

## Solution :

As we need to find the sum of subsequences, one approach that comes to our mind is to generate all subsequences and pick the one with the maximum sum.

To generate all the subsequences, we can use the pick/non-pick technique. This technique can be briefly explained as follows:

- At every index of the array, we have two options.
- First, to pick the array element at that index and consider it in our subsequence.
- Second, to leave the array element at that index and not to consider it in our subsequence.

A more detailed explanation of this technique is taught in [Recursion on subsequences](#). Readers are highly advised to watch that video.

First, we will try to form the recursive solution to the problem with the pick/non-pick technique. There is one more catch, the problem wants us to have only non-adjacent elements of the array in the subsequence, therefore we need to address that too.

### Steps to form the recursive solution

We will use the steps mentioned in the article [Dynamic Programming Introduction](#) in order to form our recursive solution.

**Step 1:** Form the function in terms of indexes:

- We are given an array which can be easily thought of in terms of indexes.
- We can define our function  $f(ind)$  as : Maximum sum of the subsequence starting from index 0 to index  $ind$ .
- We need to return  $f(n-1)$  as our final answer.

**Step 2:** Try all the choices to reach the goal.

As mentioned earlier we will use the pick/non-pick technique to generate all subsequences. We also need to take care of the non-adjacent elements in this step.

- If we pick an element then,  $pick = arr[ind] + f(ind-2)$ . The reason we are doing  $f(ind-2)$  is because we have picked the current index element so we need to pick a non-adjacent element so we choose the index 'ind-2' instead of 'ind-1'.
- Next we need to ignore the current element in our subsequence. So  $nonPick = 0 + f(ind-1)$ . As we don't pick the current element, we can consider the adjacent element in the subsequence.

Our pseudocode till this step will be:

**Step 3:** Take the maximum of all the choices

As the problem statement asks to find the maximum subsequence total, we will return the maximum of two choices of step2.

```
f(ind,arr[]) {  
    //base conditions  
  
    pick= arr[ind]+ f(ind-2,arr)  
  
    notPick= 0+ f(ind-1,arr)  
  
}
```

```

f(ind,arr[ ]) {
    //base conditions

    pick= arr[ind]+ f(ind-2,arr)

    notPick= 0+ f(ind-1,arr)

    return max(pick, notPick)

}

```

## Base Conditions

The base conditions for the recursive function will be as follows:

- If  $\text{ind}=0$ , then we know to reach at  $\text{index}=0$ , we would have ignored the element at  $\text{index} = 1$ . Therefore, we can simply return the value of  $\text{arr}[\text{ind}]$  and consider it in the subsequence.
- If  $\text{ind}<0$ , this case can hit when we call  $f(\text{ind}-2)$  at  $\text{ind}=1$ . In this case we want to return to the calling function so we simply return 0 so that nothing is added to the subsequence sum.

Our final pseudo-code will be:

```

f(ind,arr[ ]) {
    if(ind == 0) return arr[ind]

    if(ind < 0) return 0

    pick= arr[ind]+ f(ind-2,arr)

    notPick= 0+ f(ind-1,arr)

    return max(pick, notPick)

}

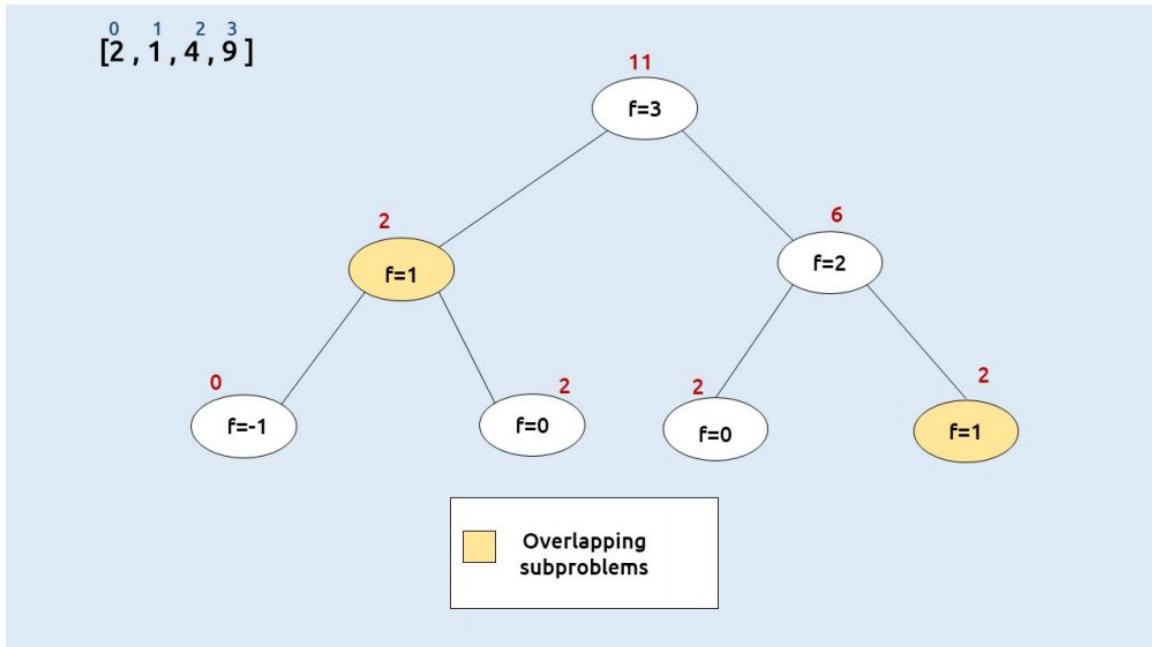
```

Once we form the recursive solution, we can use the approach told in [Dynamic Programming Introduction](#) to convert it into a dynamic programming one.

## Memoization approach

If we observe the recursion tree, we will observe a number of overlapping subproblems. Therefore the recursive solution can be memoized to reduce the time complexity.

### Recursion tree diagram:



**Note:** To watch a detailed dry run of this approach, please watch the video attached below

### Steps to convert Recursive code to memoization solution:

- Create a  $dp[n]$  array initialized to -1.
- Whenever we want to find the answer of a particular value (say  $n$ ), we first check whether the answer is already calculated using the  $dp$  array (i.e  $dp[n] \neq -1$ ). If yes, simply return the value from the  $dp$  array.
- If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[n]$  to the solution we get.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int solveUtil(int ind, vector<int>& arr, vector<int>& dp){

    if(dp[ind]!=-1) return dp[ind];

    if(ind==0) return arr[ind];
    if(ind<0) return 0;

    int pick= arr[ind]+ solveUtil(ind-2, arr, dp);
    int nonPick = 0 + solveUtil(ind-1, arr, dp);

    return dp[ind]=max(pick, nonPick);
}

int solve(int n, vector<int>& arr){
    vector<int> dp(n,-1);
    return solveUtil(n-1, arr, dp);
}

int main() {

    vector<int> arr{2,1,4,9};
    int n=arr.size();
    cout<<solve(n,arr);

}

```

**Output:** 11

### Time Complexity: O(N)

Reason: The overlapping subproblems will return the answer in constant time O(1). Therefore the total number of new subproblems we solve is ‘n’. Hence total time complexity is O(N).

### Space Complexity: O(N)

Reason: We are using a recursion stack space(O(N)) and an array (again O(N)). Therefore total space complexity will be  $O(N) + O(N) \approx O(N)$

### Tabulation approach

- Declare a dp[] array of size n.
- First initialize the base condition values, i.e dp[0] as 0.
- Set an iterative loop which traverses the array( from index 1 to n-1) and for every index calculate pick and nonPick
- And then we can set  $dp[i] = \max (\text{pick}, \text{nonPick})$

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int n, vector<int>& arr, vector<int>& dp){

    dp[0]= arr[0];

    for(int i=1 ;i<n; i++){
        int pick = arr[i];
        if(i>1)
            pick += dp[i-2];
        int nonPick = 0+ dp[i-1];

        dp[i]= max(pick, nonPick);
    }

    return dp[n-1];
}

int solve(int n, vector<int>& arr){
    vector<int> dp(n,-1);
    return solveUtil(n, arr, dp);
}

int main() {

    vector<int> arr{2,1,4,9};
    int n=arr.size();
    cout<<solve(n,arr);

}
```

**Output:** 11

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(N)**

Reason: We are using an external array of size ‘n+1’.

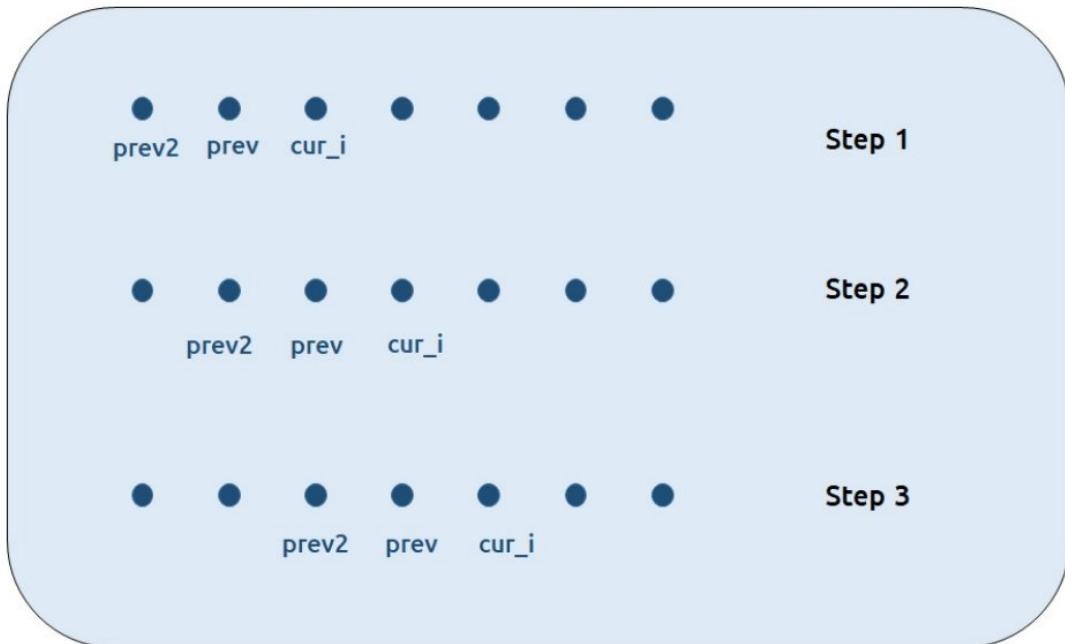
### Part 3: Space Optimization

If we closely look the values required at every iteration,

**dp[i] , dp[i-1] and dp[i-2]**

we see that for any  $i$ , we do need only the last two values in the array. So is there a need to maintain a whole array for it?

The answer is 'No'. Let us call  $dp[i-1]$  as  $prev$  and  $dp[i-2]$  as  $prev2$ . Now understand the following illustration.



- Each iteration's  $cur\_i$  and  $prev$  becomes the next iteration's  $prev$  and  $prev2$  respectively.
- Therefore after calculating  $cur\_i$ , if we update  $prev$  and  $prev2$  according to the next step, we will always get the answer.
- After the iterative loop has ended we can simply return  $prev$  as our answer.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int solve(int n, vector<int>& arr){
    int prev = arr[0];
    int prev2 = 0;

    for(int i=1; i<n; i++){
        int pick = arr[i];
        if(i>1)
            pick += prev2;
        int nonPick = 0 + prev;

        int cur_i = max(pick, nonPick);
        prev2 = prev;
        prev= cur_i;
    }
    return prev;
}

int main() {

    vector<int> arr{2,1,4,9};
    int n=arr.size();
    cout<<solve(n,arr);

}

```

**Output: 11**

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(1)**

Reason: We are not using any extra space.

# Dynamic Programming: House Robber (DP 6)

 [takeuforward.org/data-structure/dynamic-programming-house-robber-dp-6](https://takeuforward.org/data-structure/dynamic-programming-house-robber-dp-6)

January 15, 2022

**Problem Statement:** Dynamic Programming: House Robber (DP 6)

*Problem Statement Link: [House Robber](#)*

A thief needs to rob money in a street. The houses in the street are arranged in a circular manner. Therefore the first and the last house are adjacent to each other. The security system in the street is such that if adjacent houses are robbed, the police will get notified.

Given an array of integers “Arr” which represents money at each house, we need to return the maximum amount of money that the thief can rob without alerting the police.

**Examples:**

$N = 4$	$N = 5$
$[2, 1, 4, 9]$	$[1, 5, 2, 1, 6]$
Output: <b>10</b>	Output: <b>11</b>
$[2, \textcolor{blue}{1}, 4, 9]$	$[1, \textcolor{blue}{5}, 2, 1, 6]$

The table shows two examples of circular house arrangements. The left example (N=4) has houses with values [2, 1, 4, 9]. The thief can rob houses 2 and 4, totaling 10, as indicated by the circled numbers. The right example (N=5) has houses with values [1, 5, 2, 1, 6]. The thief can rob houses 1, 3, and 5, totaling 11, as indicated by the circled numbers.

**Pre-req:** [Maximum Sum of non-adjacent elements](#)

**Solution :**

This question can be solved using the approach discussed in the [Maximum Sum of non-adjacent elements](#). Readers are highly advised to go through that article first and then read this. The rest of the article will refer to the previous article as Article DP5 and will relate to that approach.

Now, we have a single test case. Three houses have money as shown.

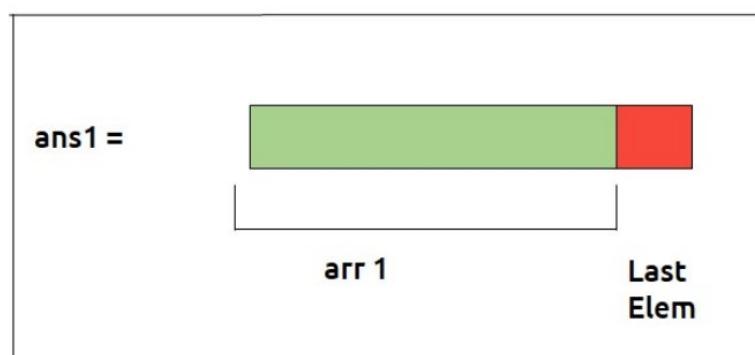
$N = 3$  $[2, 3, 2]$  <b>Answer:</b> 4  $(2, 3, 2)$  <b>Article DP5 Approach</b>	$N = 3$  $[2, 3, 2]$  <b>Answer:</b> 3  $[2, 3, 2]$  <b>This Question</b>
--	---

- According to article DP5, the answer will be 4( $2+2$ ) as we are taking the maximum sum of non-adjacent elements.
- In this question, the first and last element are also adjacent(circular street), therefore the answer will be 3.

### Modification to Article DP5's Approach

We were finding the maximum sum of non-adjacent elements in the previous questions. For a circular street, the first and last house are adjacent, therefore one thing we know for sure is that the answer will not consider the first and last element simultaneously (as they are adjacent).

Now building on the article DP5, we can say that maybe the last element is not considered in the answer. In that case, we can consider the first element. Let's call this answer ans1. Hence we have reduced our array(arr- last element), say arr1, and found ans1 on it by using the article DP5 approach.



Now, it can also happen that the final answer does consider the last element. If we consider the last element, we can't consider the first element( again adjacent elements). We again use the same approach on our reduced array( arr – first element), say arr2. Let's call the answer we get as ans2.

Now, the final answer can be either ans1 or ans2. As we have to return the maximum money robbed by the robber, we will return  $\max(\text{ans1}, \text{ans2})$  as our final answer.



### Approach:

The approach to solving this problem can be summarized as:

- Make two reduced arrays – arr1(arr-last element) and arr2(arr-first element).
- Find the maximum of non-adjacent elements as mentioned in article DP5 on arr1 and arr2 separately. Let's call the answers we got as ans1 and ans2 respectively.
- Return  $\max(\text{ans1}, \text{ans2})$  as our final answer.

### Code:

Java Code

```

#include <bits/stdc++.h>

using namespace std;

long long int solve(vector<int>& arr){
    int n = arr.size();
    long long int prev = arr[0];
    long long int prev2 = 0;

    for(int i=1; i<n; i++){
        long long int pick = arr[i];
        if(i>1)
            pick += prev2;
        int long long nonPick = 0 + prev;

        long long int cur_i = max(pick, nonPick);
        prev2 = prev;
        prev= cur_i;

    }
    return prev;
}

long long int robStreet(int n, vector<int> &arr){
    vector<int> arr1;
    vector<int> arr2;

    if(n==1)
        return arr[0];

    for(int i=0; i<n; i++){

        if(i!=0) arr1.push_back(arr[i]);
        if(i!=n-1) arr2.push_back(arr[i]);
    }

    long long int ans1 = solve(arr1);
    long long int ans2 = solve(arr2);

    return max(ans1,ans2);
}
}

int main() {

    vector<int> arr{1,5,1,2,6};
    int n=arr.size();
    cout<<robStreet(n,arr);
}

```

**Output:** 11

**Time Complexity:** O(N )

Reason: We are running a simple iterative loop, two times. Therefore total time complexity will be  $O(N) + O(N) \approx O(N)$

### **Space Complexity: O(1)**

Reason: We are not using extra space.

# Dynamic Programming: Ninja's Training (DP 7)

 [takeuforward.org/data-structure/dynamic-programming-ninjas-training-dp-7](https://takeuforward.org/data-structure/dynamic-programming-ninjas-training-dp-7)

January 18, 2022

## Introduction To 2D Dynamic Programming / Ninja Training

In this article, we will understand the concept of 2D dynamic programming. We will use the problem 'Ninja Training' to understand this concept.

Pre-req: [Dynamic Programming Introduction](#)

### Problem Link: [Ninja Training](#)

A Ninja has an 'N' Day training schedule. He has to perform one of these three activities (Running, Fighting Practice, or Learning New Moves) each day. There are merit points associated with performing an activity each day. The same activity can't be performed on two consecutive days. We need to find the maximum merit points the ninja can attain in N Days.

We are given a 2D Array POINTS of size 'N\*3' which tells us the merit point of specific activity on that particular day. Our task is to calculate the maximum number of merit points that the ninja can earn.

**Example:**

```
Days = 3
Points =    10, 40, 70    // Day 0
            20, 50, 80    // Day 1
            30, 60, 90    // Day 2
Output: 210    // 70 (Day 0 )+ 50 (Day 2)+ 90 (Day 3)
```

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: Recursion

**Solution :**

### Why a Greedy Solution doesn't work?

The first approach that comes to our mind is the greedy approach. We will see with an example how a greedy solution doesn't give the correct solution.

Consider this example:

**Days = 2**

**Points =      10, 50, 1      // Day 0**  
                  5, 100, 11     // Day 1

We want to know the maximum amount of merit points. For the greedy approach, we will consider the maximum point activity each day, respecting the condition that activity can't be performed on consecutive days.

- On Day 0, we will consider the activity with maximum points i.e 50.
- On Day 1, the maximum point activity is 100 but we can't perform the same activity in two consecutive days. Therefore we will take the next maximum point activity of 11 points.
- Total Merit points by Greedy Solution :  $50+11 = 61$

As this is a small example we can clearly see that we have a better approach, to consider activity with 10 points on day0 and 100 points on day1. It gives us the total merit points as 110 which is better than the greedy solution.

**Days = 2**

**Points =      10, 50, 1      // Day 0**  
                  5, 100, 11     // Day 1

**Greedy Solution**

**Answer : 61**

**(50+11)**

**Non-Greedy Solution**

**Answer : 110**

**(10+100)**

So we see that the greedy solution restricts us from choices and we can lose activity with better points on the next day in the greedy solution. Therefore, it is better to try out all the possible choices as our next solution. We will use **recursion** to generate all the possible choices.

### **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

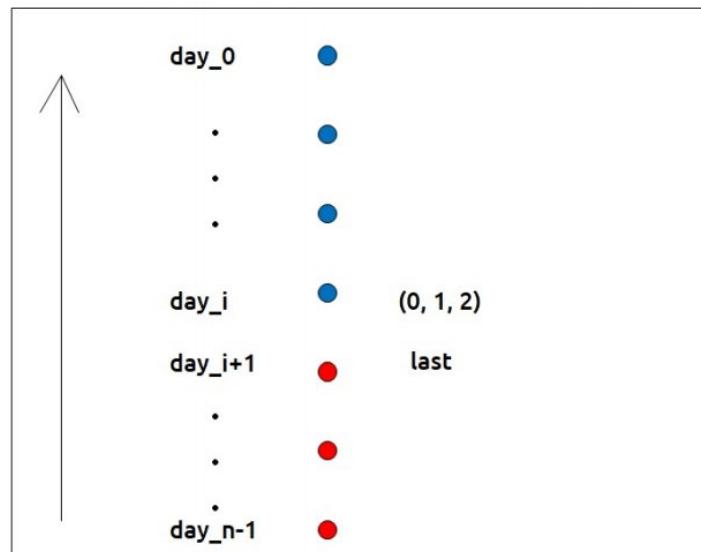
**Step 1:** Express the problem in terms of indexes.

Now given an example, what is the one clear parameter which breaks the problem in different steps?

It is the number of days. Clearly, we have n days (from 0 to n-1), so one changing parameter which can be expressed in terms of indexes is '**day**'.

Every day we have the option of three activities(say 0,1,2) to be performed. Suppose that we are at a day<sub>i</sub> ( shown in the fig) in the example given below. What is one more parameter along with the day that we must know to try out the correct choices at day<sub>i</sub>?

**f(day, ...)**



It is the '**last**' choice that we tried out on day<sub>i+1</sub> (  $i+1$  in case of top-down recursion). Unless we know the last choice we made, how can we decide whether a choice we are making is correct or not?

Now there are three options each day(say 0,1,2) which becomes the '**last**' of the next day. If we are just starting from the day<sub>n-1</sub>, then for the day<sub>n-1</sub> we can try all three options. We can say '**last**' for day<sub>n-1</sub> is 3.

last	Activities that can be considered for current day
0	1 and 2
1	0 and 2
2	0 and 1
3	0 , 1 and 2

Therefore our function will take two parameters – **day** and **last**.

**f(day, last) -> The maximum points till given day with the last option on previous day**

**Step 2:** Try out all possible choices at a given index.

We are writing a top-down recursive function. We start from day\_n-1 to day\_0. Therefore whenever we call the recursive function for the next day we call it for f(day-1, //second parameter).

Now let's discuss the second parameter. The choices we have for a current day depend on the 'last' variable. If we are at our **base case** (day=0), we will have the following choices.

day=0,last =	Options to consider
0	points[0][1], points [0][2]
1	points[0][0], points [0][2]
2	points[0][0], points [0][1]
3	points[0][0], points [0][1] and points[0][2]

Other than the base case, whenever we perform an activity 'i' its merit points will be given by **points[day][i]** and to get merit points of the remaining days we will let recursion do its job by passing **f(d-1, i)**. We are passing the second parameter as i because the current day's activity will become the next day's last.

**Step 3: Take the maximum of all choices**

As the problem statement wants us to find the maximum merit points, we will take the maximum of all choices.

The final pseudocode after steps 1, 2, and 3:

```

f(day,last) {
    // base case
    if( day == 0){
        maxi = 0
        for(int i=0; i<=2; i++){
            if(i != last){
                maxi = max(maxi, points[0][i])
            }
        }
        return maxi
    }

    maxi = 0
    for(int i=0; i<=2; i++){
        if(i != last){
            activity = points[day][i] + f(day-1,i)
            maxi = max(maxi, activity)
        }
    }
    return maxi
}

```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution to the following steps will be taken:

1. Create a dp array of size [n][4]. There are total ‘n’ days and for every day, there can be 4 choices (0,1,2 and 3). Therefore we take the dp array as dp[n][4].
2. Whenever we want to find the answer of particular parameters (say f(day,last)), we first check whether the answer is already calculated using the dp array(i.e dp[day][last]!= -1 ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[day][last] to the solution we get.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int f(int day, int last, vector<vector<int>> &points, vector<vector<int>> &dp) {

    if (dp[day][last] != -1) return dp[day][last];

    if (day == 0) {
        int maxi = 0;
        for (int i = 0; i <= 2; i++) {
            if (i != last)
                maxi = max(maxi, points[0][i]);
        }
        return dp[day][last] = maxi;
    }

    int maxi = 0;
    for (int i = 0; i <= 2; i++) {
        if (i != last) {
            int activity = points[day][i] + f(day - 1, i, points, dp);
            maxi = max(maxi, activity);
        }
    }

    return dp[day][last] = maxi;
}

int ninjaTraining(int n, vector < vector < int > > & points) {

    vector < vector < int > > dp(n, vector < int > (4, -1));
    return f(n - 1, 3, points, dp);
}

int main() {

    vector < vector < int > > points = {{10,40,70},
                                         {20,50,80},
                                         {30,60,90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}

```

**Output:** 210

**Time Complexity: O(N\*4\*3)**

Reason: There are  $N \times 4$  states and for every state, we are running a for loop iterating three times.

**Space Complexity: O(N) + O(N\*4)**

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (again  $O(N^*4)$ ).  
Therefore total space complexity will be  $O(N) + O(N) \approx O(N)$

### **Steps to convert Recursive Solution to Tabulation one.**

- Declare a dp[] array of size  $[n][4]$
- First initialize the base condition values. We know that base condition arises when day = 0. Therefore, we can say that the following will be the base conditions
  - $dp[0][0] = \max(points[0][1], points[0][2])$
  - $dp[0][1] = \max(points[0][0], points[0][2])$
  - $dp[0][2] = \max(points[0][0], points[0][1])$
  - $dp[0][3] = \max(points[0][0], points[0][1] \text{ and } points[0][2])$
- Set an iterative loop which traverses dp array (from index 1 to n) and for every index set its value according to the recursive logic

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector < vector < int > > & points) {

    vector < vector < int > > dp(n, vector < int > (4, 0));

    dp[0][0] = max(points[0][1], points[0][2]);
    dp[0][1] = max(points[0][0], points[0][2]);
    dp[0][2] = max(points[0][0], points[0][1]);
    dp[0][3] = max(points[0][0], max(points[0][1], points[0][2]));

    for (int day = 1; day < n; day++) {
        for (int last = 0; last < 4; last++) {
            dp[day][last] = 0;
            for (int task = 0; task <= 2; task++) {
                if (task != last) {
                    int activity = points[day][task] + dp[day - 1][task];
                    dp[day][last] = max(dp[day][last], activity);
                }
            }
        }
    }

    return dp[n - 1][3];
}

int main() {

    vector<vector<int>> points = {{10, 40, 70},
                                    {20, 50, 80},
                                    {30, 60, 90}};
    int n = points.size();
    cout << ninjaTraining(n, points);
}

```

**Output:** 210

**Time Complexity:**  $O(N^4 \cdot 3)$

Reason: There are three nested loops

**Space Complexity:**  $O(N^4)$

Reason: We are using an external array of size ' $N^4$ '.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[day][last] = \max(dp[day][last], points[day][task] + dp[day-1][task])$$

Here the task can be anything from 0 to 3 and day-1 is the previous stage of recursion. So in order to compute any dp array value, we only require the last row to calculate it.

<b>day \ last</b>	0	1	2	3
0	✓	✓	✓	✓
1				
.				
.				
.				
n-1				

This row is initially filled

This row only needs previous row values

$$dp[day][last] = \dots + dp[day-1][task]$$

<b>day \ last</b>	0	1	2	3
0				
1	✓	✓	✓	✓
.				
.				
.				
n-1				

This row is filled in previous step

This row only needs previous row values

$$dp[day][last] = \dots + dp[day-1][task]$$

- So rather than storing the entire 2D Array of size  $N*4$ , we can just store values of size 4 (say prev).
- We can then take a dummy array, again of size 4 (say temp) and calculate the next row's value using the array we stored in step 1.
- After that whenever we move to the next day, the temp array becomes our prev for the next step.
- At last  $prev[3]$  will give us the answer.

## Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector < vector < int > > & points) {

    vector < int > prev(4, 0);

    prev[0] = max(points[0][1], points[0][2]);
    prev[1] = max(points[0][0], points[0][2]);
    prev[2] = max(points[0][0], points[0][1]);
    prev[3] = max(points[0][0], max(points[0][1], points[0][2]));

    for (int day = 1; day < n; day++) {

        vector < int > temp(4, 0);
        for (int last = 0; last < 4; last++) {
            temp[last] = 0;
            for (int task = 0; task <= 2; task++) {
                if (task != last) {
                    temp[last] = max(temp[last], points[day][task] + prev[task]);
                }
            }
        }
        prev = temp;
    }

    return prev[3];
}

int main() {

    vector<vector<int>> points = {{10,40,70},
                                    {20,50,80},
                                    {30,60,90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}
```

## Output:

210

**Time Complexity: O(N\*4\*3)**

Reason: There are three nested loops

## **Space Complexity: O(4)**

Reason: We are using an external array of size '4' to store only one row.

# Grid Unique Paths : DP on Grids (DP8)

 [takeuforward.org/data-structure/grid-unique-paths-dp-on-grids-dp8](https://takeuforward.org/data-structure/grid-unique-paths-dp-on-grids-dp8)

January 22, 2022

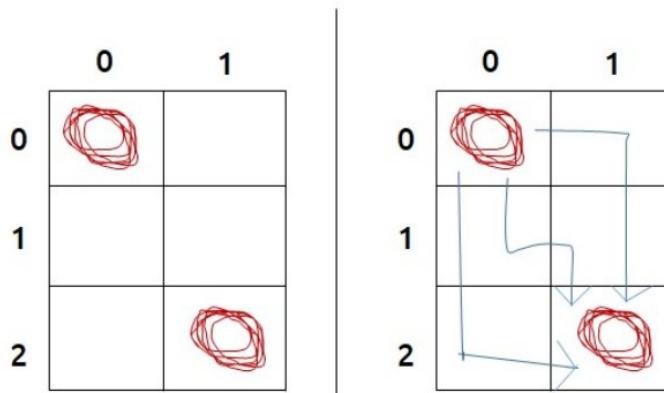
In this article, we will solve the most asked coding interview problem: Grid Unique Paths

Given two values M and N, which represent a matrix[M][N]. We need to find the total unique paths from the top-left cell (matrix[0][0]) to the rightmost cell (matrix[M-1][N-1]).

At any cell we are allowed to move in only two directions:- bottom and right.

**Example:**

M = 3 , N = 2



**Output : 3**

**There are three  
unique paths to go**

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** [Patterns in Recursion](#)

**Solution :**

As we have to count all possible ways to go from matrix[0,0] to matrix[m-1,n-1], we can try recursion to generate all possible paths.

**Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two variables M and N, representing the dimensions of a 2D matrix. For every different problem, this M and N will change.

We can define the function with two parameters i and j, where i and j represent the row and column of the matrix.

**f(i,j) -> Total amount of unique paths from matrix[0,0] to matrix[i][j].**

$f(i,j)$  will give us a sub-answer for the region (marked in blue) below:

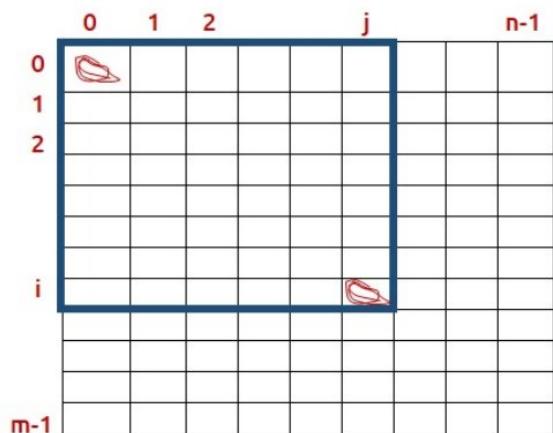
We will be doing a top-down recursion, i.e we will move from the cell[M-1][N-1] and try to find our way to the cell[0][0]. Therefore at every index, we will try to move up and towards the left.

#### Base Case:

As discussed in [Patterns in Recursion](#), there will be two base cases:

- When  $i=0$  and  $j=0$ , that is we have reached the destination so we can count the current path that is going on, hence we return 1.
- When  $i < 0$  and  $j < 0$ , it means that we have crossed the boundary of the matrix and we couldn't find a right path, hence we return 0.

**Given M,N**



**f(i,j) gives us the total unique paths from cell (0,0) to cell (i,j)**

The pseudocode till this step will be:

**Step 2:** Try out all possible choices at a given index.

As we are writing a top-down recursion, at every index we have two choices, one to go up( $\uparrow$ ) and the other to go left( $\leftarrow$ ). To go upwards, we will reduce i by 1, and move towards left we will reduce j by 1.

**Step 3: Take the maximum of all choices**

As we have to **count** all the possible unique paths, we will return the **sum** of the choices(up and left)

The final pseudocode after steps 1, 2, and 3:

#### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [m][n]
2. Whenever we want to find the answer of a particular row and column (say  $f(i,j)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[i][j] \neq -1$ ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j]$  to the solution we get.

#### Code:

- C++ Code
- Java Code

```
f(i,j) {  
  
    if( i==0 && j==0)  return 1  
  
    if( i<0 || j<0) return 0  
  
}
```

```
f(i,j) {  
  
    if( i==0 && j==0)  return 1  
  
    if( i<0 || j<0) return 0  
  
    up = f(i-1,j)  
    left = f(i,j-1)  
  
}
```

```

#include <bits/stdc++.h>

using namespace std;

int countWaysUtil(int i, int j,
vector<vector<int> > &dp) {
    if(i==0 && j == 0)
        return 1;
    if(i<0 || j<0)
        return 0;
    if(dp[i][j]!=-1) return dp[i][j];

    int up = countWaysUtil(i-1,j,dp);
    int left = countWaysUtil(i,j-1,dp);

    return dp[i][j] = up+left;
}

int countWays(int m, int n){
    vector<vector<int> >
dp(m,vector<int>(n,-1));
    return countWaysUtil(m-1,n-1,dp);
}

int main() {

    int m=3;
    int n=2;

    cout<<countWays(m,n);
}

```

### **Output:**

3

### **Time Complexity: O(M\*N)**

Reason: At max, there will be M\*N calls of recursion.

### **Space Complexity: O((N-1)+(M-1)) + O(M\*N)**

Reason: We are using a recursion stack space:O((N-1)+(M-1)), here (N-1)+(M-1) is the path length and an external DP Array of size ‘M\*N’.

### **Steps to convert Recursive Solution to Tabulation one.**

Tabulation is the bottom-up approach, which means we will go from the base case to the main problem.

The steps to convert to the tabular solution are given below:

```

f(i,j) {

    if( i==0 && j==0) return 1

    if( i<0 || j<0) return 0

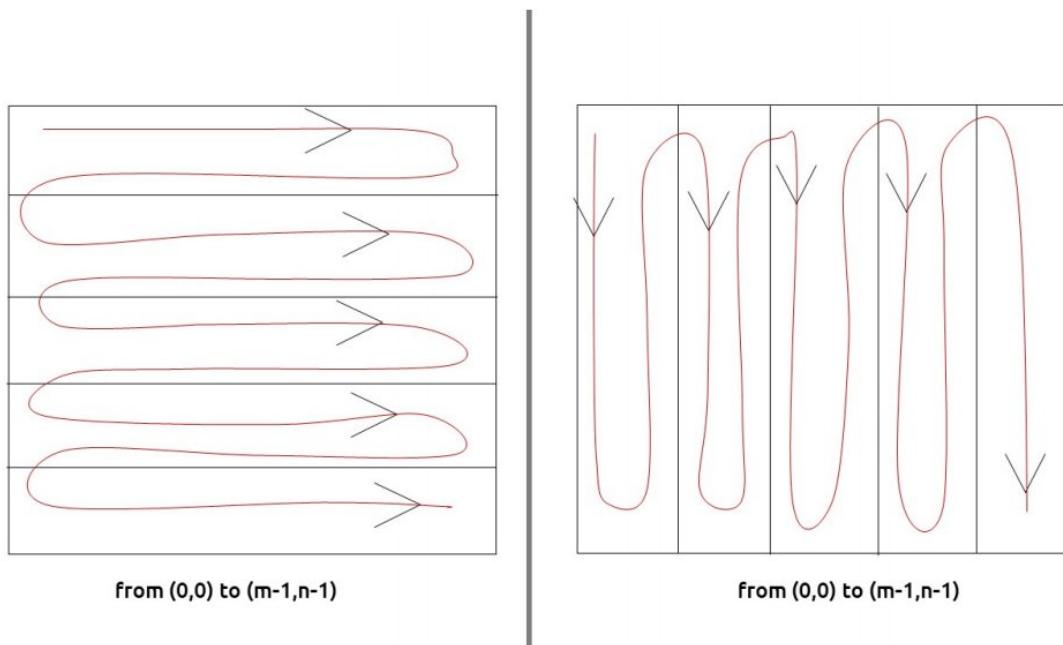
    up = f(i-1,j)

    left = f(i,j-1)

    return up+left
}

```

- Declare a  $dp[]$  array of size  $[m][n]$ .
- First initialize the base condition values, i.e  $dp[0][0] = 1$
- Our answer should get stored in  $dp[m-1][n-1]$ . We want to move from  $(0,0)$  to  $(m-1,n-1)$ . But we can't move arbitrarily, we should move such that at a particular  $i$  and  $j$ , we have all the values required to compute  $dp[i][j]$ .
- If we see the memoized code, values required for  $dp[i][j]$  are:  $dp[i-1][j]$  and  $dp[i][j-1]$ . So we only use the previous row and column value.
- We have already filled the top-left corner ( $i=0$  and  $j=0$ ), if we move in any of the two following ways(given below), at every cell we do have all the previous values required to compute its value.



- We can use two nested loops to have this traversal
- At every cell we calculate up and left as we had done in the recursive solution and then assign the cell's value as  $(up+left)$

**Note:** For the first row and first column (except for the top-left cell), then up and left values will be zero respectively.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int countWaysUtil(int m, int n, vector<vector<int> > &dp) {
    for(int i=0; i<m ;i++){
        for(int j=0; j<n; j++){

            //base condition
            if(i==0 && j==0){
                dp[i][j]=1;
                continue;
            }

            int up=0;
            int left = 0;

            if(i>0)
                up = dp[i-1][j];
            if(j>0)
                left = dp[i][j-1];

            dp[i][j] = up+left;
        }
    }

    return dp[m-1][n-1];
}

int countWays(int m, int n){
    vector<vector<int> > dp(m, vector<int>(n, -1));
    return countWaysUtil(m, n, dp);
}

int main() {

    int m=3;
    int n=2;

    cout<<countWays(m, n);
}

```

### **Output:**

3

### **Time Complexity: O(M\*N)**

Reason: There are two nested loops

### **Space Complexity: O(M\*N)**

Reason: We are using an external array of size 'M\*N'.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

We see that we only need the previous row and column, in order to calculate  $dp[i][j]$ .  
Therefore we can space optimize it.

Initially, we can take a dummy row ( say prev) and initialize it as 0.

Now the current row(say temp) **only needs the** previous row value and the current row's value in order to calculate  $dp[i][j]$ .

		prev[n]	0	0	0	0
temp[n]	j	0	1	...	n-1	
	0	1	1			
	1					
	.					
	.					
	m-1					

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

This row is initialized to 0

This row's cells only need row prev's values and current row's previous values, and first cell is initialized to 1

At the next step, the temp array becomes the prev of the next step and using its values we can still calculate the next row's values.

$i \backslash j$	0	1	...	$n-1$
prev[n]	0	1	1	1
temp[n]	1	1	2	
.				
.				
.				
m-1				

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

This row's cells only need row prev's values and current row's previous values

At last  $prev[n-1]$  will give us the required answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int countWays(int m, int n){
    vector<int> prev(n, 0);
    for(int i=0; i<m; i++){
        vector<int> temp(n, 0);
        for(int j=0; j<n; j++){
            if(i==0 && j==0){
                temp[j]=1;
                continue;
            }

            int up=0;
            int left =0;

            if(i>0)
                up = prev[j];
            if(j>0)
                left = temp[j-1];

            temp[j] = up + left;
        }
        prev = temp;
    }

    return prev[n-1];
}

```

int main() {

```

    int m=3;
    int n=2;

    cout<<countWays(m,n);
}
```

### **Output:**

3

### **Time Complexity: O(M\*N)**

Reason: There are two nested loops

### **Space Complexity: O(N)**

Reason: We are using an external array of size ‘N’ to store only one row.

# Grid Unique Paths 2 (DP 9)

 [takeuforward.org/data-structure/grid-unique-paths-2-dp-9](https://takeuforward.org/data-structure/grid-unique-paths-2-dp-9)

January 22, 2022

In this article, we will solve the most asked coding interview problem: Grid Unique Paths 2.

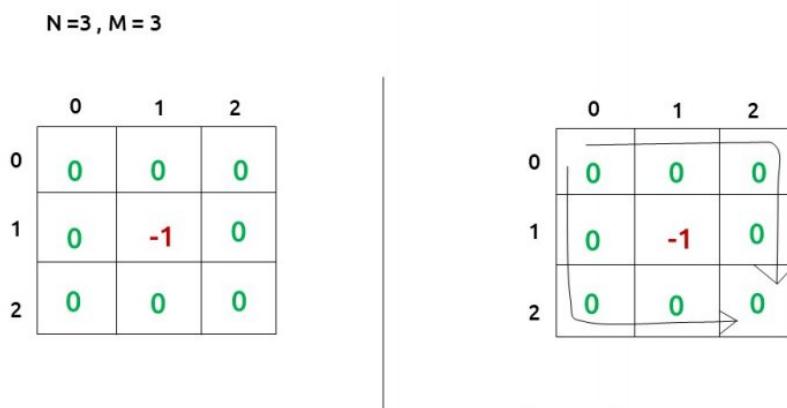
## Problem Link: Maze Obstacles

### Problem Description:

We are given an “N\*M” Maze. The maze contains some obstacles. A cell is ‘blockage’ in the maze if its value is -1. 0 represents non-blockage. There is no path possible through a blocked cell.

We need to count the total number of unique paths from the top-left corner of the maze to the bottom-right corner. At every cell, we can move either down or towards the right.

**Example:**



**Note:** If the question asks to return the answer by performing a modulo operation, please do so.

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** Grid Unique Paths

**Solution :**

This question is a slight modification of the question discussed in [/\\*\\*\\* DP 8 Link \\*\\*\\*/](#). In the previous problem, there were no obstacles whereas this problem has them.

Let us look at this example:

A path through the cell[1][1] is not possible as it is blocked, therefore we need to count every other legit path which **doesn't include** the cell[1][1].

$$N = 3, M = 3$$

	0	1	2
0	0	0	0
1	0	-1	0
2	0	0	0

$$N = 3, M = 3$$



	0	1	2
0	0	0	0
1	0	-1	0
2	0	0	0

These two paths doesn't pass through the blocked cell.

	0	1	2
0	0	0	0
1	0	-1	0
2	0	0	0

These paths are not allowed as they pass through the blocked cell.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two variables N and M, representing the dimensions of the maze.

We can define the function with two parameters i and j, where i and j represent the row and column of the maze.

$f(i,j) \rightarrow$  Total amount of unique paths from matrix[0,0] to matrix[i][j].

We will form  $f(i,j)$  in such a way that it will handle the obstacles.

We will be doing a top-down recursion, i.e we will move from the cell[M-1][N-1] and try to find our way to the cell[0][0]. Therefore at every index, we will try to move up and towards the left.

### Base Case:

There will be three base cases:

- When  $i > 0$  and  $j > 0$  and  $\text{mat}[i][j] = -1$ , it means that the current cell is an obstacle, so we can't find a path from here. Therefore, we return 0.
- When  $i = 0$  and  $j = 0$ , that is we have reached the destination so we can count the current path that is going on, hence we return 1.
- When  $i < 0$  and  $j < 0$ , it means that we have crossed the boundary of the matrix and we couldn't find a right path, hence we return 0.

The pseudocode till this step will be:

```
f(i,j) {  
    if( i>0 && j>0 && mat[i][j]==-1)  
        return 0  
    if( i==0 && j==0)  return 1  
    if( i<0 || j<0) return 0  
}
```

**Step 2:** Try out all possible choices at a given index.

As we are writing a top-down recursion, at every index we have two choices, one to go up( $\uparrow$ ) and the other to go left( $\leftarrow$ ). To go upwards, we will reduce  $i$  by 1, and move towards left we will reduce  $j$  by 1.

```

f(i,j) {
    if( i>0 && j>0 && mat[i][j]==-1)
        return 0
    if( i==0 && j==0) return 1
    if( i<0 || j<0) return 0

    up = f(i-1,j)
    left = f(i,j-1)

}

```

### Step 3: Take the maximum of all choices

As we have to **count** all the possible unique paths, we will return the **sum** of the choices(up and left)

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {
    if( i>0 && j>0 && mat[i][j]==-1)
        return 0
    if( i==0 && j==0) return 1
    if( i<0 || j<0) return 0

    up = f(i-1,j)
    left = f(i,j-1)

    return up+left

}

```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][m]$
2. Whenever we want to find the answer of a particular row and column (say  $f(i,j)$ ), we first check whether the answer is already calculated using the dp array (i.e  $dp[i][j] \neq -1$ ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j]$  to the solution we get.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mazeObstaclesUtil(int i, int j, vector<vector<int> > &maze, vector<vector<int>
>
&dp) {
    if(i>0 && j>0 && maze[i][j]==-1) return 0;
    if(i==0 && j == 0)
        return 1;
    if(i<0 || j<0)
        return 0;
    if(dp[i][j]!=-1) return dp[i][j];

    int up = mazeObstaclesUtil(i-1,j,maze,dp);
    int left = mazeObstaclesUtil(i,j-1,maze,dp);

    return dp[i][j] = up+left;
}

int mazeObstacles(int n, int m, vector<vector<int> > &maze){
    vector<vector<int> > dp(n, vector<int>(m,-1));
    return mazeObstaclesUtil(n-1,m-1,maze,dp);
}

int main() {

    vector<vector<int> > maze{{0,0,0},
                                {0,-1,0},
                                {0,0,0}};

    int n = maze.size();
    int m = maze[0].size();

    cout<<mazeObstacles(n,m,maze);
}

```

### **Output:**

2

### **Time Complexity: O(N\*M)**

Reason: At max, there will be N\*M calls of recursion.

### **Space Complexity: O((M-1)+(N-1)) + O(N\*M)**

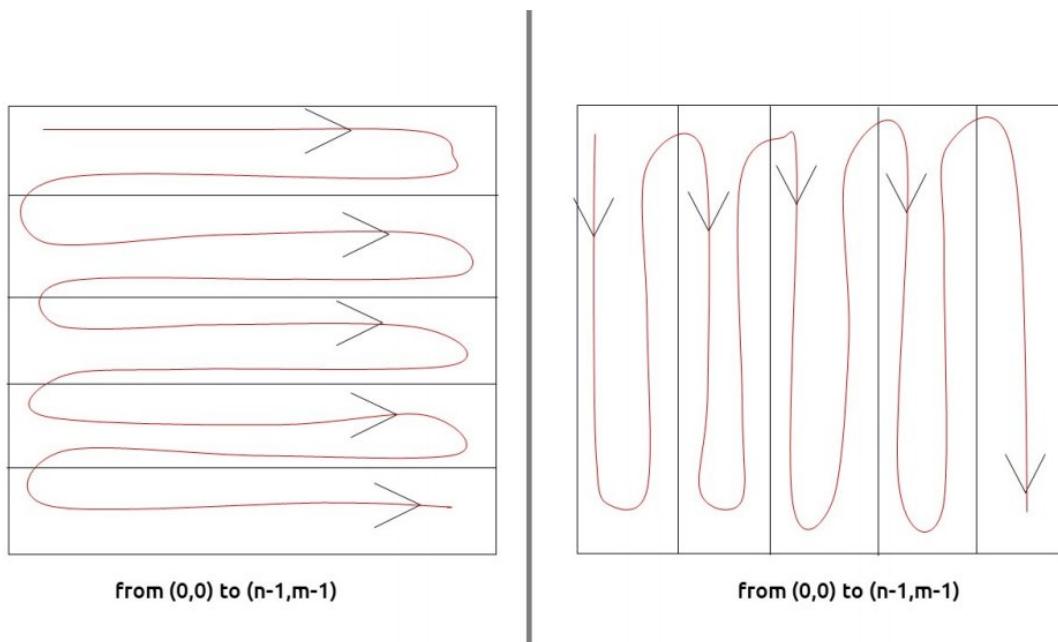
Reason: We are using a recursion stack space:O((M-1)+(N-1)), here (M-1)+(N-1) is the path length and an external DP Array of size ‘N\*M’.

### **Steps to convert Recursive Solution to Tabulation one.**

Tabulation is the bottom-up approach, which means we will go from the base case to the main problem.

The steps to convert to the tabular solution are given below:

- Declare a  $dp[]$  array of size  $[n][m]$ .
- First initialize the base condition values, i.e  $dp[0][0] = 1$
- Our answer should get stored in  $dp[n-1][m-1]$ . We want to move from  $(0,0)$  to  $(n-1, m-1)$ . But we can't move arbitrarily, we should move such that at a particular  $i$  and  $j$ , we have all the values required to compute  $dp[i][j]$ .
- If we see the memoized code, values required for  $dp[i][j]$  are:  $dp[i-1][j]$  and  $dp[i][j-1]$ . So we only use the previous row and column value.
- We have already filled the top-left corner ( $i=0$  and  $j=0$ ), if we move in any of the two following ways(given below), at every cell we do have all the previous values required to compute its value.



- We can use two nested loops to have this traversal
- Whenever  $i > 0$ ,  $j > 0$  and  $mat[i][j] == -1$ , we will simply mark  $dp[i][j] = 0$ , it means that this cell is a blocked one and no path is possible through it.
- At every cell we calculate up and left as we had done in the recursive solution and then assign the cell's value as  $(up+left)$

**Note:** For the first row and first column (except for the top-left cell), then up and left values will be zero respectively.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mazeObstaclesUtil(int n, int m, vector<vector<int>> &maze, vector<vector<int>>
&dp)
{
    for(int i=0; i<n ;i++){
        for(int j=0; j<m; j++){

            //base conditions
            if(i>0 && j>0 && maze[i][j]==-1){
                dp[i][j]=0;
                continue;
            }
            if(i==0 && j==0){
                dp[i][j]=1;
                continue;
            }

            int up=0;
            int left = 0;

            if(i>0)
                up = dp[i-1][j];
            if(j>0)
                left = dp[i][j-1];

            dp[i][j] = up+left;
        }
    }

    return dp[n-1][m-1];
}

int mazeObstacles(int n, int m, vector<vector<int> > &maze){
    vector<vector<int> > dp(n, vector<int>(m, -1));
    return mazeObstaclesUtil(n,m,maze,dp);
}

int main() {

    vector<vector<int> > maze{{0,0,0},
                                {0,-1,0},
                                {0,0,0}};

    int n = maze.size();
    int m = maze[0].size();

    cout<<mazeObstacles(n,m,maze);
}

```

## Output:

## Time Complexity: O(N\*M)

Reason: There are two nested loops

## Space Complexity: O(N\*M)

Reason: We are using an external array of size 'N\*M'.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

We see that we only need the previous row and column, in order to calculate  $dp[i][j]$ . Therefore we can space optimize it.

Initially, we can take a dummy row (say prev) and initialize it as 0.

Now the current row(say temp) **only needs the** previous row value and the current row's value in order to calculate  $dp[i][j]$ .

		prev[n]					
		0	0	0	0		
		i \ j	0	1	...	m-1	
temp[n]		0	1	1			This row is initialized to 0
0	1						This row's cells only need row prev's values and current row's previous values, and first cell is initialized to 1
1							
.							
.							
n-1							

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

At the next step, the temp array becomes the prev of the next step and using its values we can still calculate the next row's values.

	$i \searrow j$	0	1	...	$m-1$
$prev[n]$	0	1	1	1	1
$temp[n]$	1	1	2		
.					
.					
.					
$n-1$					

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

This row's cells only need row prev's values and current row's previous values

At last  $prev[n-1]$  will give us the required answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mazeObstacles(int n, int m, vector<vector<int> > &maze){

    vector<int> prev(m, 0);
    for(int i=0; i<n; i++){
        vector<int> temp(m, 0);
        for(int j=0; j<m; j++){
            if(i>0 && j>0 && maze[i][j]==-1){
                temp[j]=0;
                continue;
            }
            if(i==0 && j==0){
                temp[j]=1;
                continue;
            }

            int up=0;
            int left =0;

            if(i>0)
                up = prev[j];
            if(j>0)
                left = temp[j-1];

            temp[j] = up + left;
        }
        prev = temp;
    }

    return prev[n-1];
}

int main() {

    vector<vector<int> > maze{{{0,0,0},
                                {0,-1,0},
                                {0,0,0}}};

    int n = maze.size();
    int m = maze[0].size();

    cout<<mazeObstacles(n,m,maze);
}

```

### **Output:**

2

**Time Complexity: O(M\*N)**

Reason: There are two nested loops

### **Space Complexity: O(N)**

Reason: We are using an external array of size 'N' to store only one row.

# Minimum Path Sum In a Grid (DP 10)

 [takeuforward.org/data-structure/minimum-path-sum-in-a-grid-dp-10](https://takeuforward.org/data-structure/minimum-path-sum-in-a-grid-dp-10)

January 25, 2022

In this article, we will solve the most asked coding interview problem: Minimum Path Sum In a Grid.

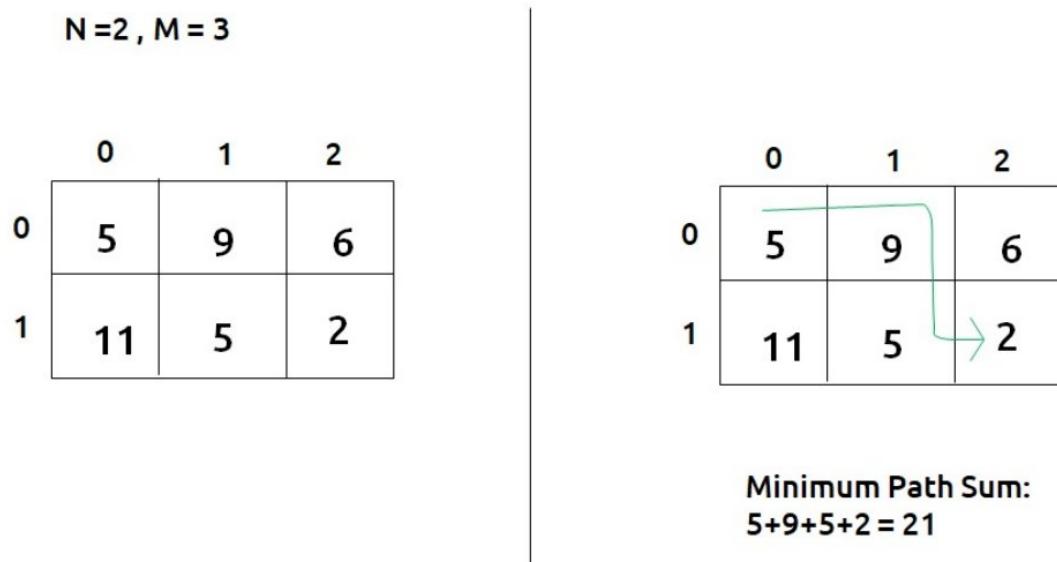
## Problem Link: Minimum Path Sum in A Grid

### Problem Description:

We are given an “N\*M” matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost path that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

**Example:**



**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** Grid Unique Path 2

### Solution :

This question is a slight modification of the question discussed in Grid Unique Path 2 . In the previous problem, there were obstacles whereas this problem has cost associated with a cell and we need to return the minimum cost path.

## Why a Greedy Solution doesn't work?

As we have to return the minimum path sum, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the cheaper option.

This approach will not give us the correct answer. Let us look at this example to understand:

At every cell, we have two choices: to move right and move down. Our ultimate aim is to provide a path that provides us the least path sum. Therefore at every cell, we will make the choice to move which costs are less.

$$N=3, M=3$$

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

$$N=3, M=3$$

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

Greedy Solution

$$10+8+2+100+2 = 112$$

$$N=3, M=3$$

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

Non-Greedy Solution

$$10+10+1+1+2 = 24$$

- Figure on the left gives us a greedy solution, where we move by taking the local best choice.
- Figure on the right gives us a non-greedy solution.

We can clearly see the problem with the greedy solution. Whenever we are making a local choice, we may tend to choose a path that may cost us way more later.

Therefore, the other alternative left to us is to generate all the possible paths and see which is the path with the minimum path sum. To generate all paths we will use **recursion**.

### **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two variables N and M, representing the dimensions of the matrix.

We can define the function with two parameters i and j, where i and j represent the row and column of the matrix.

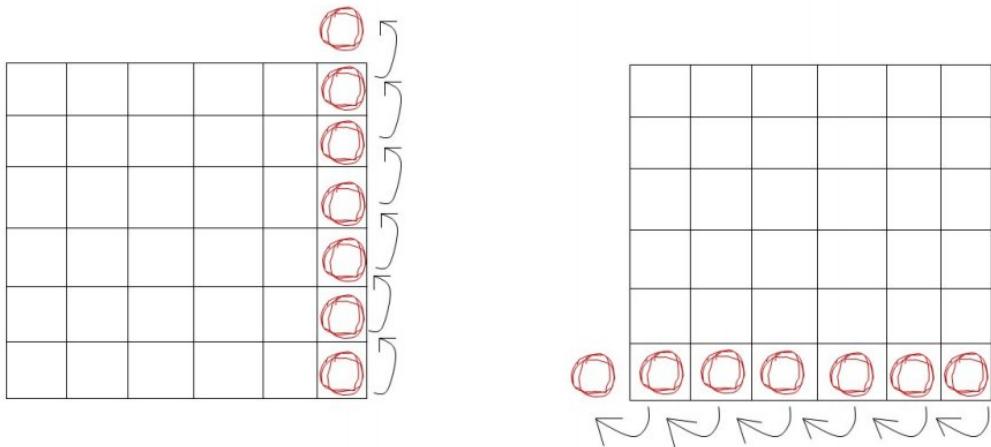
**f(i,j) -> Minimum path sum from matrix[0,0] to matrix[i][j].**

We will be doing a top-down recursion, i.e we will move from the cell[M-1][N-1] and try to find our way to the cell[0][0]. Therefore at every index, we will try to move up and towards the left.

### **Base Case:**

There will be three base cases:

- When i=0 and j=0, that is we have reached the destination so we can add to path the current cell value, hence we return mat[0][0].
- When i<0 or j<0, it means that we have crossed the boundary of the matrix and we don't want to find a path from here, so we return a very large number( say, 1e9) so that this path is rejected by the calling function.



As the function call crosses the boundary of the matrix, we return a large value(say  $10^9$ ) so that this path is not considered by the last calling function (as we will be returning the minimum path)

The pseudocode till this step will be:

```
f(i,j) {
    if( i==0 && j==0)  return mat[0][0]
    if( i<0 || j<0) return 1e9
}
```

**Step 2:** Try out all possible choices at a given index.

As we are writing a top-down recursion, at every index we have two choices, one to go up( $\uparrow$ ) and the other to go left( $\leftarrow$ ). To go upwards, we will reduce  $i$  by 1, and move towards left we will reduce  $j$  by 1.

Now when we get our answer for the recursive call ( $f(i-1,j)$  or  $f(i,j-1)$ ), we need to also add the current cell value to it as we have to include it too for the current path sum.

```

f(i,j) {
    if( i==0 && j==0) return mat[0][0]
    if( i<0 || j<0) return 1e9
    up = mat[i][j] + f(i-1,j)
    left = mat[i][j] + f(i,j-1)
}

```

### Step 3: Take the maximum of all choices

As we have to find the **minimum path sum** of all the possible unique paths, we will return the **minimum** of the choices(up and left)

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {
    if( i==0 && j==0) return mat[0][0]
    if( i<0 || j<0) return 1e9
    up = mat[i][j] + f(i-1,j)
    left = mat[i][j] + f(i,j-1)
    return min(up,left)
}

```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][m]
2. Whenever we want to find the answer of a particular row and column (say f(i,j)), we first check whether the answer is already calculated using the dp array(i.e dp[i][j]!= -1 ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[i][j] to the solution we get.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int minSumPathUtil(int i, int j, vector<vector<int>> &matrix, vector<vector<int>>
&dp)
{
    if(i==0 && j == 0)
        return matrix[0][0];
    if(i<0 || j<0)
        return 1e9;
    if(dp[i][j]!=-1) return dp[i][j];

    int up = matrix[i][j]+minSumPathUtil(i-1,j,matrix,dp);
    int left = matrix[i][j]+minSumPathUtil(i,j-1,matrix,dp);

    return dp[i][j] = min(up,left);
}

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<vector<int> > dp(n, vector<int>(m, -1));
    return minSumPathUtil(n-1,m-1,matrix,dp);
}

int main() {

    vector<vector<int> > matrix{{5,9,6},
                                {11,5,2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}
```

### Output: 21

**Time Complexity: O(N\*M)**

Reason: At max, there will be  $N*M$  calls of recursion.

### Space Complexity: $O((M-1)+(N-1)) + O(N*M)$

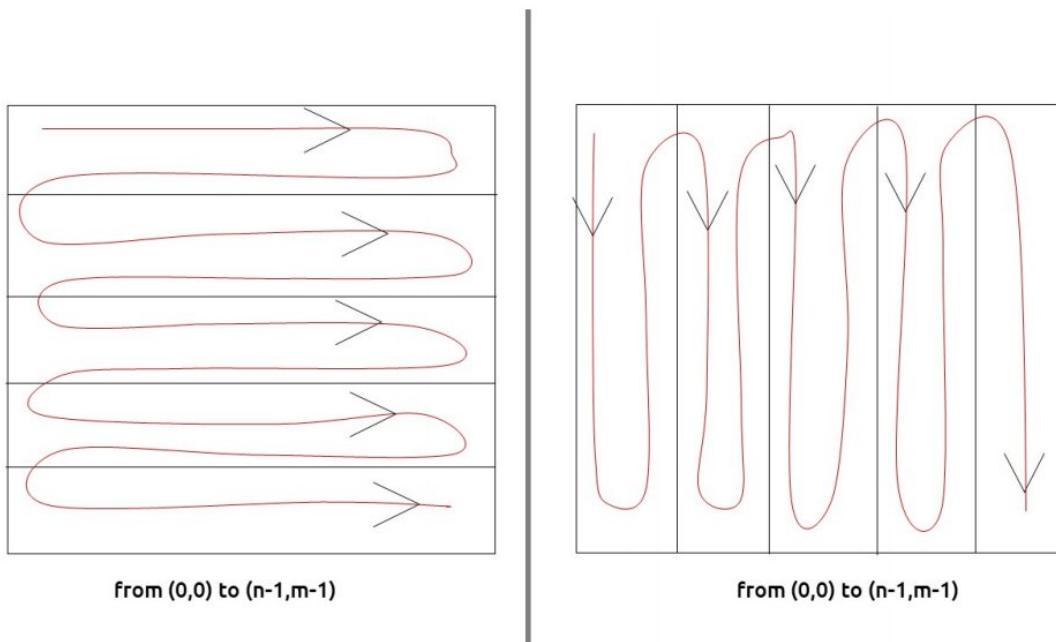
Reason: We are using a recursion stack space:  $O((M-1)+(N-1))$ , here  $(M-1)+(N-1)$  is the path length and an external DP Array of size ' $N*M$ '.

### Steps to convert Recursive Solution to Tabulation one.

Tabulation is the bottom-up approach, which means we will go from the base case to the main problem.

The steps to convert to the tabular solution are given below:

- Declare a  $dp[]$  array of size  $[n][m]$ .
- First initialize the base condition values, i.e  $dp[0][0] = matrix[0][0]$
- Our answer should get stored in  $dp[n-1][m-1]$ . We want to move from  $(0,0)$  to  $(n-1,m-1)$ . But we can't move arbitrarily, we should move such that at a particular  $i$  and  $j$ , we have all the values required to compute  $dp[i][j]$ .
- If we see the memoized code, values required for  $dp[i][j]$  are:  $dp[i-1][j]$  and  $dp[i][j-1]$ . So we only use the previous row and column value.
- We have already filled the top-left corner ( $i=0$  and  $j=0$ ), if we move in any of the two following ways(given below), at every cell we do have all the previous values required to compute its value.



- We can use two nested loops to have this traversal
- Whenever  $i > 0, j > 0$ , we will simply mark  $dp[i][j] = matrix[i][j] + \min(dp[i-1][j], dp[i][j-1])$ , according to our recursive relation.
- When  $i=0$  or  $j=0$ , we add to up( or left)  $1e9$ , so that this path can be rejected.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<vector<int> > dp(n, vector<int>(m, 0));
    for(int i=0; i<n ; i++){
        for(int j=0; j<m; j++){
            if(i==0 && j==0) dp[i][j] = matrix[i][j];
            else{
                int up = matrix[i][j];
                if(i>0) up += dp[i-1][j];
                else up += 1e9;

                int left = matrix[i][j];
                if(j>0) left+=dp[i][j-1];
                else left += 1e9;

                dp[i][j] = min(up, left);
            }
        }
    }

    return dp[n-1][m-1];
}

int main() {

    vector<vector<int> > matrix{{5, 9, 6},
                                {11, 5, 2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}
```

## **Output:**

21

**Time Complexity: O(N\*M)**

Reason: There are two nested loops

**Space Complexity: O(N\*M)**

Reason: We are using an external array of size ‘N\*M’.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = \text{matrix}[i][j] + \min(dp[i-1][j] + dp[i][j-1])$$

We see that we only need the previous row and column, in order to calculate  $dp[i][j]$ . Therefore we can space optimize it.

Initially, we can take a dummy row ( say prev) and initialize it as 0.

Now the current row(say temp) **only needs the** previous row value and the current row's value in order to calculate  $dp[i][j]$ .

		prev[n]	0	0	0	0
		i \ j	0	1	...	m-1
temp[n]	0	1	1			
	1					
	.					
	.					
	.					
	n-1					

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

This row is initialized to 0

This row's cells only need row prev's values and current row's previous values, and first cell is initialized to 1

At the next step, the temp array becomes the prev of the next step and using its values we can still calculate the next row's values.

	$i \searrow j$	0	1	...	$m-1$
$prev[n]$	0	1	1	1	1
$temp[n]$	1	1	2		
.					
.					
.					
$n-1$					

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

This row's cells only need row prev's values and current row's previous values

At last  $prev[n-1]$  will give us the required answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<int> prev(m, 0);
    for(int i=0; i<n ; i++){
        vector<int> temp(m, 0);
        for(int j=0; j<m; j++){
            if(i==0 && j==0) temp[j] = matrix[i][j];
            else{
                int up = matrix[i][j];
                if(i>0) up += prev[j];
                else up += 1e9;

                int left = matrix[i][j];
                if(j>0) left+=temp[j-1];
                else left += 1e9;

                temp[j] = min(up, left);
            }
        }
        prev=temp;
    }

    return prev[m-1];
}

int main() {

    vector<vector<int> > matrix{{5,9,6},
                                {11,5,2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}

```

## Output:

21

### Time Complexity: O(M\*N)

Reason: There are two nested loops

### Space Complexity: O(N)

Reason: We are using an external array of size 'N' to store only one row.



# Minimum path sum in Triangular Grid (DP 11)

 [takeuforward.org/data-structure/minimum-path-sum-in-triangular-grid-dp-11](https://takeuforward.org/data-structure/minimum-path-sum-in-triangular-grid-dp-11)

January 28, 2022

In this article, we will solve the most asked coding interview problem: Minimum path sum in Triangular Grid

## Problem Link: Fixed Starting and Ending Point

### Problem Description:

We are given a Triangular matrix. We need to find the minimum path sum from the first row to the last row.

At every cell we can move in only two directions: either to the bottom cell ( $\downarrow$ ) or to the bottom-right cell()

**Example:**

1			
2	3		
3	6	7	
8	9	6	10

1			
2	3		
3	6	7	
8	9	6	10

Minimum Path Sum = 14  
(1 + 2 + 3 + 8)

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** Minimum Path Sum in a Grid

### Solution :

This question is a slight modification of the question discussed in Minimum Path Sum in a Grid. In the previous problem, we were given a rectangular matrix whereas here the matrix is in the form of a triangle. Moreover, we don't have a fixed destination, we need to

return the minimum sum path from the top cell to any cell of the bottom row.

## Why a Greedy Solution doesn't work?

As we have to return the minimum path sum, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the cheaper option.

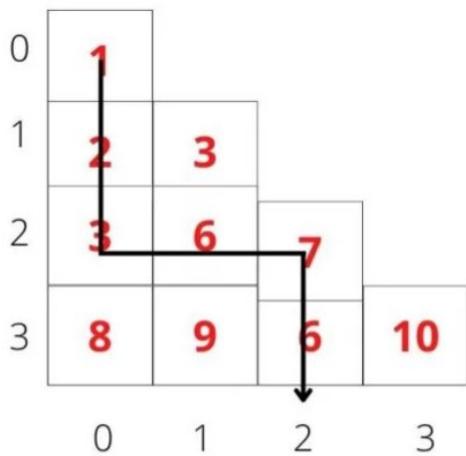
This approach will not give us the correct answer. Let us look at this example to understand:

At every cell, we have two choices: to move to the bottom cell or move to the bottom-right cell. Our ultimate aim is to provide a path that provides us the least path sum. Therefore at every cell, we will make the choice to move which costs us less.

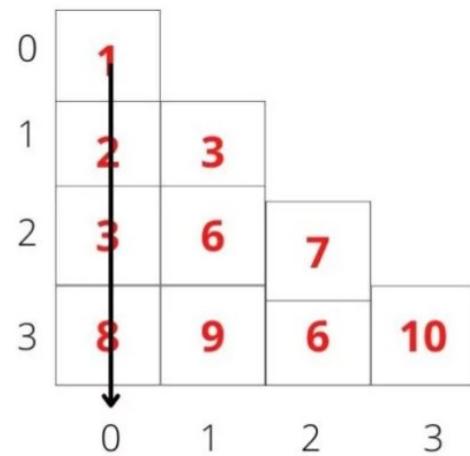
**N=4**

0	1			
1	2	3		
2	3	6	7	
3	8	9	6	10

0 1 2 3



Greedy Solution:  $1+2+3+6+7+6 = 25$



Non Greedy Solution:  $1+2+3+8 = 14$

- Figure on the left gives us a greedy solution, where we move by taking the local best choice.
- Figure on the right gives us a non-greedy solution.

We can clearly see the problem with the greedy solution. Whenever we are making a local choice, we may tend to choose a path that may cost us way more later.

Therefore, the other alternative left to us is to generate all the possible paths and see which is the path with the minimum path sum. To generate all paths we will use **recursion**.

### **Steps to form the recursive solution:**

---

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given a triangular matrix with the number of rows equal to N.

We can define the function with two parameters i and j, where i and j represent the row and column of the matrix.

Now our ultimate aim is to reach the last row. We can define  $f(i,j)$  such that it gives us the minimum path sum from the cell  $[i][j]$  to the last row.

**$f(i,j) \rightarrow \text{Minimum path sum from matrix}[0,0] \text{ to the last row of matrix.}$**

We want to find  $f(0,0)$  and return it as our answer.

#### **Base Case:**

There will be a single base case:

When  $i == N-1$ , that is when we have reached the last row, so the min path from that cell to the last row will be the value of that cell itself, hence we return  $\text{mat}[i][j]$ .

At every cell, we have two options to move to the bottom cell( $\downarrow$ ) or to the bottom-right cell( $\rightarrow$ ). If we closely observe the triangular grid, at max we can reach the last row from where we return so we will not be able to go move out of the index of the grid. Therefore only one base condition is required.



At max, we will reach the last row from which we will return therefore only one base case is needed

The pseudocode till this step will be:

**Step 2:** Try out all possible choices at a given index.

At every index we have two choices, one to go to the bottom cell( $\downarrow$ ) other to the bottom-right cell(). To go to the bottom, we will increase  $i$  by 1, and to move towards the bottom-right we will increase both  $i$  and  $j$  by 1.

Now when we get our answer for the recursive call ( $f(i+1,j)$  or  $f(i+1,j+1)$ ), we need to also add the current cell value to it as we have to include it too for the current path sum.

```
f(i,j) {
    if( i==N-1) return mat[i][j]
}
```

**Step 3: Take the maximum of all choices**

As we have to find the **minimum path sum** of all the possible unique paths, we will return the **minimum** of the choices(down and diagonal)

The final pseudocode after steps 1, 2, and 3:

```
f(i,j) {  
  
    if( i==N-1) return mat[i][j]  
  
    down = mat[i][j] + f(i+1,j)  
  
    diagonal = mat[i][j] + f(i+1,j+1)
```

```
}
```

```
f(i,j) {  
  
    if( i==N-1) return mat[i][j]  
  
    down = mat[i][j] + f(i+1,j)  
  
    diagonal = mat[i][j] + f(i+1,j+1)  
  
    return min(down,diagonal)
```

```
}
```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [N][N]
2. Whenever we want to find the answer of a particular row and column (say  $f(i,j)$ ), we first check whether the answer is already calculated using the dp array (i.e  $dp[i][j] \neq -1$ ). If yes, simply return the value from the dp array.

3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j]$  to the solution we get.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int minimumPathSumUtil(int i, int j, vector<vector<int> > &triangle, int n,
vector<vector<int> > &dp) {

    if(dp[i][j]!=-1)
        return dp[i][j];

    if(i==n-1) return triangle[i][j];

    int down = triangle[i][j]+minimumPathSumUtil(i+1,j,triangle,n,dp);
    int diagonal = triangle[i][j]+minimumPathSumUtil(i+1,j+1,triangle,n,dp);

    return dp[i][j] = min(down, diagonal);
}

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<vector<int> > dp(n, vector<int>(n,-1));
    return minimumPathSumUtil(0,0,triangle,n,dp);
}

int main() {

    vector<vector<int> > triangle{{1},
                                    {2, 3},
                                    {3, 6, 7},
                                    {8, 9, 6, 10}};

    int n = triangle.size();

    cout<<minimumPathSum(triangle,n);
}
```

### Output: 14

### Time Complexity: O(N\*N)

Reason: At max, there will be (half of, due to triangle)  $N*N$  calls of recursion.

### Space Complexity: O(N) + O(N\*N)

Reason: We are using a recursion stack space:  $O(N)$ , where  $N$  is the path length and an external DP Array of size ' $N \times N$ '.

## Steps to convert Recursive Solution to Tabulation one.

---

Recursion/Memoization is a top-down approach whereas tabulation is a bottom-up approach. As in recursion/memoization, we have moved from  $0$  to  $N-1$ , in tabulation we move from  $N-1$  to  $0$ , i.e last row to the first one.

The steps to convert to the tabular solution are given below:

- Declare a  $dp[]$  array of size  $[N][N]$ .
- First initialize the base condition values, i.e the last row of  $dp$  matrix to the last row of the triangle matrix.
- Our answer should get stored in  $dp[0][0]$ . We want to move from the last row to the first row. So that whenever we compute values for a cell, we have all the values required to calculate it.
- If we see the memoized code, values required for  $dp[i][j]$  are:  $dp[i+1][j]$  and  $dp[i+1][j+1]$ . So we only need the values from the ' $i+1$ ' row.
- We have already filled the last row ( $i=N-1$ ), if we start from row ' $N-2$ ' and move upwards we will find the values correctly.
- We can use two nested loops to have this traversal.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<vector<int> > dp(n, vector<int>(n, 0));

    for(int j=0; j<n; j++){
        dp[n-1][j] = triangle[n-1][j];
    }

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = triangle[i][j]+dp[i+1][j];
            int diagonal = triangle[i][j]+dp[i+1][j+1];

            dp[i][j] = min(down, diagonal);
        }
    }

    return dp[0][0];
}

int main() {

    vector<vector<int> > triangle{{1},
                                    {2, 3},
                                    {3, 6, 7},
                                    {8, 9, 6, 10}};

    int n = triangle.size();

    cout<<minimumPathSum(triangle, n);
}

```

### **Output:**

21

### **Time Complexity: O(N\*N)**

Reason: There are two nested loops

### **Space Complexity: O(N\*N)**

Reason: We are using an external array of size 'N\*N'. The stack space will be eliminated.

---

## **Part 3: Space Optimization**

If we closely look the relation,

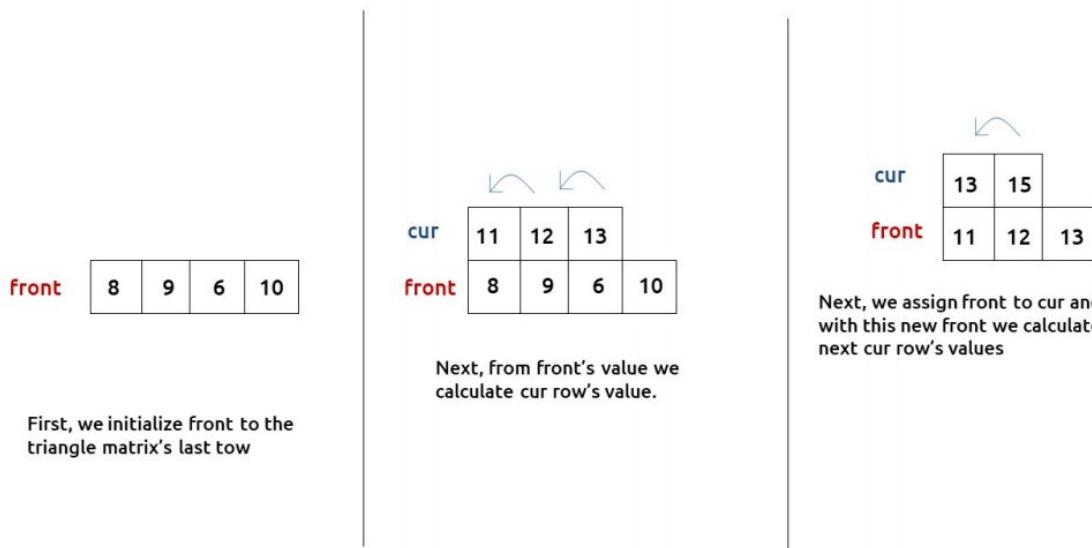
$$dp[i][j] = matrix[i][j] + \min(dp[i+1][j] + dp[i+1][j+1])$$

We see that we only need the next row, in order to calculate  $dp[i][j]$ . Therefore we can space optimize it.

Initially we can take a dummy row ( say front). We initialize this row to the triangle matrix last row( as done in tabulation).

Now the current row(say cur) only needs the front **row's** value in order to calculate  $dp[i][j]$ .

### Space Optimization



At last, front [0] will give us the required answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<int> front(n, 0), cur(n, 0);

    for(int j=0;j<n;j++){
        front[j] = triangle[n-1][j];
    }

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = triangle[i][j]+front[j];
            int diagonal = triangle[i][j]+front[j+1];

            cur[j] = min(down, diagonal);
        }
        front=cur;
    }

    return front[0];
}

int main() {

    vector<vector<int> > triangle{{1},
                                    {2, 3},
                                    {3, 6, 7},
                                    {8, 9, 6, 10}};

    int n = triangle.size();

    cout<<minimumPathSum(triangle,n);
}

```

### **Output:**

14

### **Time Complexity: O(N\*N)**

Reason: There are two nested loops

### **Space Complexity: O(N)**

Reason: We are using an external array of size 'N' to store only one row.

# Minimum/Maximum Falling Path Sum (DP-12)

 [takeuforward.org/data-structure/minimum-maximum-falling-path-sum-dp-12](https://takeuforward.org/data-structure/minimum-maximum-falling-path-sum-dp-12)

January 28, 2022

In this article, we will solve the most asked coding interview problem:  
Minimum/Maximum falling path sum.

## Problem Link: Variable Starting and Ending Point

### Problem Description:

We are given an 'N\*M' matrix. We need to find the maximum path sum from any cell of the first row to any cell of the last row.

At every cell we can move in three directions: to the bottom cell ( $\downarrow$ ), to the bottom-right cell(), or to the bottom-left cell().

### Example:

N=4, M=4

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

N=4, M=4

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

Maximum Path Sum: 105

(2+100+1+2)

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

### Pre-req: Minimum Path Sum in a Triangular Grid

### Solution :

This question is a slight modification of the question discussed in **Minimum Path Sum in a Triangular Grid**. In the previous problem, we were given a fixed starting and a variable ending point, whereas here the problem states that the starting point can be any

cell from the first row and the ending point can be any cell in the last row.

### Why a Greedy Solution doesn't work?

As we have to return the minimum path sum, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the cheaper option. But there is no '**uniformity**' in the values of the string, therefore it can happen that whenever we are making a local choice that gives us a better path, we actually take a path which in the later stages is giving us the lesser path sum.

As a greedy solution doesn't work, our next choice will be to try out all the possible paths. To generate all possible paths we will use **recursion**.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

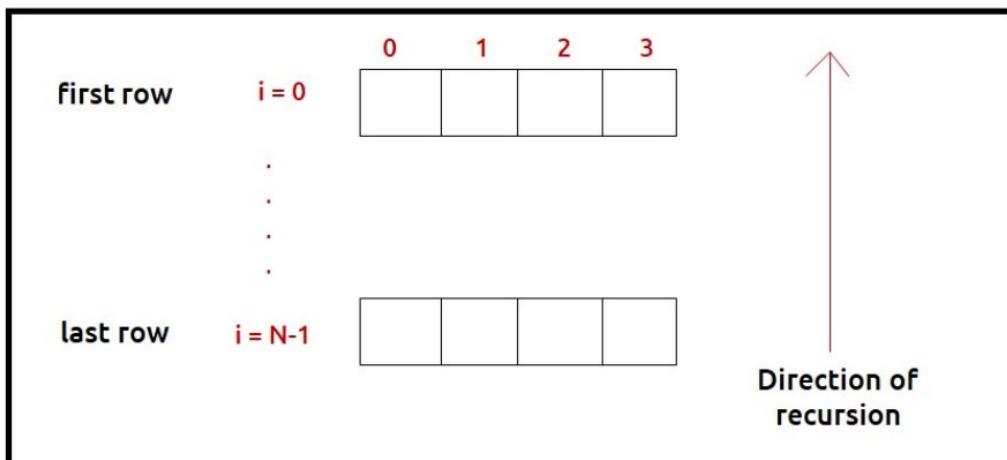
**Step 1:** Express the problem in terms of indexes.

We are given an 'N\*M' matrix. We can define the function with two parameters i and j, where i and j represent the row and column of the matrix.

Now our ultimate aim is to reach the last row. We can define  $f(i,j)$  such that it gives us the maximum path sum from any cell in the first row to the cell[i][j].

**$f(i,j) \rightarrow$  Maximum path sum from the first row to the cell[i][j].**

If we see the figure given below:



We have a top row and a bottom row, we will be writing a recursion in the direction of the last row to the first row. For the last row,  $i=N-1$  therefore we need to find four different answers:

$f(N-1,0), f(N-1,1), f(N-1,2), f(N-1,3)$

These recursive calls will give the maximum path sum from a cell in the first row to the respective four cells for which the recursion calls are made. We need to return the **maximum** value among these as the final answer.

### Base Case:

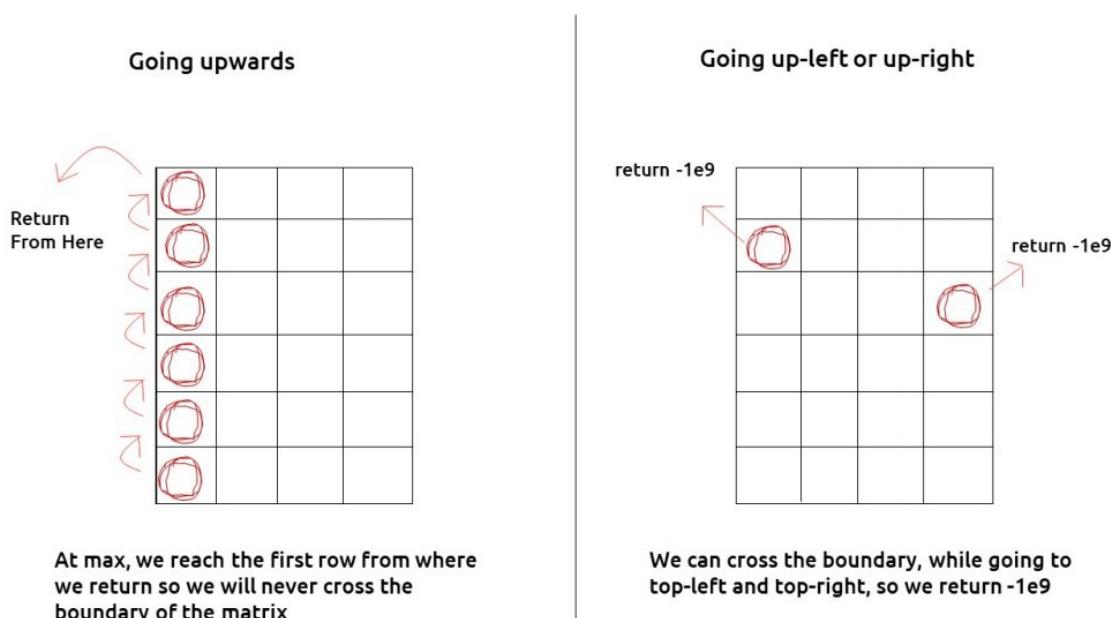
There will be the following base cases:

When  $i == 0$ , it means we are at the first row, so the min path from that cell to the first will be the value of that cell itself, hence we return  $\text{mat}[0][j]$ .

At every cell we have three options (we are writing recursion from the last row to the first row): to the top cell ( $\uparrow$ ), to the top-right cell(), or to the top-left cell().

As we are moving to the top cell ( $\uparrow$ ), at max we will reach the first row, from where we return, so we will never go out of the bounding index.

To move to the top-left cell() or to the top-right cell(), it can happen that we may go out of bound as shown in the figure(below). So we need to handle it, we can return  $-1e9$ , whenever we go out of bound, in this way this path will not be selected by the calling function as we have to return the maximum path.



If  $j < 0$  or  $j >= M$ , then we return  $-1e9$

The pseudocode till this step will be:

**Step 2:** Try out all possible choices at a given index.

At every cell we have three options (we are writing recursion from the last row to the first row): to the top cell ( $\uparrow$ ), to the top-right cell(), or to the top-left cell().

To go to the top, we will decrease i by 1, and to move towards top-left, we will decrease both i and j by 1 whereas to move to top-right, we will decrease i by 1 and increase j by 1.

```
f(i,j) {  
    if(j<0 || j>=M)  
        return -1e9  
    if(i==0) return mat[0][j]  
}  
}
```

Now when we get our answer for the recursive call ( $f(i-1,j)$ ,  $f(i-1,j-1)$  or  $f(i-1,j+1)$ ), we need to also add the current cell value to it as we have to include it too for the current path sum.

```
f(i,j) {  
    if(j<0 || j>=M)  
        return -1e9  
    if(i==0) return mat[0][j]  
  
    up = mat[i-1][j] + f(i-1,j)  
    leftDiagonal = mat[i-1][j] + f(i-1,j-1)  
    rightDiagonal = mat[i-1][j] + f(i-1,j+1)  
}  
}
```

**Step 3: Take the maximum of all choices**

As we have to find the **maximum path sum** of all the possible unique paths, we will return the **maximum** of all the choices(up, leftDiagonal, right diagonal)

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {

    if(j<0 || j>=M)
        return -1e9

    if(i==0) return mat[0][j]

    up = mat[i-1][j] + f(i-1,j)

    leftDiagonal = mat[i-1][j] + f(i-1,j-1)

    rightDiagonal = mat[i-1][j] + f(i-1,j+1)

    return max(up, leftDiagonal,
               rightDiagonal)
}

```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [N][M]
2. Whenever we want to find the answer of a particular row and column (say  $f(i,j)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[i][j] \neq -1$ ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j]$  to the solution we get.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int getMaxUtil(int i, int j, int m, vector<vector<int>> &matrix,
vector<vector<int> > &dp){

    // Base Conditions
    if(j<0 || j>=m)
        return -1e9;
    if(i==0)
        return matrix[0][j];

    if(dp[i][j]!=-1) return dp[i][j];

    int up = matrix[i][j] + getMaxUtil(i-1,j,m,matrix,dp);
    int leftDiagonal = matrix[i][j] + getMaxUtil(i-1,j-1,m,matrix,dp);
    int rightDiagonal = matrix[i][j] + getMaxUtil(i-1,j+1,m,matrix,dp);

    return dp[i][j]= max(up,max(leftDiagonal,rightDiagonal));
}

int getMaxPathSum(vector<vector<int> > &matrix){

    int n = matrix.size();
    int m = matrix[0].size();

    vector<vector<int>> dp(n,vector<int>(m,-1));

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        int ans = getMaxUtil(n-1,j,m,matrix,dp);
        maxi = max(maxi,ans);
    }

    return maxi;
}

int main() {

    vector<vector<int> > matrix{{1,2,10,4},
                                {100,3,2,1},
                                {1,1,20,2},
                                {1,2,2,1}};

    cout<<getMaxPathSum(matrix);
}

```

### **Output:**

105

**Time Complexity: O(N\*N)**

Reason: At max, there will be  $M \times N$  calls of recursion to solve a new problem,

### **Space Complexity: $O(N) + O(N \times M)$**

Reason: We are using a recursion stack space:  $O(N)$ , where  $N$  is the path length and an external DP Array of size ' $N \times M$ '.

### **Steps to convert Recursive Solution to Tabulation one.**

The steps to convert to the tabular solution are given below:

- Declare a  $dp[]$  array of size  $[N][M]$ .
- First initialize the base condition values, i.e the first row of the dp array to the first row of the input matrix.
- We want to move from the first row to the last row. Whenever we compute values for a cell, we want to have all the values required to calculate it.
- If we see the memoized code, values required for  $dp[i][j]$  are:  $dp[i-1][j]$ ,  $dp[i-1][j-1]$  and  $dp[i-1][j+1]$ . So we only need the values from the ' $i-1$ ' row.
- We have already filled the first row ( $i=0$ ), if we start from row ' $1$ ' and move downwards we will find the values correctly.
- We can use two nested loops to have this traversal.
- At last we need to return the maximum among the last row of dp array as our answer.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int getMaxPathSum(vector<vector<int> > &matrix){

    int n = matrix.size();
    int m = matrix[0].size();

    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Initializing first row - base condition
    for(int j=0; j<m; j++){
        dp[0][j] = matrix[0][j];
    }

    for(int i=1; i<n; i++){
        for(int j=0;j<m;j++){

            int up = matrix[i][j] + dp[i-1][j];

            int leftDiagonal= matrix[i][j];
            if(j-1>=0) leftDiagonal += dp[i-1][j-1];
            else leftDiagonal += -1e9;

            int rightDiagonal = matrix[i][j];
            if(j+1<m) rightDiagonal += dp[i-1][j+1];
            else rightDiagonal += -1e9;

            dp[i][j] = max(up, max(leftDiagonal,rightDiagonal));
        }
    }

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        maxi = max(maxi,dp[n-1][j]);
    }

    return maxi;
}

int main() {

    vector<vector<int> > matrix{{1,2,10,4},
                                {100,3,2,1},
                                {1,1,20,2},
                                {1,2,2,1}};

    cout<<getMaxPathSum(matrix);
}

```

### **Output:**

105

## Time Complexity: O(N\*M)

Reason: There are two nested loops

## Space Complexity: O(N\*M)

Reason: We are using an external array of size 'N\*M'. The stack space will be eliminated.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = \text{matrix}[i][j] + \max(dp[i-1][j], dp[i-1][j-1], dp[i-1][j+1])$$

We see that we only need the previous row, in order to calculate  $dp[i][j]$ . Therefore we can space optimize it.

Initially we can take a dummy row ( say prev). We initialize this row to the input matrix first row( as done in tabulation).

Now the current row(say cur) only needs the **prev row's** value inorder to calculate  $dp[i][j]$ .

### Space Optimization

prev	1	2	10	4
------	---	---	----	---

First, we initialize prev to the input matrix's first row

prev	1	2	10	4
cur	102	13	12	11

Next, from prev's value we calculate cur row's value.

prev	102	13	12	11
cur	103	103	33	14

Next, we assign prev to cur and with this new prev we calculate next cur row's values

At last, we will return the maximum value among the values of the prev row as our answer.

### Code:

C++ Code

```

#include <bits/stdc++.h>

using namespace std;

int getMaxPathSum(vector<vector<int> > &matrix){

    int n = matrix.size();
    int m = matrix[0].size();

    vector<int> prev(m,0), cur(m,0);

    // Initializing first row - base condition
    for(int j=0; j<m; j++){
        prev[j] = matrix[0][j];
    }

    for(int i=1; i<n; i++){
        for(int j=0;j<m;j++){

            int up = matrix[i][j] + prev[j];

            int leftDiagonal= matrix[i][j];
            if(j-1>=0) leftDiagonal += prev[j-1];
            else leftDiagonal += -1e9;

            int rightDiagonal = matrix[i][j];
            if(j+1<m) rightDiagonal += prev[j+1];
            else rightDiagonal += -1e9;

            cur[j] = max(up, max(leftDiagonal,rightDiagonal));
        }

        prev = cur;
    }

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        maxi = max(maxi,prev[j]);
    }

    return maxi;
}

int main() {

    vector<vector<int> > matrix{{1,2,10,4},
                                {100,3,2,1},
                                {1,1,20,2},
                                {1,2,2,1}};

    cout<<getMaxPathSum(matrix);
}

```

**Output:**

105

**Time Complexity: O(N\*M)**

Reason: There are two nested loops

**Space Complexity: O(M)**

Reason: We are using an external array of size 'M' to store only one row.

# 3-d DP : Ninja and his friends (DP-13)

 [takeuforward.org/data-structure/3-d-dp-ninja-and-his-friends-dp-13](https://takeuforward.org/data-structure/3-d-dp-ninja-and-his-friends-dp-13)

February 2, 2022

In this article, we will solve the most asked coding interview problem: Ninja and his friends.

## Problem Link: [Ninja and his friends](#)

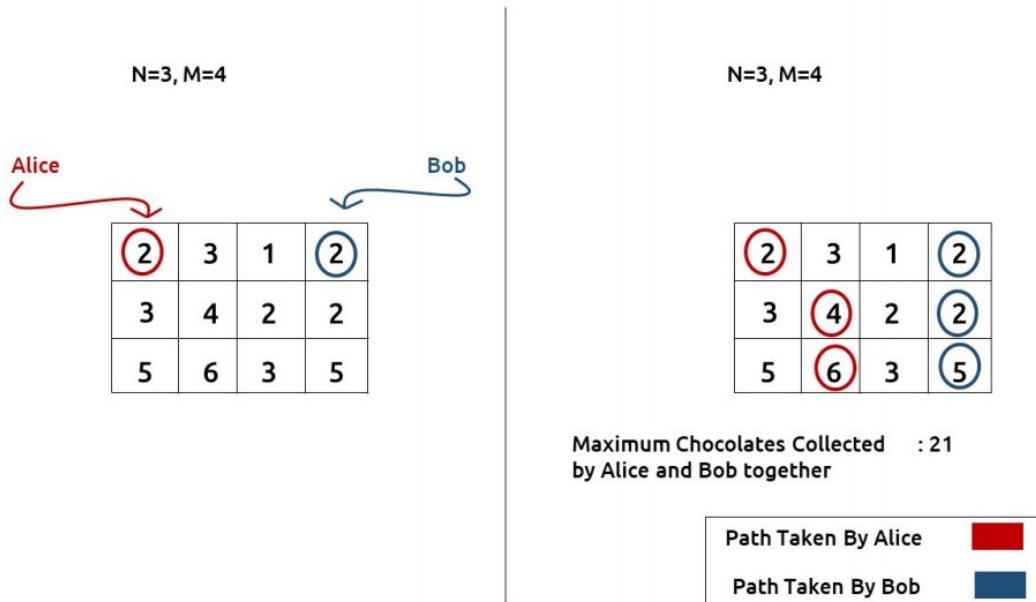
### Problem Description:

We are given an ' $N \times M$ ' matrix. Every cell of the matrix has some chocolates on it,  $\text{mat}[i][j]$  gives us the number of chocolates. We have two friends 'Alice' and 'Bob'. initially, Alice is standing on the cell(0,0) and Bob is standing on the cell(0,  $M-1$ ). Both of them can move only to the cells below them in these three directions: to the bottom cell ( $\downarrow$ ), to the bottom-right cell(), or to the bottom-left cell().

When Alice and Bob visit a cell, they take all the chocolates from that cell with them. It can happen that they visit the same cell, in that case, the chocolates need to be considered only once.

They cannot go out of the boundary of the given matrix, we need to return the maximum number of chocolates that Bob and Alice can **together** collect.

### Example:



**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** 2D DP

### Solution :

In this question, there are two fixed starting and variable ending points, but as per the movement of Alice and Bob, we know that they will end in the last row. They have to move together at a time to the next row.

#### Why a Greedy Solution doesn't work?

As we have to return the **maximum** chocolates collected, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the option that gives us more chocolates. But there is no '**uniformity**' in the values of the matrix, therefore it can happen that whenever we are making a local choice that gives us a better path, we actually take a path that in the later stages is giving us fewer chocolates.

As a greedy solution doesn't work, our next choice will be to try out all the possible paths. To generate all possible paths we will use **recursion**.

#### Steps to form the recursive solution:

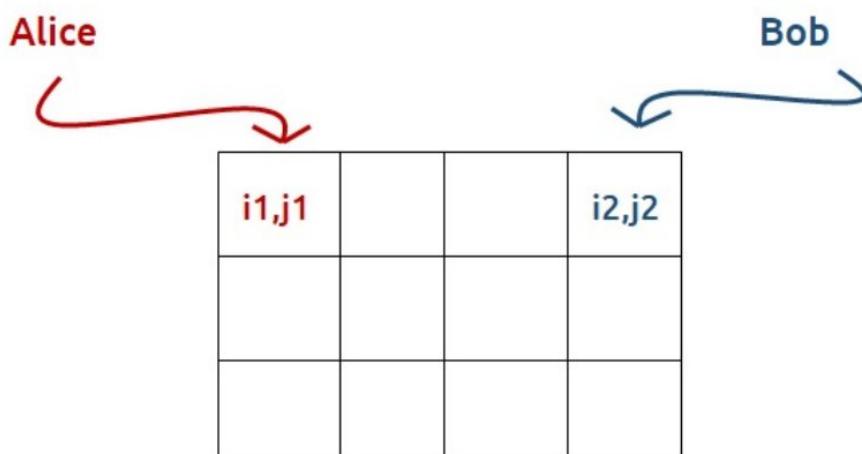
We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

This question is slightly different from all the previous questions, here we are given two starting points from where Alice and Bob can move.

We are given an 'N\*M' matrix. We need to define the function with four parameters  $i_1, j_1, i_2, j_2$  to describe the positions of Alice and Bob at a time.

$$N=3, M=4$$



If we observe, initially Alice and Bob are at the first row, and they always move to the row below them every time, so they will always be in the same row. Therefore two different variables  $i_1$  and  $i_2$ , to describe their positions are redundant. We can just use single parameter  $i$ , which tells us in which row of the grid both of them are.

Therefore, we can modify the function. It now takes three parameters:  $i, j_1$ , and  $j_2$ .  $f(i, j_1, j_2)$  will give us the maximum number of chocolates collected by Alice and Bob from their current positions to the last position.

**$f(i, j_1, j_2) \rightarrow$  Maximum chocolates collected by Alice from cell[i][j1] and Bob from cell[i][j2] till the last row.**

### **Base Case:**

There will be the following base cases:

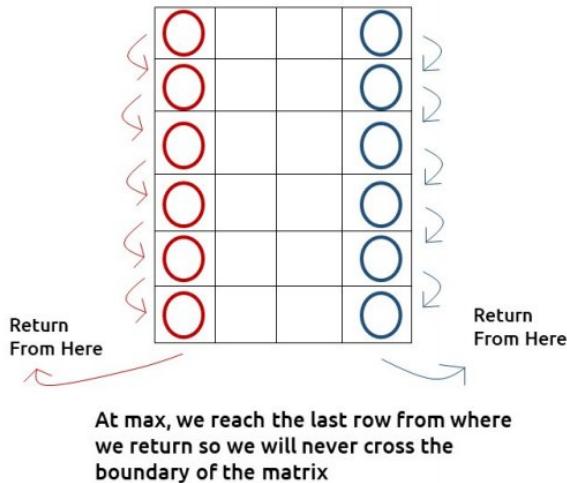
When  $i == N-1$ , it means we are at the last row, so we need to return from here. Now it can happen that at the last row, both Alice and Bob are at the same cell, in this condition we will return only chocolates collected by Alice,  $\text{mat}[i][j_1]$  (as question states that the chocolates cannot be doubly calculated), otherwise we return sum of chocolates collected by both,  $\text{mat}[i][j_1] + \text{mat}[i][j_1][j_2]$ .

At every cell, we have three options to go: to the bottom cell ( $\downarrow$ ), to the bottom-right cell() or to the bottom-left cell()

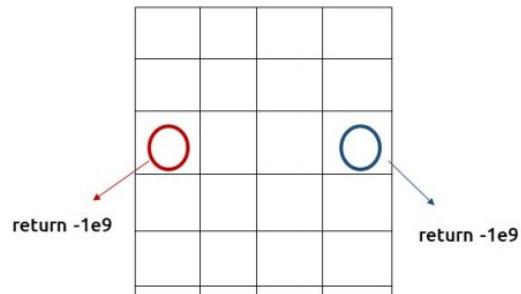
As we are moving to the bottom cell ( $\downarrow$ ), at max we will reach the last row, from where we return, so we will never go out of the bounding index.

To move to the bottom-right cell() or to the bottom-left cell(), it can happen that we may go out of bound as shown in the figure(below). So we need to handle it, we can return  $-1e9$ , whenever we go out of bound, in this way this path will not be selected by the calling function as we have to return the maximum chocolates.

### Going Downwards



### Going bottom-left or bottom-right



If  $j_1 < 0$  or  $j_1 > M$  or  $j_2 < 0$  or  $j_2 > M$ , then we return  $-1e9$

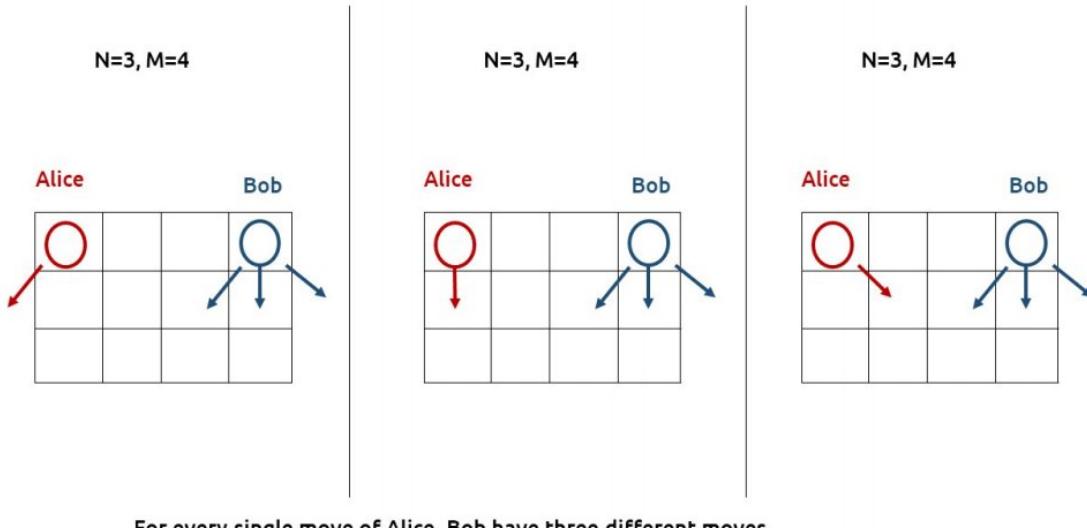
The pseudocode till this step will be:

```
f(i,j1,j2) {
    if(j1<0 || j1>=m || j2<0 || j2>=m)
        return -1e9
    if( i==N-1){
        if(j1==j2)
            return mat[i][j1]
        else
            return mat[i][j1] + mat[i][j2]
    }
}
```

**Step 2:** Try out all possible choices at a given index.

At every cell, we have three options to go: to the bottom cell ( $\downarrow$ ), to the bottom-right cell() or to the bottom-left cell()

Now, we need to understand that we want to move Alice and Bob together. Both of them can individually move three moves but say Alice moves to bottom-left, then Bob can have three different moves for Alice's move, and so on. The following figures will help to understand this:



**For every single move of Alice, Bob have three different moves**

Hence we have a total of 9 different options at every  $f(i,j_1,j_2)$  to move Alice and Bob. Now we can manually write these 9 options or we can observe a pattern in them, first Alice moves to one side and Bob tries all three choices, then again Alice moves, then Bob, and so on. This pattern can be easily captured by using two nested loops that change the column numbers( $j_1$  and  $j_2$ ).

**Note:** if  $(j_1 == j_2)$ , as discussed in the base case, we will only consider chocolates collected by one of them otherwise we will consider chocolates collected by both of them.

```

f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
        return -1e9
    if( i==N-1){
        if( j1==j2)
            return mat[i][j1]
        else
            return mat[i][j1] + mat[i][j2]
    }
    for(int di= -1; di<=1; di++){
        for(int dj= -1; dj<=1; dj++){
            if( j1==j2)
                return mat[i][j1] + f(i,j1+di,j2+dj)
            else
                return mat[i][j1] + mat[i][j2] + f(i,j1+di,j2+dj)
        }
    }
}

```

### Step 3: Take the maximum of all choices

As we have to find the **maximum chocolates collected** of all the possible paths, we will return the **maximum** of all the choices(the 9 choices of step 2). We will take a maxi variable( initialized to INT\_MIN). We will update maxi to the maximum of the previous maxi and the answer of the current choice. At last, we will return maxi from our function as the answer.

The final pseudocode after steps 1, 2, and 3:

```

f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
        return -1e9
    if( i==N-1){
        if( j1==j2)
            return mat[i][j1]
        else
            return mat[i][j1] + mat[i][j2]
    }

    maxi = INT_MIN
    for(int di= -1; di<=1; di++){
        for(int dj= -1; dj<=1; dj++){
            if( j1==j2)
                ans =  mat[i][j1] + f(i,j1+di,j2+dj)
            else
                ans =  mat[i][j1] + mat[i][j2] + f(i,j1+di,j2+dj)
            maxi = max(maxi,ans)
        }
    }

    return maxi
}

```

### Steps to memoize a recursive solution:

Before moving to the memoization steps, we need to understand the dp array we are taking. The recursive function has three parameters: i, j1, and j2. Therefore, we will also need to take a 3D DP Array. Its dimensions will be [N][M][M] because when we are moving, i can go from 0 to N-1, and j1 and j2 can go from 0 to M-1.

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [N][M][M], initialized to -1.

2. Whenever we want to find the answer of a particular row and column (say  $f(i,j_1,j_2)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[i][j_1][j_2] \neq -1$ ). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j_1][j_2]$  to the solution we get.

**Code:**

- C++ Code
- Java Code

```

#include<bits/stdc++.h>

using namespace std;

int maxChocoUtil(int i, int j1, int j2, int n, int m, vector < vector < int >>
& grid, vector < vector < vector < int >>> & dp) {
    if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
        return -1e9;

    if (i == n - 1) {
        if (j1 == j2)
            return grid[i][j1];
        else
            return grid[i][j1] + grid[i][j2];
    }

    if (dp[i][j1][j2] != -1)
        return dp[i][j1][j2];

    int maxi = INT_MIN;
    for (int di = -1; di <= 1; di++) {
        for (int dj = -1; dj <= 1; dj++) {
            int ans;
            if (j1 == j2)
                ans = grid[i][j1] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n, m, grid, dp);
            else
                ans = grid[i][j1] + grid[i][j2] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n,
                    m, grid, dp);
            maxi = max(maxi, ans);
        }
    }
    return dp[i][j1][j2] = maxi;
}

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {

    vector < vector < vector < int >>> dp(n, vector < vector < int >> (m, vector <
    int
    > (m, -1)));
}

return maxChocoUtil(0, 0, m - 1, n, m, grid, dp);
}

int main() {

    vector<vector<int>> matrix{
        {2,3,1,2},
        {3,4,2,2},
        {5,6,3,5},
    };

    int n = matrix.size();
    int m = matrix[0].size();

    cout << maximumChocolates(n, m, matrix);
}

```

## Output:

21

**Time Complexity:  $O(N \cdot M \cdot M) * 9$**

Reason: At max, there will be  $N \cdot M \cdot M$  calls of recursion to solve a new problem and in every call, two nested loops together run for 9 times.

**Space Complexity:  $O(N) + O(N \cdot M \cdot M)$**

Reason: We are using a recursion stack space:  $O(N)$ , where  $N$  is the path length and an external DP Array of size ' $N \cdot M \cdot M$ '.

**Steps to convert Recursive Solution to Tabulation one.**

For the tabulation approach, it is better to understand what a cell in the 3D DP array means. As we had done in memoization, we will initialize a  $dp[]$  array of size  $[N][M][M]$ .

So now, when we say  $dp[2][0][3]$ , what does it mean? It means that we are getting the value of maximum chocolates collected by Alice and Bob, when Alice is at  $(2,0)$  and Bob is at  $(2,3)$ .

The below figure gives us a bit more clarity.

**Understanding the 3D DP Grid**

dp[1][1][2]					dp[2][1][2]					dp[2][1][1]				
0	1	2	3		0	1	2	3		0	1	2	3	
0	2	3	1	2	0	2	3	1	2	0	2	3	1	2
1	3	4	2	2	1	3	4	2	2	1	3	4	2	2
2	5	6	3	5	2	5	6	3	5	2	5	6	3	5

dp[1][1][2] gives the Maximum chocolates collected when Alice is at (1,1) and Bob is at (1,2).

dp[2][1][2] gives the Maximum chocolates collected when Alice is at (2,1) and Bob is at (2,2).

dp[2][1][1] gives the Maximum chocolates collected when Alice is at (2,1) and Bob is at (2,1).

Next we need to initialize for the base value conditions. In the recursive code, our base condition is when we reach the last row, therefore in our dp array we will also initialize  $dp[n-1][j1][j2]$ , i.e (the last plane of 3D Array) as the base condition.  $Dp[n-1][j1][j2]$  means Alice is at  $(n-1, j1)$  and Bob is at  $(n-1, j2)$ . As this is the last row, its value will be equal to  $mat[i][j1]$ , if  $(j1 == j2)$  and  $mat[i][j1] + mat[i][j2]$  otherwise.

Once we have filled the last plane, we can move to the second-last plane and so on, we will need three nested loops to do this traversal.

The steps to convert to the tabular solution are given below:

- Declare a  $dp[]$  array of size  $[N][M][M]$
- First initialize the base condition values as explained above.
- We will then move from  $dp[n-2][][]$  to  $dp[0][][]$ . We will set three nested loops to do this traversal.
- Inside the three nested loops( say  $i,j_1,j_2$  as loop variables), we will use the recursive relations, i.e we will again set two nested loops to try all the nine options.
- The outer three loops are just for traversal, the inner two loops that run for 9 times mainly decide, what should be the value of the cell. If you are getting confused, please see the code.
- Inside the inner two nested loops, we will calculate an answer as we had done in the recursive relation, but this time using values from the next plane of the 3D DP Array(  $dp[i+1][x][y]$  instead of recursive calls, where  $x$  and  $y$  will vary according to inner 2 nested loops).
- At last we will set  $dp[i][j_1][j_2]$  as the maximum of all the 9 options.
- After the outer three nested loops iteration has ended, we will return  $dp[0][0][m-1]$  as our answer.

#### **Code:**

- C++ Code
- Java Code

```

#include<bits/stdc++.h>

using namespace std;

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
    // Write your code here.
    vector < vector < vector < int >> dp(n, vector < vector < int >> (m,
        vector < int > (m, 0)));

    for (int j1 = 0; j1 < m; j1++) {
        for (int j2 = 0; j2 < m; j2++) {
            if (j1 == j2)
                dp[n - 1][j1][j2] = grid[n - 1][j1];
            else
                dp[n - 1][j1][j2] = grid[n - 1][j1] + grid[n - 1][j2];
        }
    }

    //Outer Nested Loops for traversing DP Array
    for (int i = n - 2; i >= 0; i--) {
        for (int j1 = 0; j1 < m; j1++) {
            for (int j2 = 0; j2 < m; j2++) {

                int maxi = INT_MIN;

                //Inner nested loops to try out 9 options
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {

                        int ans;

                        if (j1 == j2)
                            ans = grid[i][j1];
                        else
                            ans = grid[i][j1] + grid[i][j2];

                        if ((j1 + di < 0 || j1 + di >= m) ||
                            (j2 + dj < 0 || j2 + dj >= m))

                            ans += -1e9;
                        else
                            ans += dp[i + 1][j1 + di][j2 + dj];

                        maxi = max(ans, maxi);
                    }
                }
                dp[i][j1][j2] = maxi;
            }
        }
    }

    return dp[0][0][m - 1];
}

int main() {

```

```

vector<vector<int>> matrix{
    {2,3,1,2},
    {3,4,2,2},
    {5,6,3,5},
};

int n = matrix.size();
int m = matrix[0].size();

cout << maximumChocolates(n, m, matrix);
}

```

### **Output:**

21

### **Time Complexity: O(N\*M\*M)\*9**

Reason: The outer nested loops run for  $(N \cdot M \cdot M)$  times and the inner two nested loops run for 9 times.

### **Space Complexity: O(N\*M\*M)**

Reason: We are using an external array of size ' $N \cdot M \cdot M$ '. The stack space will be eliminated.

### **Part 3: Space Optimization**

If we look closely, to compute  $dp[i][j_1][j_2]$ , we need values only from  $dp[i+1][\cdot][\cdot]$ . Therefore it is not necessary to store a three-dimensional array. Instead, we can store a two-dimensional array and update it as we move from one plane to the other in the 3D Array.

The Steps to space optimize the tabulation approach are as follows:

- Initially we can take a dummy 2D Array ( say front).We initialize this 2D Array as we had done in the Tabulation Approach.
- Next we also initialize a 2D Array( say cur), which we will need in the traversal.
- Now we set our three nested loops to traverse the 3D Array, from the second last plane.
- Following the same approach as we did in the tabulation approach, we find the maximum number of chocolates collected at each cell. To calculate it we have all the values in our 'front' 2D Array.
- Previously, we assigned  $dp[i][j_1][j_2]$  to maxi, now we will simply assign  $cur[j_1][j_2]$  to maxi.
- Then whenever the plane of the 3D DP(the first parameter) is going to change, we assign front to cur.

At last, we will return  $front[0][m-1]$  as our answer.

**Code:**

- C++ Code
- Java Code

```

#include<bits/stdc++.h>

using namespace std;

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
    // Write your code here.
    vector < vector < int >> front(m, vector < int > (m, 0)), cur(m, vector < int > (m, 0));

    for (int j1 = 0; j1 < m; j1++) {
        for (int j2 = 0; j2 < m; j2++) {
            if (j1 == j2)
                front[j1][j2] = grid[n - 1][j1];
            else
                front[j1][j2] = grid[n - 1][j1] + grid[n - 1][j2];
        }
    }

    //Outer Nested Loops for traversing DP Array
    for (int i = n - 2; i >= 0; i--) {
        for (int j1 = 0; j1 < m; j1++) {
            for (int j2 = 0; j2 < m; j2++) {

                int maxi = INT_MIN;

                //Inner nested loops to try out 9 options
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {

                        int ans;

                        if (j1 == j2)
                            ans = grid[i][j1];
                        else
                            ans = grid[i][j1] + grid[i][j2];

                        if ((j1 + di < 0 || j1 + di >= m) ||
                            (j2 + dj < 0 || j2 + dj >= m))

                            ans += -1e9;
                        else
                            ans += front[j1 + di][j2 + dj];

                        maxi = max(ans, maxi);

                    }
                }
                cur[j1][j2] = maxi;
            }
        }
        front = cur;
    }

    return front[0][m - 1];
}

```

```
int main() {  
  
    vector<vector<int> > matrix{  
        {2,3,1,2},  
        {3,4,2,2},  
        {5,6,3,5},  
    };  
  
    int n = matrix.size();  
    int m = matrix[0].size();  
  
    cout << maximumChocolates(n, m, matrix);  
}
```

### **Output:**

21

### **Time Complexity: O(N\*M\*M)\*9**

Reason: The outer nested loops run for  $(N*M*M)$  times and the inner two nested loops run for 9 times.

### **Space Complexity: O(M\*M)**

Reason: We are using an external array of size ' $M*M$ '.

# Subset sum equal to target (DP- 14)

 [takeuforward.org/data-structure/subset-sum-equal-to-target-dp-14](https://takeuforward.org/data-structure/subset-sum-equal-to-target-dp-14)

February 4, 2022

In this article, we will solve the most asked coding interview problem: Subset sum equal to target.

In this article, we will be going to understand the pattern of dynamic programming on subsequences of an array. We will be using the problem "Subset Sum Equal to K".

First, we need to understand **what a subsequence/subset is.**

A subset/subsequence is a contiguous or non-contiguous part of an array, where elements appear in the same order as the original array.

For example, for the array: [2,3,1] , the subsequences will be [{2},{3},{1},{2,3},{2,1},{3,1}, {2,3,1}] but {3,2} is **not** a subsequence because its elements are not in the same order as the original array.

**Problem Link:** [Subset Sum Equal to K](#)

We are given an array ‘ARR’ with N positive integers. We need to find if there is a subset in “ARR” with a sum equal to K. If there is, return true else return false.

**Example:**

Arr	1	2	3	4
-----	---	---	---	---

**Target: 4**

We will return **true**, as there are 2 subsets with sum equal to 4 {1,3} and {4}.

**Pre-req:** [Dynamic Programming Introduction](#), [Recursion on Subsequences](#)

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Solution :**

**Why a Greedy Solution doesn't work?**

---

A Greedy Solution doesn't make sense because we are not looking to optimize anything. We can rather try to generate all subsequences using recursion and whenever we get a single subsequence whose sum is equal to the given target, we can return true.

**Note:** Readers are highly advised to watch this video "[Recursion on Subsequences](#)" to understand how we generate subsequences using recursion.

### Steps to form the recursive solution:

---

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter "target". We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find( $n-1$ , target) which means that we need to find whether there exists a subsequence in the array from index 0 to  $n-1$ , whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

**f(ind,target) -> Check whether a subsequence exists in the Array from index 0 to ind, whose sum is equal to target**

### Base Cases:

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return true.
- If ind==0, it means we are at the first element, so we need to return  $\text{arr}[ind]==\text{target}$ . If the element is equal to the target we return true else false.

```
f(ind,target) {  
    if(target==0)  return true  
    if( ind==0) return arr[ind] == target  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video “[Recursion on Subsequences](#)”.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to  $f(ind-1, target)$ .
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $arr[ind]$ , the updated target which we need to find in the rest of the array will be  $target - arr[ind]$ . Therefore, we will call  $f(ind-1, target - arr[ind])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less or equal to the target.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
  
    bool notTaken = f(ind-1,target)  
    bool taken = false  
    if( arr[ind]<=target)  
        taken = f(ind-1,target - arr[ind])  
    }  
}
```

**Step 3: Return (taken || notTaken)**

As we are looking for only one subset, if any of the one among taken or not taken returns true, we can return true from our function. Therefore, we return ‘or(||)’ of both of them.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,target) {

    if(target==0) return true

    if( ind==0) return arr[ind] == target

    bool notTaken = f(ind-1,target)

    bool taken = false

    if( arr[ind]<=target)

        taken = f(ind-1,target - arr[ind])

    return notTaken || taken

}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][k+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as  $dp[n][k+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] != -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumUtil(int ind, int target, vector<int>& arr, vector<vector<int>> &dp)
{
    if(target==0)
        return true;

    if(ind == 0)
        return arr[0] == target;

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notTaken = subsetSumUtil(ind-1,target,arr,dp);

    bool taken = false;
    if(arr[ind]<=target)
        taken = subsetSumUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= notTaken||taken;
}

bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<vector<int>> dp(n, vector<int>(k+1,-1));

    return subsetSumUtil(n-1,k,arr,dp);
}

int main() {

    vector<int> arr = {1,2,3,4};
    int k=4;
    int n = arr.size();

    if(subsetSumToK(n,k,arr))
        cout<<"Subset with given target found";
    else
        cout<<"Subset with given target not found";
}

```

### **Output:**

Subset with given target found

### **Time Complexity: O(N\*K)**

Reason: There are N\*K states therefore at max 'N\*K' new problems will be solved.

### **Space Complexity: O(N\*K) + O(N)**

Reason: We are using a recursion stack space(O(N)) and a 2D array ( O(N\*K)).

### **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can set its type as bool and initialize it as false.

Eg: [4,2,3,7,...,23]

Target: 11

	target							
ind	0	1	2	3	4	...	K	
0	false	false	false	false	false	false	false	
1	false	false	false	false	false	false	false	
.	false	false	false	false	false	false	false	
N-1	false	false	false	false	false	false	false	

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as true.

Eg: [4,2,3,7,...,23]

Target: 11

	target							
ind	0	1	2	3	4	...	K	
0	true	false	false	false	false	false	false	
1	true	false	false	false	false	false	false	
.	true	false	false	false	false	false	false	
N-1	true	false	false	false	false	false	false	

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = \text{true}$ , ( $dp[0][arr[0]]$ ) means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > \text{target}$ , so we first check it: if( $arr[0] \leq \text{target}$ ) then set  $dp[0][arr[0]] = \text{true}$ .

Eg: [4,2,3,7,...,23]  
Target: 11

$arr[0] = 4$        $dp[0][4]$

target →

ind ↓

<del>target ind</del>	0	1	2	3	4	...	K
0	true	false	false	false	true	false	false
1	true	false	false	false	false	false	false
⋮	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

- After that , we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return  $dp[n-1][k]$  as our answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<vector<bool>> dp(n, vector<bool>(k+1, false));

    for(int i=0; i<n; i++){
        dp[i][0] = true;
    }

    if(arr[0]<=k)
        dp[0][arr[0]] = true;

    for(int ind = 1; ind<n; ind++){
        for(int target= 1; target<=k; target++){

            bool notTaken = dp[ind-1][target];

            bool taken = false;
            if(arr[ind]<=target)
                taken = dp[ind-1][target-arr[ind]];

            dp[ind][target]= notTaken||taken;
        }
    }

    return dp[n-1][k];
}

int main() {

    vector<int> arr = {1,2,3,4};
    int k=4;
    int n = arr.size();

    if(subsetSumToK(n,k,arr))
        cout<<"Subset with given target found";
    else
        cout<<"Subset with given target not found";
}

```

### **Output:**

Subset with given target found

### **Time Complexity: O(N\*K)**

Reason: There are two nested loops

### **Space Complexity: O(N\*K)**

Reason: We are using an external array of size 'N\*K'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] \mid\mid dp[ind-1][target-arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element is true according to our base condition.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<bool> prev(k+1, false);

    prev[0] = true;

    if(arr[0]<=k)
        prev[arr[0]] = true;

    for(int ind = 1; ind<n; ind++){
        vector<bool> cur(k+1, false);
        cur[0] = true;
        for(int target= 1; target<=k; target++){
            bool notTaken = prev[target];

            bool taken = false;
            if(arr[ind]<=target)
                taken = prev[target-arr[ind]];

            cur[target]= notTaken| taken;
        }
        prev = cur;
    }

    return prev[k];
}

int main() {

    vector<int> arr = {1,2,3,4};
    int k=4;
    int n = arr.size();

    if(subsetSumToK(n,k,arr))
        cout<<"Subset with given target found";
    else
        cout<<"Subset with given target not found";
}

```

## **Output:**

Subset with given target found

## **Time Complexity: O(N\*K)**

Reason: There are three nested loops

## **Space Complexity: O(K)**

Reason: We are using an external array of size ‘K+1’ to store only one row.



# Partition Equal Subset Sum (DP- 15)

 [takeuforward.org/data-structure/partition-equal-subset-sum-dp-15](https://takeuforward.org/data-structure/partition-equal-subset-sum-dp-15)

February 4, 2022

## Problem Link: Partition Equal Subset Sum

We are given an array ‘ARR’ with N positive integers. We need to find if we can partition the array into two subsets such that the sum of elements of each subset is equal to the other.

If we can partition, return true else return false

**Example:**

Arr	2	3	3	3	4	5
-----	---	---	---	---	---	---

We can partition the array in two subsequences.

{2,3,5}

{3,3,4}

Sum = 10

In this example, as we are able to partition, we return true.

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** Subset sum equal to target, Recursion on Subsequences

**Solution :**

This question is a slight modification of the problem discussed in Subset-sum equal to target. We need to partition the array(say S) into two subsets(say S1 and S2). According to the question:

- Sum of elements of S1 + sum of elements of S2 = sum of elements of S.

- Sum of elements of S<sub>1</sub> = sum of elements of S<sub>2</sub>.

These two conditions imply that S<sub>1</sub> = S<sub>2</sub> = (S/2).

Now,

- If S (sum of elements of the input array) is odd , there is no way we can divide it into two equal halves, so we can simply return false.
- If S is even, then we need to find a subsequence in the input array whose sum is equal to S/2 because if we find one subsequence with sum S/2, the remaining elements sum will be automatically S/2. So, we can partition the given array. Hence we return true.

**Note:** Readers are highly advised to watch this video "[Recursion on Subsequences](#)" to understand how we generate subsequences using recursion.

From here we will try to find a subsequence in the array with target = S/2 as discussed in [\*\*Subset-sum equal to the target\*\*](#)

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to n-1, whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

**f(ind,target) -> Check whether a subsequence exists in the Array from index 0 to ind, whose sum is equal to target**

### Base Cases:

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return true.
- If ind==0, it means we are at the first element, so we need to return arr[ind]==target. If the element is equal to the target we return true else false.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
  
    }  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to  $f(ind-1, \text{target})$ .
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $\text{arr}[ind]$ , the updated target which we need to find in the rest of the array will be  $\text{target} - \text{arr}[ind]$ . Therefore, we will call  $f(ind-1, \text{target}-\text{arr}[ind])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less or equal to the target.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
  
    bool notTaken = f(ind-1,target)  
  
    bool taken = false  
    if( arr[ind]<=target)  
        taken = f(ind-1,target - arr[ind])  
  
    }  
}
```

### Step 3: Return (taken || notTaken)

As we are looking for only one subset, if any of the one among taken or not taken returns true, we can return true from our function. Therefore, we return ‘or(||)’ of both of them.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,target) {

    if(target==0) return true

    if( ind==0) return arr[ind] == target

    bool notTaken = f(ind-1,target)

    bool taken = false

    if( arr[ind]<=target)

        taken = f(ind-1,target - arr[ind])

    return notTaken || taken

}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][k+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as  $dp[n][k+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] != -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumUtil(int ind, int target, vector<int>& arr, vector<vector<int>> &dp)
{
    if(target==0)
        return true;

    if(ind == 0)
        return arr[0] == target;

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notTaken = subsetSumUtil(ind-1,target,arr,dp);

    bool taken = false;
    if(arr[ind]<=target)
        taken = subsetSumUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= notTaken||taken;
}

bool canPartition(int n, vector<int> &arr){

    int totSum=0;

    for(int i=0; i<n;i++){
        totSum+= arr[i];
    }

    if (totSum%2==1) return false;

    else{
        int k = totSum/2;
        vector<vector<int>> dp(n, vector<int>(k+1,-1));
        return subsetSumUtil(n-1,k,arr,dp);
    }
}

int main() {

    vector<int> arr = {2,3,3,3,4,5};
    int n = arr.size();

    if(canPartition(n,arr))
        cout<<"The Array can be partitioned into two equal subsets";
    else
        cout<<"The Array cannot be partitioned into two equal subsets";
}

```

### **Output:**

The array can be partitioned into two equal subsets

## Time Complexity: $O(N*K) + O(N)$

Reason: There are  $N*K$  states therefore at max ' $N*K$ ' new problems will be solved and we are running a for loop for ' $N$ ' times to calculate the total sum

## Space Complexity: $O(N*K) + O(N)$

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (  $O(N*K)$ ).

### Steps to convert Recursive Solution to Tabulation one.

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can set its type as bool and initialize it as false.

Eg: [4,2,3,7,...,23]

Target: 11

target ind	0	1	2	3	4	...	K
0	false						
1	false						
.	false						
N-1	false						

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as true.

Eg: [4,2,3,7,...,23]

Target: 11

target →

target ind \	0	1	2	3	4	...	K
0	true	false	false	false	false	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = \text{true}$ , ( $dp[0][arr[0]]$  means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > \text{target}$ , so we first check it: if( $arr[0] \leq \text{target}$ ) then set  $dp[0][arr[0]] = \text{true}$ .

Eg: [4,2,3,7,...,23]

Target: 11

target →

$arr[0] = 4$        $dp[0][4]$

target ind \	0	1	2	3	4	...	K
0	true	false	false	false	true	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

- After that, we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return  $dp[n-1][k]$  as our answer.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

bool canPartition(int n, vector<int> &arr){

    int totSum=0;

    for(int i=0; i<n; i++){
        totSum+= arr[i];
    }

    if (totSum%2==1) return false;

    else{
        int k = totSum/2;
        vector<vector<bool>> dp(n, vector<bool>(k+1, false));

        for(int i=0; i<n; i++){
            dp[i][0] = true;
        }

        if(arr[0]<=k)
            dp[0][arr[0]] = true;

        for(int ind = 1; ind<n; ind++){
            for(int target= 1; target<=k; target++){

                bool notTaken = dp[ind-1][target];

                bool taken = false;
                if(arr[ind]<=target)
                    taken = dp[ind-1][target-arr[ind]];

                dp[ind][target]= notTaken||taken;
            }
        }

        return dp[n-1][k];
    }
}

int main() {

    vector<int> arr = {2,3,3,3,4,5};
    int n = arr.size();

    if(canPartition(n,arr))
        cout<<"The Array can be partitioned into two equal subsets";
    else
        cout<<"The Array cannot be partitioned into two equal subsets";
}

```

### **Output:**

The array can be partitioned into two equal subsets

## **Time Complexity: O(N\*K) +O(N)**

Reason: There are two nested loops that account for O(N\*K) and at starting we are running a for loop to calculate totSum.

## **Space Complexity: O(N\*K)**

Reason: We are using an external array of size 'N\*K'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] \mid\mid dp[ind-1][target-arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element is true according to our base condition.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool canPartition(int n, vector<int> &arr){

    int totSum=0;

    for(int i=0; i<n; i++){
        totSum+= arr[i];
    }

    if (totSum%2==1) return false;

    else{
        int k = totSum/2;
        vector<bool> prev(k+1, false);

        prev[0] = true;

        if(arr[0]<=k)
            prev[arr[0]] = true;

        for(int ind = 1; ind<n; ind++){
            vector<bool> cur(k+1, false);
            cur[0] = true;
            for(int target= 1; target<=k; target++){
                bool notTaken = prev[target];

                bool taken = false;
                if(arr[ind]<=target)
                    taken = prev[target-arr[ind]];

                cur[target]= notTaken||taken;
            }
            prev = cur;
        }

        return prev[k];
    }
}

int main() {

    vector<int> arr = {2,3,3,3,4,5};
    int n = arr.size();

    if(canPartition(n,arr))
        cout<<"The Array can be partitioned into two equal subsets";
    else
        cout<<"The Array cannot be partitioned into two equal subsets";
}

```

### **Output:**

The Array can be partitioned into two equal subsets

**Time Complexity: O(N\*K) +O(N)**

Reason: There are two nested loops that account for  $O(N*K)$  and at starting we are running a for loop to calculate totSum.

**Space Complexity: O(K)**

Reason: We are using an external array of size ' $K+1$ ' to store only one row.

# Partition Set Into 2 Subsets With Min Absolute Sum Diff (DP- 16)

 [takeuforward.org/data-structure/partition-set-into-2-subsets-with-min-absolute-sum-diff-dp-16](https://takeuforward.org/data-structure/partition-set-into-2-subsets-with-min-absolute-sum-diff-dp-16)

February 7, 2022

## Problem Statement:

Partition A Set Into Two Subsets With Minimum Absolute Sum Difference

Pre-req: Subset Sum equal to target, Recursion on Subsequences

## Problem Link: Partition A Set Into Two Subsets With Minimum Absolute Sum Difference

We are given an array 'ARR' with N positive integers. We need to partition the array into two subsets such that the absolute difference of the sum of elements of the subsets is minimum.

We need to return only the minimum absolute difference of the sum of elements of the two partitions.

## Examples:

Example 1:

Arr	1	2	3	4
-----	---	---	---	---

We can partition the array in these two subsequences.

{1,4}

{2,3}

$$\text{Minimum absolute difference} = (1+4) - (2+3) = 0$$

**Example 2:**

Arr	8	6	5
-----	---	---	---

We can partition the array in these two subsequences.

{8}

{6,5}

Minimum absolute difference =  $(8) - (6+5) = 3$

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

### Solution :

---

This question is a slight modification of the problem discussed in the Subset Sum equal to target.

Before discussing the approach for this question, it is important to understand what we did in the previous question of the Subset Sum equal to the target. There we found whether or not a subset exists in an array with a given target sum. We used a dp array to get to our answer.

Eg: [4,2,3,7,...,23]

Target: 11

target ind	0	1	2	3	4	...	K
0	true	false	false	false	true	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

We used to return  $dp[n-1][k]$  as our answer. One interesting thing that is happening is that for calculating our answer for  $dp[n-1][k]$ , we are also solving multiple sub-problems and at the same time storing them as well. We will use this property to solve the question of this article.

In this question, we need to partition the array into two subsets( say with sum  $S_1$  and  $S_2$ ) and we need to return the minimized absolute difference of  $S_1$  and  $S_2$ . But do we need two variables for it? The answer is **No**. We can use a variable  $totSum$ , which stores the sum of all elements of the input array, and then we can simply say  $S_2 = totSum - S_1$ . Therefore we only need one variable  $S_1$ .

Now, what values can  $S_1$  take? Well, it can go anywhere from 0 (no elements in  $S_1$ ) to  $totSum$  (all elements in  $S_1$ ). If we observe the last row of the dp array which we had discussed above, it gives us the targets for which there exists a subset. We will set its column value to  $totSum$ , to find the answer from 0(smaller limit of  $S_1$ ) to  $totSum$  (the larger limit of  $S_1$ ).

Our work is very simple, using the last row of the dp array, we will first find which all  $S_1$  values are valid. Using the valid  $S_1$  values, we will find  $S_2$  ( $totSum - S_1$ ). From this  $S_1$  and  $S_2$ , we will find their absolute difference. We will return the minimum value of this absolute difference as our answer.

From here we will try to find a subsequence in the array with a target as discussed in Subset Sum equal to target after generating the dp array, we will use the last row to find our answer.

**Note:** Readers are highly advised to watch this video "Recursion on Subsequences" to understand how we generate subsequences using recursion.

**Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find( $n-1$ , target) which means that we need to find whether there exists a subsequence in the array from index 0 to  $n-1$ , whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

**f(ind,target) -> Check whether a subsequence exists in the Array from index 0 to ind, whose sum is equal to target**

**Base Cases:**

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return true.
- If ind==0, it means we are at the first element, so we need to return  $\text{arr}[ind]==\text{target}$ . If the element is equal to the target we return true else false.

```
f(ind,target) {  
    if(target==0)  return true  
    if( ind==0) return arr[ind] == target  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video “[Recursion on Subsequences](#)”.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to  $f(ind-1, target)$ .
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $arr[ind]$ , the updated target which we need to find in the rest of the array will be  $target - arr[ind]$ . Therefore, we will call  $f(ind-1, target - arr[ind])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less or equal to the target.

```

f(ind,target) {

    if(target==0) return true

    if( ind==0) return arr[ind] == target

    bool notTaken = f(ind-1,target)

    bool taken = false

    if( arr[ind]<=target)

        taken = f(ind-1,target - arr[ind])

    }

```

### Step 3: Return (taken || notTaken)

As we are looking for only one subset, if any of the one among taken or not taken returns true, we can return true from our function. Therefore, we return ‘or(||)’ of both of them.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,target) {

    if(target==0) return true

    if( ind==0) return arr[ind] == target

    bool notTaken = f(ind-1,target)

    bool taken = false

    if( arr[ind]<=target)

        taken = f(ind-1,target - arr[ind])

    return notTaken || taken

}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][\text{totSum}+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘totSum’. Therefore we take the dp array as  $\text{dp}[n][\text{totSum}+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(\text{ind}, \text{target})$ ), we first check whether the answer is already calculated using the dp array(i.e  $\text{dp}[\text{ind}][\text{target}] != -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $\text{dp}[\text{ind}][\text{target}]$  to the solution we get.
5. When we get the dp array, we will use its last row to find the absolute minimum difference of two partitions.

### **Code:**

C++ Code

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumUtil(int ind, int target, vector < int > & arr, vector < vector
< int >> & dp) {
    if (target == 0)
        return dp[ind][target]=true;

    if (ind == 0)
        return dp[ind][target] = arr[0] == target;

    if (dp[ind][target] != -1)
        return dp[ind][target];

    bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);

    bool taken = false;
    if (arr[ind] <= target)
        taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

    return dp[ind][target] = notTaken || taken;
}

int minSubsetSumDifference(vector < int > & arr, int n) {

    int totSum = 0;

    for (int i = 0; i < n; i++)
        totSum += arr[i];
}

vector < vector < int >> dp(n, vector < int > (totSum + 1, -1));

for (int i = 0; i <= totSum; i++) {
    bool dummy = subsetSumUtil(n - 1, i, arr, dp);
}

int mini = 1e9;
for (int i = 0; i <= totSum; i++) {
    if (dp[n - 1][i] == true) {
        int diff = abs(i - (totSum - i));
        mini = min(mini, diff);
    }
}
return mini;
}

int main() {

    vector < int > arr = {1,2,3,4};
    int n = arr.size();

    cout << "The minimum absolute difference is: " << minSubsetSumDifference(arr,
n);
}

```

}

### Output:

The minimum absolute difference is: 0

### Time Complexity: $O(N*totSum) + O(N) + O(N)$

Reason: There are two nested loops that account for  $O(N*totSum)$ , at starting we are running a for loop to calculate totSum and at last a for loop to traverse the last row.

### Space Complexity: $O(N*totSum) + O(N)$

Reason: We are using an external array of size ' $N * totSum$ ' and a stack space of  $O(N)$ .

### Steps to convert Recursive Solution to Tabulation one.

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can set its type as bool and initialize it as false.

Eg: [4,2,3,7,...,23]  
Target: 11

target →

target \ ind	0	1	2	3	4	...	K
0	false						
1	false						
.	false						
N-1	false						

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as true.

Eg: [4,2,3,7,...,23]

Target: 11

<del>target ind</del>	0	1	2	3	4	...	K
0	true	false	false	false	false	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = \text{true}$ , ( $dp[0][arr[0]]$  means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > \text{target}$ , so we first check it: if( $arr[0] \leq \text{target}$ ) then set  $dp[0][arr[0]] = \text{true}$ .

Eg: [4,2,3,7,...,23]

Target: 11

<del>target ind</del>	0	1	2	3	4	...	K
0	true	false	false	false	<span style="border: 1px solid red; border-radius: 50%; padding: 2px;">true</span>	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
N-1	true	false	false	false	false	false	false

- After that , we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- When we get the dp array, we will use its last row to find the absolute minimum difference of two partitions.

**Code:**

C++ Code

```
#include <bits/stdc++.h>

using namespace std;

int minSubsetSumDifference(vector < int > & arr, int n) {
    int totSum = 0;

    for (int i = 0; i < n; i++) {
        totSum += arr[i];
    }

    vector < vector < bool >> dp(n, vector < bool > (totSum + 1, false));

    for (int i = 0; i < n; i++) {
        dp[i][0] = true;
    }

    if (arr[0] <= totSum)
        dp[0][totSum] = true;

    for (int ind = 1; ind < n; ind++) {
        for (int target = 1; target <= totSum; target++) {

            bool notTaken = dp[ind - 1][target];

            bool taken = false;
            if (arr[ind] <= target)
                taken = dp[ind - 1][target - arr[ind]];

            dp[ind][target] = notTaken || taken;
        }
    }

    int mini = 1e9;
    for (int i = 0; i <= totSum; i++) {
        if (dp[n - 1][i] == true) {
            int diff = abs(i - (totSum - i));
            mini = min(mini, diff);
        }
    }
    return mini;
}

int main() {

    vector<int> arr = {1,2,3,4};
    int n = arr.size();

    cout << "The minimum absolute difference is: " << minSubsetSumDifference(arr,
n);
}
```

## **Output:**

The minimum absolute difference is: 0

## **Time Complexity: O(N\*totSum) +O(N) +O(N)**

Reason: There are two nested loops that account for  $O(N*totSum)$ , at starting we are running a for loop to calculate totSum and at last a for loop to traverse the last row.

## **Space Complexity: O(N\*totSum)**

Reason: We are using an external array of size 'N \* totSum'. Stack Space is eliminated.

## **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] \mid\mid dp[ind-1][target-arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element is true according to our base condition.

## **Code:**

C++ Code

```

#include <bits/stdc++.h>

using namespace std;

int minSubsetSumDifference(vector < int > & arr, int n) {
    int totSum = 0;

    for (int i = 0; i < n; i++) {
        totSum += arr[i];
    }

    vector < bool > prev(totSum + 1, false);

    prev[0] = true;

    if (arr[0] <= totSum)
        prev[arr[0]] = true;

    for (int ind = 1; ind < n; ind++) {
        vector < bool > cur(totSum + 1, false);
        cur[0] = true;
        for (int target = 1; target <= totSum; target++) {
            bool notTaken = prev[target];

            bool taken = false;
            if (arr[ind] <= target)
                taken = prev[target - arr[ind]];

            cur[target] = notTaken || taken;
        }
        prev = cur;
    }

    int mini = 1e9;
    for (int i = 0; i <= totSum; i++) {
        if (prev[i] == true) {
            int diff = abs(i - (totSum - i));
            mini = min(mini, diff);
        }
    }
    return mini;
}

int main() {

    vector<int> arr = {1,2,3,4};
    int n = arr.size();

    cout << "The minimum absolute difference is: " << minSubsetSumDifference(arr,
n);
}

```

### **Output:**

The minimum absolute difference is: 0

**Time Complexity: O(N\*totSum) +O(N) +O(N)**

Reason: There are two nested loops that account for  $O(N*totSum)$ , at starting we are running a for loop to calculate totSum and at last a for loop to traverse the last row.

**Space Complexity: O(totSum)**

Reason: We are using an external array of size ‘totSum+1’ to store only one row.

# Count Subsets with Sum K (DP – 17)

---

 [takeuforward.org/data-structure/count-subsets-with-sum-k-dp-17](https://takeuforward.org/data-structure/count-subsets-with-sum-k-dp-17)

February 11, 2022

## Problem Statement: Count Subsets with Sum K

Pre-req: Subset Sum equal to target, Recursion on Subsequences

## Problem Link: Count Subsets With Sum K

We are given an array ‘ARR’ with N positive integers and an integer K. We need to find the number of subsets whose sum is equal to K.

**Example:**

Arr	1	2	2	3	K = 3
-----	---	---	---	---	-------

We can have the following unique subsets with target Sum of 3

$$\begin{array}{ll} [1,2] & [1,2] \\ \text{Arr[0]} \quad \text{Arr[1]} & \text{Arr[0]} \quad \text{Arr[2]} \\ & [3] \\ & \text{Arr[3]} \end{array}$$

**Total Count of subsets: 3**

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

## Solution :

---

A Greedy Solution doesn't make sense because we are not looking to optimize anything. We can rather try to generate all subsequences using recursion and whenever we get a single subsequence whose sum is equal to the given target, we can count it.

**Note:** Readers are highly advised to watch this video “Recursion on Subsequences” to understand how we generate subsequences using recursion.

## Steps to form the recursive solution:

---

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to  $f(n-1, \text{target})$  which means that we are counting the number of subsequences in the array from index 0 to  $n-1$ , whose sum is equal to the target. Similarly, we can generalize it for any index  $ind$  as follows:

**$f(ind, \text{target}) \rightarrow$  Count Number of subsequences that exists in the Array from index 0 to  $ind$ , whose sum is equal to  $\text{target}$**

**Base Cases:**

- If  $\text{target} == 0$ , it means that we have already found the subsequence from the previous steps, so we can return 1.
- If  $ind==0$ , it means we are at the first element, so we need to return  $\text{arr}[ind]==\text{target}$ . If the element is equal to the target we return 1 else we return 0.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video “[Recursion on Subsequences](#)”.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to  $f(ind-1, \text{target})$ .

- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $\text{arr}[\text{ind}]$ , the updated target which we need to find in the rest of the array will be  $\text{target} - \text{arr}[\text{ind}]$ . Therefore, we will call  $f(\text{ind}-1, \text{target} - \text{arr}[\text{ind}])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less than or equal to the target.

```

f(ind,target) {
    if(target==0) return 1
    if( ind==0) return arr[ind] == target

    notTaken = f(ind-1,target)

    taken = 0
    if( arr[ind]<=target)
        taken = f(ind-1,target - arr[ind])

}

```

### Step 3: Return sum of taken and notTaken

As we have to return the total count of subsets with the target sum, we will return the sum of taken and notTaken from our recursive call.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,target) {
    if(target==0) return 1
    if( ind==0) return arr[ind] == target

    notTaken = f(ind-1,target)

    taken = 0
    if( arr[ind]<=target)
        taken = f(ind-1,target - arr[ind])

    return notTaken + taken
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][k+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as  $dp[n][k+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int findWaysUtil(int ind, int target, vector<int>& arr, vector<vector<int>> &dp){
    if(target==0)
        return 1;

    if(ind == 0)
        return arr[0] == target;

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    int notTaken = findWaysUtil(ind-1,target,arr,dp);

    int taken = 0;
    if(arr[ind]<=target)
        taken = findWaysUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= notTaken + taken;
}

int findWays(vector<int> &num, int k){
    int n = num.size();
    vector<vector<int>> dp(n, vector<int>(k+1, -1));
    return findWaysUtil(n-1,k,num,dp);
}

int main() {

    vector<int> arr = {1,2,2,3};
    int k=3;

    cout<<"The number of subsets found are " <<findWays(arr,k);
}

```

### **Output:**

The number of subsets found are 3

### **Time Complexity: O(N\*K)**

Reason: There are N\*K states therefore at max 'N\*K' new problems will be solved.

### **Space Complexity: O(N\*K) + O(N)**

Reason: We are using a recursion stack space(O(N)) and a 2D array ( O(N\*K)).

### **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

Eg: [4,2,3,7,...,23]

Target: 11

target →

target ind	0	1	2	3	4	...	K
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0
N-1	0	0	0	0	0	0	0

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as 1.

Eg: [4,2,3,7,...,23]

Target: 11

target →

target ind	0	1	2	3	4	...	K
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = 1$ , ( $dp[0][arr[0]]$  means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > target$ , so we first check it: if( $arr[0] \leq target$ ) then set  $dp[0][arr[0]] = 1$ .

Eg: [4,2,3,7,...,23]

Target: 11

target ind	0	1	2	3	4	...	K
0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

- After that , we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return  $dp[n-1][k]$  as our answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int findWays(vector<int> &num, int k){
    int n = num.size();

    vector<vector<int>> dp(n, vector<int>(k+1, 0));

    for(int i=0; i<n; i++){
        dp[i][0] = 1;
    }

    if(num[0]<=k)
        dp[0][num[0]] = 1;

    for(int ind = 1; ind<n; ind++){
        for(int target= 1; target<=k; target++){

            int notTaken = dp[ind-1][target];

            int taken = 0;
            if(num[ind]<=target)
                taken = dp[ind-1][target-num[ind]];

            dp[ind][target]= notTaken + taken;
        }
    }

    return dp[n-1][k];
}

int main() {

    vector<int> arr = {1,2,2,3};
    int k=3;

    cout<<"The number of subsets found are " <<findWays(arr,k);
}

```

### **Output:**

The number of subsets found are 3

**Time Complexity: O(N\*K)**

Reason: There are two nested loops

**Space Complexity: O(N\*K)**

Reason: We are using an external array of size ‘N\*K’. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] + dp[ind-1][target - arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element is true according to our base condition.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int findWays(vector<int> &num, int k){
    int n = num.size();

    vector<int> prev(k+1, 0);

    prev[0] = 1;

    if(num[0]<=k)
        prev[num[0]] = 1;

    for(int ind = 1; ind<n; ind++){
        vector<int> cur(k+1, 0);
        cur[0]=1;
        for(int target= 1; target<=k; target++){

            int notTaken = prev[target];

            int taken = 0;
            if(num[ind]<=target)
                taken = prev[target-num[ind]];

            cur[target]= notTaken + taken;
        }

        prev = cur;
    }

    return prev[k];
}

int main() {

    vector<int> arr = {1,2,2,3};
    int k=3;

    cout<<"The number of subsets found are " <<findWays(arr,k);
}

```

### **Output:**

The number of subsets found are 3

### **Time Complexity: O(N\*K)**

Reason: There are three nested loops

### **Space Complexity: O(K)**

Reason: We are using an external array of size ‘K+1’ to store only one row.



# Count Partitions with Given Difference (DP – 18)

 [takeuforward.org/data-structure/count-partitions-with-given-difference-dp-18](https://takeuforward.org/data-structure/count-partitions-with-given-difference-dp-18)

February 14, 2022

## Problem Statement: Count Partitions with Given Difference

This article will be divided into two parts:

- First, we will discuss an extra edge case of the problem discussed in [Count Subsets with Sum K](#), and then,
- we will discuss the problem for this article: [Partitions with Given Difference](#).

### Part 1: Extra edge case for the problem [Count Subsets with Sum K](#)

In the problem [Count Subsets with Sum K](#), the problem constraints stated that an array element is greater than 0, so the code we have written there works perfectly for the given constraints.

If the constraints mentioned that an array element can also be equal to 0 and the target sum can also be 0, then that code will fail. To understand it we will take an example:

Let the target **arr = [0,0,1]** and the **target = 1**.

The previous code will give us the **answer 1** as it first takes the element arr[2] and then finds the answer by picking it. Then from the base condition, we will return 0 (as the target will become 0 by picking 1). But for this question, the answer will be **4** with the following subsets({0,1},{0,1},{0,0,1} and {1}).

Therefore we need to modify the base conditions in order to handle the changes. These are the base conditions of that problem.

```
f(ind,target) {  
    if( target == 0) return 1  
    if( ind == 0)  return arr[ind]==target  
    ...  
}
```

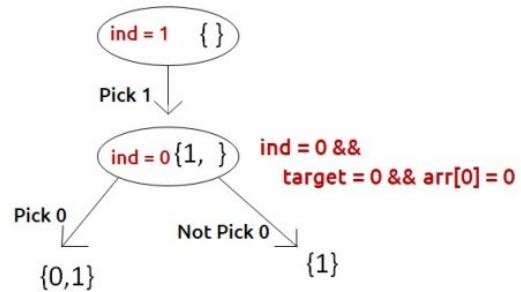
First of all, we will remove target==0 because now when target ==0, there can be many 0s present in the array which needs to be counted in the answer.

Now, the following cases can arise when we are at index 0, if the target sum is 0 and the first index is also 0, like in case [0,1], we can form the subset in two ways, either by considering the first element or leaving it, so we can return 2.

```
f(ind,target) {
    if( ind == 0)  return arr[ind]==target
}
```

```
f(ind,target) {
    if( ind == 0) {
        if( target ==0 && arr[0] == 0)
            return 2
    }
}
```

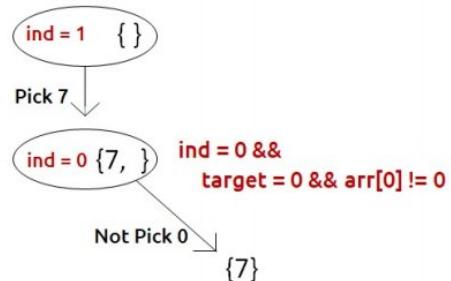
arr: [0,1] target :1



Else at index 0, if target == 0, and the first element is not 0, it means we will not pick the first element so we just return 1 way.

```
f(ind,target) {
    if( ind == 0) {
        if( target ==0 && arr[0] == 0)
            return 2
        if(target==0)
            return 1
    }
}
```

arr: [5,7] target :7

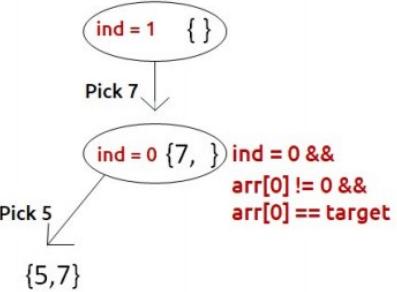


Or if at index 0, when the first element is not 0, and the target is equal to the first element , then we will include it in the subset and we will return 1 way.

```
f(ind,target) {
```

```
    if( ind == 0) {  
        if( target ==0 && arr[0] == 0)  
            return 2  
        if(target==0 || arr[0] == target)  
            return 1  
    }
```

arr: [5,7]    target :12



Else in all other cases, we simply return 0.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int findWaysUtil(int ind, int target, vector<int>& arr, vector<vector<int>> &dp){

    if(ind == 0){
        if(target==0 && arr[0]==0)
            return 2;
        if(target==0 || target == arr[0])
            return 1;
        return 0;
    }

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    int notTaken = findWaysUtil(ind-1,target,arr,dp);

    int taken = 0;
    if(arr[ind]<=target)
        taken = findWaysUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= notTaken + taken;
}

int findWays(vector<int> &num, int k){

    int n = num.size();
    vector<vector<int>> dp(n, vector<int>(k+1, -1));
    return findWaysUtil(n-1,k,num,dp);
}

int main() {

    vector<int> arr = {0,0,1};
    int k=1;

    cout<<"The number of subsets found are " <<findWays(arr,k);
}

```

### **Output:**

The number of subsets found are 4

### **Part 2:** Count Partitions with Given Difference

#### **Problem Link:** [Partitions with Given Difference](#)

#### **Problem Description:**

We are given an array ‘ARR’ with N positive integers and an integer D. We need to count the number of ways we can partition the given array into two subsets, S<sub>1</sub> and S<sub>2</sub> such that S<sub>1</sub> – S<sub>2</sub> = D and S<sub>1</sub> is always greater than or equal to S<sub>2</sub>.

**Example:**

Arr	<table border="1"><tr><td>5</td><td>2</td><td>6</td><td>4</td></tr></table>	5	2	6	4	D = 3
5	2	6	4			

**We can partition the array in only 1 way.**

$$\{6 + 4\} - \{5 + 2\} = 3$$

**Number of partitions: 1**

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: [Count Subsets with Sum K](#)

**Solution :**

---

This question is a slight modification of the problem discussed in [Count Subsets with Sum K](#).

We have the following two conditions given to us.

$$S_1 + S_2 = D \quad \text{-- (i)}$$

$$S_1 \geq S_2 \quad \text{-- (ii)}$$

If we calculate the total sum of elements of the array (say totSum), we can say that,

$$S_1 = \text{totSum} - S_2 \quad \text{-- (iii)}$$

Now solving for equations (i) and (iii), we can say that

$$S_2 = (\text{totSum} - D)/2 \quad \text{-- (iv)}$$

Therefore the question "Count Partitions with a difference D" is modified to "Count Number of subsets with sum  $(\text{totSum} - D)/2$ ". This is exactly what we had discussed in the article [Count Subsets with Sum K](#).

## **Edge Cases:**

The following edge cases need to be handled:

- As the array elements are positive integers including zero, we don't want to find the case when S2 is negative or we can say that totSum is lesser than D, therefore if totSum < D, we simply return 0.
- S2 can't be a fraction, as all elements are integers, therefore if totSum – D is odd, we can return 0.

From here on we will discuss the approach to “Count Subsets with Sum K” with the required modifications. Moreover, as the array elements can also contain 0, we will handle it as discussed in part-1 of this article.

## **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find(n-1, target) which means that we are counting the number of subsequences in the array from index 0 to n-1, whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

**f(ind,target) -> Count Number of subsequences that exists in the Array from index 0 to ind, whose sum is equal to target**

## **Base Cases:**

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return 1.
- If ind==0, it means we are at the first element, so we need to return arr[ind]==target. If the element is equal to the target we return 1 else we return 0.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
  
    }  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to  $f(ind-1, target)$ .
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $arr[ind]$ , the updated target which we need to find in the rest of the array will be  $target - arr[ind]$ . Therefore, we will call  $f(ind-1, target - arr[ind])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less than or equal to the target.

```
f(ind,target) {  
    if(target==0) return 1  
    if( ind==0) return arr[ind] == target  
  
    notTaken = f(ind-1,target)  
  
    taken = 0  
    if( arr[ind]<=target)  
        taken = f(ind-1,target - arr[ind])  
  
    }  
}
```

### Step 3: Return sum of taken and notTaken

As we have to return the total count of subsets with the target sum, we will return the sum of taken and notTaken from our recursive call.

The final pseudocode after steps 1, 2, and 3:

```
f(ind,target) {  
    if(target==0) return 1  
    if( ind==0) return arr[ind] == target  
  
    notTaken = f(ind-1,target)  
  
    taken = 0  
    if( arr[ind]<=target)  
        taken = f(ind-1,target - arr[ind])  
  
    return notTaken + taken  
}  
}
```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][k+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as  $dp[n][k+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mod = (int)1e9+7;

int countPartitionsUtil(int ind, int target, vector<int>& arr, vector<vector<int>> &dp){

    if(ind == 0){
        if(target==0 && arr[0]==0)
            return 2;
        if(target==0 || target == arr[0])
            return 1;
        return 0;
    }

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    int notTaken = countPartitionsUtil(ind-1,target,arr,dp);

    int taken = 0;
    if(arr[ind]<=target)
        taken = countPartitionsUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= (notTaken + taken)%mod;
}

int countPartitions(int d, vector<int>& arr){
    int n = arr.size();
    int totSum = 0;
    for(int i=0; i<arr.size();i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-d<0) return 0;
    if((totSum-d)%2==1) return 0;

    int s2 = (totSum-d)/2;

    vector<vector<int>> dp(n, vector<int>(s2+1,-1));
    return countPartitionsUtil(n-1,s2,arr,dp);
}

int main() {

    vector<int> arr = {5,2,6,4};
    int d=3;

    cout<<"The number of subsets found are " <<countPartitions(d,arr);
}

```

## Output:

The number of subsets found are 1

### Time Complexity: $O(N^*K)$

Reason: There are  $N^*K$  states therefore at max ' $N^*K$ ' new problems will be solved.

### Space Complexity: $O(N^*K) + O(N)$

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (  $O(N^*K)$ ).

### Steps to convert Recursive Solution to Tabulation one.

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

Eg: [4,2,3,7,...,23]

Target: 11

ind	target	0	1	2	3	4	...	K
0	target	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0	0
N-1	0	0	0	0	0	0	0	0

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as 1.

Eg: [4,2,3,7,...,23]

Target: 11

target →

target ind \	0	1	2	3	4	...	K
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = 1$ , ( $dp[0][arr[0]]$  means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > target$ , so we first check it: if( $arr[0] \leq target$ ) then set  $dp[0][arr[0]] = 1$ .

Eg: [4,2,3,7,...,23]

Target: 11

target →

$arr[0] = 4$        $dp[0][4]$

target ind \	0	1	2	3	4	...	K
0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

- After that, we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return  $dp[n-1][k]$  as our answer.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mod =(int)1e9+7;

int findWays(vector<int> &num, int tar){
    int n = num.size();

    vector<vector<int>> dp(n, vector<int>(tar+1,0));

    if(num[0] == 0) dp[0][0] =2; // 2 cases -pick and not pick
    else dp[0][0] = 1; // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) dp[0][num[0]] = 1; // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        for(int target= 0; target<=tar; target++){

            int notTaken = dp[ind-1][target];

            int taken = 0;
            if(num[ind]<=target)
                taken = dp[ind-1][target-num[ind]];

            dp[ind][target]= (notTaken + taken)%mod;
        }
    }
    return dp[n-1][tar];
}

int countPartitions(int n, int d, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-d <0 || (totSum-d)%2 ) return 0;

    return findWays(arr,(totSum-d)/2);
}

int main() {

    vector<int> arr = {5,2,6,4};
    int n = arr.size();
    int d=3;

    cout<<"The number of subsets found are " <<countPartitions(n,d,arr);
}

```

### **Output:**

The number of subsets found are 1

## **Time Complexity: O(N\*K)**

Reason: There are two nested loops

## **Space Complexity: O(N\*K)**

Reason: We are using an external array of size 'N\*K'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] + dp[ind-1][target - arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element is true according to our base condition.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mod = (int)1e9+7;

int findWays(vector<int> &num, int tar){
    int n = num.size();

    vector<int> prev(tar+1, 0);

    if(num[0] == 0) prev[0] = 2; // 2 cases -pick and not pick
    else prev[0] = 1; // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) prev[num[0]] = 1; // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        vector<int> cur(tar+1, 0);
        for(int target= 0; target<=tar; target++){
            int notTaken = prev[target];

            int taken = 0;
            if(num[ind]<=target)
                taken = prev[target-num[ind]];

            cur[target]= (notTaken + taken)%mod;
        }
        prev = cur;
    }
    return prev[tar];
}

int countPartitions(int n, int d, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-d <0 || (totSum-d)%2 ) return 0;

    return findWays(arr,(totSum-d)/2);
}

int main() {

    vector<int> arr = {5,2,6,4};
    int n = arr.size();
    int d=3;

    cout<<"The number of subsets found are " <<countPartitions(n,d,arr);
}

```

### **Output:**

The number of subsets found are 1

**Time Complexity: O(N\*K)**

Reason: There are three nested loops

**Space Complexity: O(K)**

Reason: We are using an external array of size 'K+1' to store only one row.

# 0/1 Knapsack (DP – 19)

 [takeuforward.org/data-structure/0-1-knapsack-dp-19](https://takeuforward.org/data-structure/0-1-knapsack-dp-19)

February 15, 2022

**Problem Statement:** 0/1 Knapsack

**Problem Link:** [0/1 Knapsack](#)

A thief wants to rob a store. He is carrying a bag of capacity  $W$ . The store has ‘ $n$ ’ items. Its weight is given by the ‘wt’ array and its value by the ‘val’ array. He can either include an item in its knapsack or exclude it but can’t partially have it as a fraction. We need to find the maximum value of items that the thief can steal.

**Explanation:**

		$N = 4$	$W = 5$
Wt:		1	2
		4	5
Val:		5	4
		8	6
<b>Answer: 13</b>			
The thief can select items valued 8 and 5 with weight 4 and 1 respectively.			

**Disclaimer:** Don’t jump directly to the solution, try it out yourself first.

Pre-req: [Count Subsets with Sum K](#)

**Solution :**

**Why a Greedy Solution doesn’t work?**

The first approach that comes to our mind is greedy. A greedy solution will fail in this problem because there is no ‘uniformity’ in data. While selecting a local better choice we may choose an item that will in long term give less value.

Let us understand this with help of an example:

A Greedy solution will be to take the most valuable item first, so we will take an item on index 2, with a value of 60, and put it in the knapsack.

Now the remaining capacity of the knapsack will be 1. Therefore we cannot add any other item. So a greedy **solution** gives us the answer **60**.

Now we can clearly see that a **non-greedy solution** of taking the first two items will give us the value of **70** (**30+40**) in the given capacity of the knapsack.

<b>N = 3</b>	<input type="text"/>
<b>W = 6</b>	
<b>Wt:</b>	3    2    5
<b>Val:</b>	30    40    60

As the greedy approach doesn't work, we will try to generate all possible combinations using **recursion** and select the combination which gives us the **maximum** value in the given constraints.

#### **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given 'n' items. Their weight is represented by the 'wt' array and value by the 'val' array. So clearly one parameter will be 'ind', i.e index up to which the array items are being considered.

There is one more parameter "W". We need the capacity of the knapsack to decide whether we can pick an array item or not in the knapsack.

So, we can say that initially, we need to find  $f(n-1, W)$  where  $W$  is the overall capacity given to us.  $f(n-1, W)$  means we are finding the maximum value of items that the thief can steal from items with index 0 to  $n-1$  capacity  $W$  of the knapsack.

**f(ind,W) ->Maximum value of items from index 0 to ind, with capacity of knapsack W**

#### **Base Cases:**

If  $\text{ind}==0$ , it means we are at the first item, so in that case we will check whether this item's weight is less than or equal to the current capacity  $W$ , if it is, we simply return its value ( $\text{val}[0]$ ) else we return 0.

```
f(ind,W) {  
    if( ind==0 ) {  
        if( wt[0]<=W )  
            return val[0];  
    }  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index item. If we exclude the current item, the capacity of the bag will not be affected and the value added will be 0 for the current item. So we will call the recursive function  $f(\text{ind}-1, W)$
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current item to the knapsack. As we have included the item, the capacity of the knapsack will be updated to  $W - \text{wt}[\text{ind}]$  and the current item's value ( $\text{val}[\text{ind}]$ ) will also be added to the further recursive call answer. We will make a recursive call to  $f(\text{ind}-1, W - \text{wt}[\text{ind}])$ .

**Note:** We will consider the current item in the subsequence only when the current element's weight is less than or equal to the capacity ' $W$ ' of the knapsack, if it isn't we will not be considering it.

```

f(ind,W) {
    if( ind==0) {
        if( wt[0]<=W)
            return val[0]
        return 0
    }

    notTake = 0 + f(ind-1, W)

    take = INT_MIN

    if(wt[ind]<=W)
        take = val[ind] + f(ind-1, W - wt[ind])
    }
}

```

### **Step 3: Return the maximum of take and notTake**

As we have to return the maximum amount of value, we will return the max of take and notTake as our answer.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,W) {

    if( ind==0) {
        if( wt[0]<=W)
            return val[0]
        return 0
    }

    notTake = 0 + f(ind-1, W)

    take = INT_MIN

    if(wt[ind]<=W)
        take = val[ind] + f(ind-1, W - wt[ind])

    return max(take, notTake)

}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][W+1]$ . The size of the input array is ‘N’, so the index will always lie between ‘0’ and ‘n-1’. The capacity can take any value between ‘0’ and ‘W’. Therefore we take the dp array as  $dp[n][W+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int knapsackUtil(vector<int>& wt, vector<int>& val, int ind, int W,
vector<vector<int>>& dp){

    if(ind == 0){
        if(wt[0] <=W) return val[0];
        else return 0;
    }

    if(dp[ind][W]!=-1)
        return dp[ind][W];

    int notTaken = 0 + knapsackUtil(wt,val,ind-1,W,dp);

    int taken = INT_MIN;
    if(wt[ind] <= W)
        taken = val[ind] + knapsackUtil(wt,val,ind-1,W-wt[ind],dp);

    return dp[ind][W] = max(notTaken,taken);
}

int knapsack(vector<int>& wt, vector<int>& val, int n, int W){

    vector<vector<int>> dp(n, vector<int>(W+1,-1));
    return knapsackUtil(wt, val, n-1, W, dp);
}

int main() {

    vector<int> wt = {1,2,4,5};
    vector<int> val = {5,4,8,6};
    int W=5;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<knapsack(wt,val,n,W);
}

```

### **Output:**

The Maximum value of items, thief can steal is 13

### **Time Complexity: O(N\*W)**

Reason: There are  $N \times W$  states therefore at max ' $N \times W$ ' new problems will be solved.

### **Space Complexity: O(N\*W) + O(N)**

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array ( $O(N \times W)$ ).

### **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

First, we need to initialize the base conditions of the recursive solution.

- At  $\text{ind}==0$ , we are considering the first element, if the capacity of the knapsack is greater than the weight of the first item, we return  $\text{val}[0]$  as answer. We will achieve this using a for loop.
- Next, we are done for the first row, so our ‘ $\text{ind}$ ’ variable will move from 1 to  $n-1$ , whereas our ‘ $\text{cap}$ ’ variable will move from 0 to ‘ $W$ ’. We will set the nested loops to traverse the dp array.
- Inside the nested loops we will apply the recursive logic to find the answer of the cell.
- When the nested loop execution has ended, we will return  $\text{dp}[n-1][W]$  as our answer.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int knapsack(vector<int>& wt, vector<int>& val, int n, int W){

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    //Base Condition

    for(int i=wt[0]; i<=W; i++){
        dp[0][i] = val[0];
    }

    for(int ind =1; ind<n; ind++){
        for(int cap=0; cap<=W; cap++){

            int notTaken = 0 + dp[ind-1][cap];

            int taken = INT_MIN;
            if(wt[ind] <= cap)
                taken = val[ind] + dp[ind-1][cap - wt[ind]];

            dp[ind][cap] = max(notTaken, taken);
        }
    }

    return dp[n-1][W];
}

int main() {

    vector<int> wt = {1,2,4,5};
    vector<int> val = {5,4,8,6};
    int W=5;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<knapsack(wt,val,n,W);
}

```

### **Output:**

The Maximum value of items, thief can steal is 13

### **Time Complexity: O(N\*W)**

Reason: There are two nested loops

### **Space Complexity: O(N\*W)**

Reason: We are using an external array of size ‘N\*W’. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][cap] = \max(dp[ind-1][cap], dp[ind-1][cap-wt[ind]])$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

We will be space optimizing this solution using **only one row**.

### Intuition:

If we closely observe, we fill in the following manner in two-row space optimization:

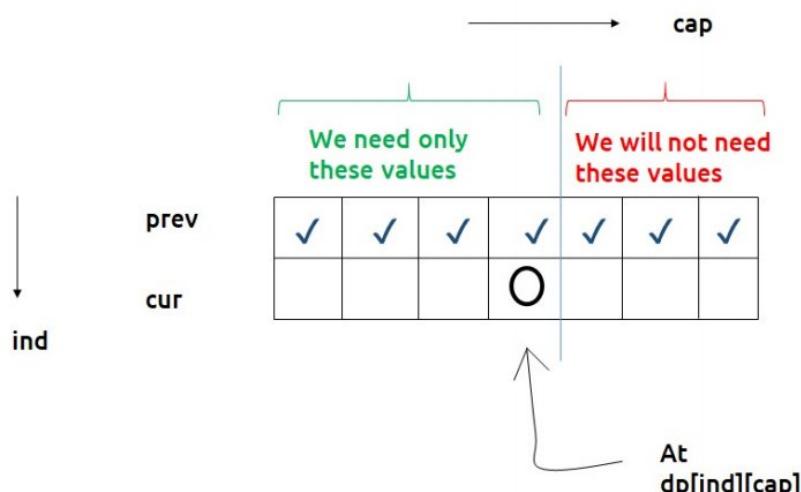
We will initialize the first row and then using its values we will fill the next row.

## Space Optimization

prev	✓	✓	✓	✓	✓	✓	✓
cur							

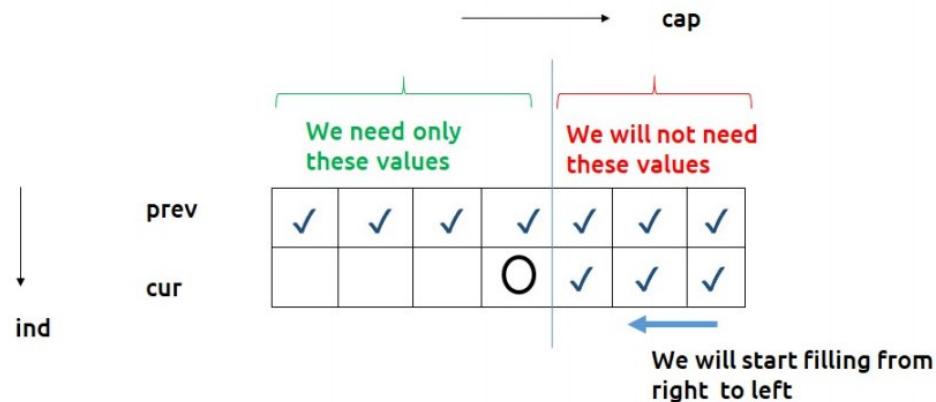
If we clearly see the values required:  $dp[ind-1][cap]$  and  $dp[ind-1][cap - wt[ind]]$ , we can say that if we are at a column cap, we will only require the values shown in the green region and none in the red region shown in the below image ( because  $cap - wt[ind]$  will always be less than the cap).

## Space Optimization



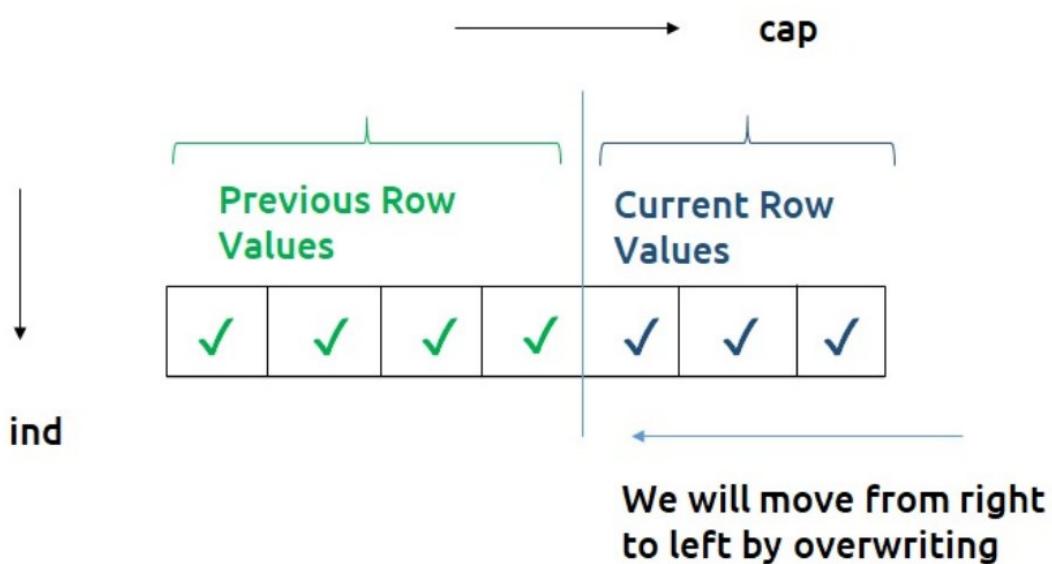
As we don't want values from the right, we can start filling this new row from the right rather than the left.

### Space Optimization



Now here is the catch, if we are filling from the right and at any time we need the previous row's value of the leftward columns only, why do we need to have two rows in the first place? We can use a single row and **overwrite** the new computed values on itself in order to store it.

### Space Optimization In a single row



### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int knapsack(vector<int>& wt, vector<int>& val, int n, int W){

    vector<int> prev(W+1, 0);

    //Base Condition

    for(int i=wt[0]; i<=W; i++){
        prev[i] = val[0];
    }

    for(int ind =1; ind<n; ind++){
        for(int cap=W; cap>=0; cap--){

            int notTaken = 0 + prev[cap];

            int taken = INT_MIN;
            if(wt[ind] <= cap)
                taken = val[ind] + prev[cap - wt[ind]];

            prev[cap] = max(notTaken, taken);
        }
    }

    return prev[W];
}

int main() {

    vector<int> wt = {1,2,4,5};
    vector<int> val = {5,4,8,6};
    int W=5;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<knapsack(wt,val,n,W);
}

```

### **Output:**

The Maximum value of items, thief can steal is 13

**Time Complexity: O(N\*W)**

Reason: There are two nested loops.

**Space Complexity: O(W)**

Reason: We are using an external array of size ‘W+1’ to store only one row.

# Minimum Coins (DP – 20)

 [takeuforward.org/data-structure/minimum-coins-dp-20](https://takeuforward.org/data-structure/minimum-coins-dp-20)

February 23, 2022

## Problem Statement: Minimum Coins

### Problem Link: Minimum Coins

We are given a target sum of 'X' and 'N' distinct numbers denoting the coin denominations. We need to tell the minimum number of coins required to reach the target sum. We can pick a coin denomination for any number of times we want.

**Example:**

**Target: 7**

arr :      

1	2	3
---	---	---

**Solution: 3**

**We will take 3 coins {3,3,1}**

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: Subset Sum equal to the target

## Solution :

### Why a Greedy Solution doesn't work?

The first approach that comes to our mind is greedy. A greedy solution will fail in this problem because there is no '**uniformity**' in data. While selecting a local better choice we may choose an item that will in the long term give less value.

Let us understand this with help of an example:

**Target: 11**

**arr:**

9	6	5	1
---	---	---	---

A Greedy solution will be to take the highest denomination coin first, so we will take an item on index 0, with a value of 9. Now the remaining target sum will be 2. Therefore we can only consider coins with a value of 1. We will take 2 coins of value 1 to meet the target. So a greedy **solution** gives us the answer **3 {9,1,1}**.

Now we can clearly see that a **non-greedy solution** of taking **2** coins valued **6 and 5** will give us a better option. So we can say that the greedy solution doesn't work for this problem.

As the greedy approach doesn't work, we will try to generate all possible combinations using **recursion** and select the combination which gives us the **minimum** number of coins.

#### **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given 'n' distinct numbers. Their denomination is represented by the 'arr' array. So clearly one parameter will be 'ind', i.e index up to which the array items are being considered.

There is one more parameter "T". We need to know the given target that we want to achieve.

So, we can say that initially, we need to find  $f(n-1, T)$  where T is the initial target given to us.  $f(n-1, T)$  means we are finding the minimum number of coins required to form the target sum by considering coins from index 0 to n-1.

**$f(ind, T) \rightarrow$  Minimum coins required to form target T, coins given by arr[0...ind]**

### Base Cases:

If  $\text{ind}==0$ , it means we are at the first item, so in that case, the following cases can arise:

**arr[0] = 4 and T = 12**

In such a case where the target is divisible by the coin element, we will return  $T \% \text{arr}[0]$ .

**arr[0] = 4 and T=1 , arr[0]=3 T=10**

In all other cases, we will not be able to form a solution, so we will return a big number like **1e9**

```
f(ind,T) {  
    if( ind==0 ) {  
        if( T%arr[0]==0 )  
            return T/arr[0];  
        else  
            return 1e9;  
    }  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)". There will be a slight change for this question which is discussed below.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index coin. If we exclude the current coin, the target sum will not be affected and the number of coins added to the solution will be **0**. So we will call the recursive function **f(ind-1,T)**
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current coin. As we have included the coin, the target sum will be updated to  $T - \text{arr}[ind]$  and we have considered **1** coin to our solution.

Now here is the catch, as there is an unlimited supply of coins, we want to again form a solution with the same coin value. So we **will not** recursively call for  $f(ind-1, T - arr[ind])$  rather we will stay at that index only and call for  $f(ind, T - arr[ind])$  to find the answer.

**Note:** We will consider the current coin only when its denomination value ( $arr[ind]$ ) is less than or equal to the target  $T$ .

```
f(ind,T) {  
  
    if( ind==0 ) {  
        if( T%arr[0]==0 )  
            return T/arr[0]  
        else  
            return 1e9  
  
    notTake = 0 + f(ind-1, T)  
  
    take = 1e9  
  
    if( arr[ind] <= T )  
        take = 1 + f(ind, T - arr[ind])  
    }  
}
```

### Step 3: Return the minimum of take and notTake

As we have to return the minimum number of coins, we will return the minimum of take and notTake as our answer.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,T) {

    if( ind==0) {
        if( T%arr[0]==0 )
            return T/arr[0]
        else
            return 1e9

    notTake = 0 + f(ind-1, T)

    take = 1e9

    if( arr[ind] <= T)
        take = 1 + f(ind, T - arr[ind])

    return min(take,notTake)
}

```

### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][T+1]$ . The size of the input array is ‘N’, so the index will always lie between ‘0’ and ‘n-1’. The target Sum can take any value between ‘0’ and ‘T’. Therefore we take the dp array as  $dp[n][T+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,T)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][T] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### Code:

- C++Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minimumElementsUtil(vector<int>& arr, int ind, int T, vector<vector<int>>& dp)
{
    if(ind == 0){
        if(T%arr[0] == 0) return T/arr[0];
        else return 1e9;
    }

    if(dp[ind][T]!=-1)
        return dp[ind][T];

    int notTaken = 0 + minimumElementsUtil(arr,ind-1,T,dp);

    int taken = 1e9;
    if(arr[ind] <= T)
        taken = 1 + minimumElementsUtil(arr,ind,T-arr[ind],dp);

    return dp[ind][T] = min(notTaken,taken);
}

int minimumElements(vector<int>& arr, int T){
    int n= arr.size();

    vector<vector<int>> dp(n,vector<int>(T+1,-1));
    int ans = minimumElementsUtil(arr, n-1, T, dp);
    if(ans >= 1e9) return -1;
    return ans;
}

int main() {
    vector<int> arr ={1,2,3};
    int T=7;

    cout<<"The minimum number of coins required to form the target sum is "
    <<minimumElements(arr,T);
}

```

### **Output:**

The minimum number of coins required to form the target sum is 3

### **Time Complexity: O(N\*T)**

Reason: There are N\*T states therefore at max 'N\*T' new problems will be solved.

### **Space Complexity: O(N\*T) + O(N)**

Reason: We are using a recursion stack space(O(N)) and a 2D array ( O(N\*T)).

## **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

First, we need to initialize the base conditions of the recursive solution.

- At  $\text{ind}==0$ , we are considering the first element, if  $T\%arr[0] == 0$ , we initialize it to  $T/arr[0]$  else we initialize it to  $1e9$ .
- Next, we are done for the first row, so our ‘ind’ variable will move from 1 to  $n-1$ , whereas our ‘target’ variable will move from 0 to ‘T’. We will set the nested loops to traverse the dp array.
- Inside the nested loops we will apply the recursive logic to find the answer of each cell.
- When the nested loop execution has ended, we will return  $dp[n-1][T]$  as our answer.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minimumElements(vector<int>& arr, int T){

    int n= arr.size();

    vector<vector<int>> dp(n, vector<int>(T+1, 0));

    for(int i=0; i<=T; i++){
        if(i%arr[0] == 0)
            dp[0][i] = i/arr[0];
        else dp[0][i] = 1e9;
    }

    for(int ind = 1; ind<n; ind++){
        for(int target = 0; target<=T; target++){

            int notTake = 0 + dp[ind-1][target];
            int take = 1e9;
            if(arr[ind]<=target)
                take = 1 + dp[ind][target - arr[ind]];

            dp[ind][target] = min(notTake, take);
        }
    }

    int ans = dp[n-1][T];
    if(ans >=1e9) return -1;
    return ans;
}

int main() {

    vector<int> arr ={1,2,3};
    int T=7;

    cout<<"The minimum number of coins required to form the target sum is "
    <<minimumElements(arr,T);
}

```

### **Output:**

The minimum number of coins required to form the target sum is 3

### **Time Complexity: O(N\*T)**

Reason: There are two nested loops

### **Space Complexity: O(N\*T)**

Reason: We are using an external array of size 'N\*T'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] + dp[ind-1][target - arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** We first need to initialize the first row as we had done in the tabulation approach.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minimumElements(vector<int>& arr, int T){

    int n= arr.size();

    vector<int> prev(T+1, 0), cur(T+1, 0);

    for(int i=0; i<=T; i++){
        if(i%arr[0] == 0)
            prev[i] = i/arr[0];
        else prev[i] = 1e9;
    }

    for(int ind = 1; ind<n; ind++){
        for(int target = 0; target<=T; target++){

            int notTake = 0 + prev[target];
            int take = 1e9;
            if(arr[ind]<=target)
                take = 1 + cur[target - arr[ind]];

            cur[target] = min(notTake, take);
        }
        prev = cur;
    }

    int ans = prev[T];
    if(ans >=1e9) return -1;
    return ans;
}

int main() {

    vector<int> arr ={1,2,3};
    int T=7;

    cout<<"The minimum number of coins required to form the target sum is "
    <<minimumElements(arr,T);
}

```

### **Output:**

The minimum number of coins required to form the target sum is 3

### **Time Complexity: O(N\*T)**

Reason: There are two nested loops.

### **Space Complexity: O(T)**

Reason: We are using two external arrays of size ‘T+1’.



# Target Sum (DP – 21)

---

 [takeuforward.org/data-structure/target-sum-dp-21](https://takeuforward.org/data-structure/target-sum-dp-21)

February 27, 2022

**Problem Link:** [Target Sum](#)

## Problem Description:

We are given an array ‘ARR’ of size ‘N’ and a number ‘Target’. Our task is to build an expression from the given array where we can place a ‘+’ or ‘-’ sign in front of an integer. We want to place a sign in front of every integer of the array and get our required target. We need to count the number of ways in which we can achieve our required target.

**Example:**

Arr :	1	2	3	1
-------	---	---	---	---

Target: 3

---

$$+1 -2 +3 +1 = 3$$

$$+1 -2 +3 +1 = 3$$

There are **2** ways

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: [Count Partitions with Given Difference \(DP – 18\)](#)

## Solution :

---

The first approach that comes to our mind is to generate all subsequences and try both options of placing ‘-’ and ‘+’ signs and count the expression if it evaluates the answer.

This surely will give us the answer but can we try something familiar to the previous problems we have solved?

The answer is **yes!** We can use the concept we studied in the following article [Count Partitions with Given Difference \(DP – 18\)](#).

The following insights will help us to understand intuition better:

If we think deeper, we can say that the given ‘target’ can be expressed as addition of two integers (say S<sub>1</sub> and S<sub>2</sub>).

$$S_1 + S_2 = \text{target} \quad - (\text{i})$$

Now, from where will this S<sub>1</sub> and S<sub>2</sub> come? If we are given the array as [a,b,c,d,e], we want to place ‘+’ or ‘-’ signs in front of every array element and then add it. One example is :

$$+a - b - c + d + e \text{ which can be written as } (+a + d + e) + (-b - c).$$

Therefore, we can say that **S<sub>1</sub>=(+a+d+e)** and **S<sub>2</sub>=(-b-c)** for this example.

If we calculate the total sum of elements of the array (say totSum), we can say that,

$$S_1 = \text{totSum} - S_2 \quad - (\text{ii})$$

Now solving for equations (i) and (ii), we can say that

$$S_2 = (\text{totSum} - \text{target})/2 \quad - (\text{iv})$$

Therefore this question is modified to “Count Number of subsets with sum (totSum – target)/2”. This is exactly what we had discussed in the article [Count Subsets with Sum K](#).

### Edge Cases:

The following edge cases need to be handled:

- As the array elements are positive integers including zero, we don’t want to find the case when S<sub>2</sub> is negative or we can say that totSum is lesser than D, therefore if totSum < target, we simply return 0.
- S<sub>2</sub> can’t be a fraction, as all elements are integers, therefore if totSum – target is odd, we can return 0.

From here on we will discuss the approach to “Count Subsets with Sum K” with the required modifications. Moreover, as the array elements can also contain 0, we will handle it as discussed in part-1 of this article.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

### **Step 1:** Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find( $n-1$ , target) which means that we are counting the number of subsequences in the array from index 0 to  $n-1$ , whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

**f(ind,target) -> Count Number of subsequences that exists in the Array from index 0 to ind, whose sum is equal to target**

### **Base Cases:**

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return 1.
- If ind==0, it means we are at the first element, so we need to return arr[ind]==target. If the element is equal to the target we return 1 else we return 0.

```
f(ind,target) {  
    if(target==0) return true  
    if( ind==0) return arr[ind] == target  
}
```

### **Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video “[Recursion on Subsequences](#)”.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to f(ind-1,target).

- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included  $\text{arr}[\text{ind}]$ , the updated target which we need to find in the rest of the array will be  $\text{target} - \text{arr}[\text{ind}]$ . Therefore, we will call  $f(\text{ind}-1, \text{target} - \text{arr}[\text{ind}])$ .

**Note:** We will consider the current element in the subsequence only when the current element is less than or equal to the target.

```

f(ind,target) {
    if(target==0) return 1
    if(ind==0) return arr[ind] == target

    notTaken = f(ind-1,target)

    taken = 0
    if( arr[ind]<=target)
        taken = f(ind-1,target - arr[ind])

}

```

### Step 3: Return sum of taken and notTaken

As we have to return the total count of subsets with the target sum, we will return the sum of taken and notTaken from our recursive call.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,target) {
    if(target==0) return 1
    if( ind==0) return arr[ind] == target

    notTaken = f(ind-1,target)

    taken = 0
    if( arr[ind]<=target)
        taken = f(ind-1,target - arr[ind])

    return notTaken + taken
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][k+1]$ . The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as  $dp[n][k+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int countPartitionsUtil(int ind, int target, vector<int>& arr, vector<vector<int>>
&dp){

    if(ind == 0){
        if(target==0 && arr[0]==0)
            return 2;
        if(target==0 || target == arr[0])
            return 1;
        return 0;
    }

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    int notTaken = countPartitionsUtil(ind-1,target,arr,dp);

    int taken = 0;
    if(arr[ind]<=target)
        taken = countPartitionsUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= (notTaken + taken);
}

int targetSum(int n,int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<arr.size();i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-target<0) return 0;
    if((totSum-target)%2==1) return 0;

    int s2 = (totSum-target)/2;

    vector<vector<int>> dp(n,vector<int>(s2+1,-1));
    return countPartitionsUtil(n-1,s2,arr,dp);
}

int main() {

    vector<int> arr = {1,2,3,1};
    int target=3;

    int n = arr.size();
    cout<<"The number of ways found is " <<targetSum(n,target,arr);
}

```

### **Output:**

The number of ways found is 2

## Time Complexity: $O(N*K)$

Reason: There are  $N*K$  states therefore at max ' $N*K$ ' new problems will be solved.

## Space Complexity: $O(N*K) + O(N)$

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (  $O(N*K)$ ).

### Steps to convert Recursive Solution to Tabulation one.

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

Eg: [4,2,3,7,...,23]

Target: 11

target \ ind	0	1	2	3	4	...	K
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0
N-1	0	0	0	0	0	0	0

First, we need to initialize the base conditions of the recursive solution.

If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as 1.

Eg: [4,2,3,7,...,23]

Target: 11

target →

target ind	0	1	2	3	4	...	K
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

The first row  $dp[0][]$  indicates that only the first element of the array is considered, therefore for the target value equal to  $arr[0]$ , only cell with that target will be true, so explicitly set  $dp[0][arr[0]] = 1$ , ( $dp[0][arr[0]]$  means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that  $arr[0] > target$ , so we first check it: if( $arr[0] \leq target$ ) then set  $dp[0][arr[0]] = 1$ .

Eg: [4,2,3,7,...,23]

Target: 11

target →

$arr[0] = 4$        $dp[0][4]$

target ind	0	1	2	3	4	...	K
0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0
.	1	0	0	0	0	0	0
N-1	1	0	0	0	0	0	0

- After that, we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return  $dp[n-1][k]$  as our answer.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mod = (int)1e9+7;

int findWays(vector<int> &num, int tar){
    int n = num.size();

    vector<vector<int>> dp(n, vector<int>(tar+1, 0));

    if(num[0] == 0) dp[0][0] = 2; // 2 cases -pick and not pick
    else dp[0][0] = 1; // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) dp[0][num[0]] = 1; // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        for(int target= 0; target<=tar; target++){

            int notTaken = dp[ind-1][target];

            int taken = 0;
            if(num[ind]<=target)
                taken = dp[ind-1][target-num[ind]];

            dp[ind][target]= (notTaken + taken)%mod;
        }
    }
    return dp[n-1][tar];
}

int targetSum(int n, int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-target <0 || (totSum-target)%2 ) return 0;

    return findWays(arr,(totSum-target)/2);
}

int main() {

    vector<int> arr = {1,2,3,1};
    int target=3;

    int n = arr.size();
    cout<<"The number of ways found is " <<targetSum(n,target,arr);
}

```

### **Output:**

The number of ways found is 2

### **Time Complexity: O(N\*K)**

Reason: There are two nested loops

### **Space Complexity: O(N\*K)**

Reason: We are using an external array of size 'N\*K'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target] + dp[ind-1][target - arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** Whenever we create a new row ( say cur), we need to explicitly set its first element as true according to our base condition.

#### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int mod = (int)1e9+7;

int findWays(vector<int> &num, int tar){
    int n = num.size();

    vector<int> prev(tar+1, 0);

    if(num[0] == 0) prev[0] = 2; // 2 cases -pick and not pick
    else prev[0] = 1; // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) prev[num[0]] = 1; // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        vector<int> cur(tar+1, 0);
        for(int target= 0; target<=tar; target++){
            int notTaken = prev[target];

            int taken = 0;
            if(num[ind]<=target)
                taken = prev[target-num[ind]];

            cur[target]= (notTaken + taken)%mod;
        }
        prev = cur;
    }
    return prev[tar];
}

int targetSum(int n, int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-target <0 || (totSum-target)%2 ) return 0;

    return findWays(arr,(totSum-target)/2);
}
}

int main() {

    vector<int> arr = {1,2,3,1};
    int n = arr.size();
    int target=3;

    cout<<"The number of subsets found is " <<targetSum(n,target,arr);
}

```

### **Output:**

The number of ways found is 2

**Time Complexity: O(N\*K)**

Reason: There are three nested loops

**Space Complexity: O(K)**

Reason: We are using an external array of size 'K+1' to store only one row.

## Coin Change 2 (DP – 22)

 [takeuforward.org/data-structure/coin-change-2-dp-22](https://takeuforward.org/data-structure/coin-change-2-dp-22)

February 27, 2022

### Problem Link: Ways to Make a Coin Change

We are given an array Arr with N distinct coins and a target. We have an infinite supply of each coin denomination. We need to find the number of ways we sum up the coin values to give us the target.

Each coin can be used any number of times.

**Example:**

N=3	Arr :	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>2</td><td>3</td></tr></table>	1	2	3	Target: 4
1	2	3				
$\{1+1+1+1\} = 4$		$\{1+1+2\} = 4$				
$\{1+3\} = 4$		$\{2+2\} = 4$				
There are <b>4</b> ways						

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: 0/1 Knapsack

### Solution :

#### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in Dynamic Programming Introduction.

**Step 1:** Express the problem in terms of indexes.

We are given 'n' coins. Their denomination value is given by the array 'arr'. So clearly one parameter will be 'ind', i.e index up to which the array items are being considered.

There is one more parameter, the given target value “T” which we want to achieve so that while generating subsequences, we can decide whether we want to include a particular coin or not.

So, we can say that initially, we need to find  $f(n-1, T)$  where T is the initial target given to us in the question.  $f(n-1, T)$  means we are finding the total number of ways to form the target T by considering coins from index 0 to index n-1 of the arr array.

**$f(ind, T) > \text{Total number of ways, by considering coins from index 0 to } ind-1, \text{ to get target } T.$**

### Base Cases:

If  $ind==0$ , it means we are at the first item so we have only one coin denomination, therefore the following two cases can arise:

**T is divisible by arr[0] (eg: arr[0] = 4 and T = 12)**

In such a case where the target is divisible by the coin element value, we will return 1 as we will be able to form the target.

**T is not divisible by arr[0] (eg: arr[0] = 4 and T = 7)**

In all other cases, we will not be able to form the target, so we will return 0.

```
f(ind, T) {  
    if( ind==0){  
        return (T%arr[0]==0)  
    }  
}
```

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video “[Recursion on Subsequences](#)”.

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index coin. If we exclude the current coin, the target sum will not be affected. So we will call the recursive function **f(ind-1,T)** to find the remaining answer.
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current coin. As we have included the coin, the target sum will be updated to T-arr[ind].

Now here is the catch, as there is an unlimited supply of coins, we want to again form a solution with the same coin value. So we **will not** recursively call for f(ind-1, T-arr[ind]) rather we will stay at that index only and call for f(ind, **T-arr[ind]**) to find the answer.

**Note:** We will consider the current coin only when its denomination value (arr[ind]) is less than or equal to the target T.

```

f(ind,T) {
    if( ind==0){
        return (T%arr[0]==0)
    }

    notTake = f(ind-1,T)

    take = 0

    if(arr[ind]<=T)
        take = f(ind,T-arr[ind])
}

```

### Step 3: Return the sum of take and notTake

As we have to return the total number of ways we can form the target, we will return the sum of notTake and take as our answer.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,T) {
    if( ind==0){
        return (T%arr[0]==0)
    }

    notTake = f(ind-1,T)

    take = 0

    if(arr[ind]<=T)

        take = f(ind,T-arr[ind])

    return notTake + take
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution to the following steps will be taken:

1. Create a dp array of size  $[n][T+1]$ . The size of the input array is 'N', so the index will always lie between '0' and 'n-1'. The target can take any value between '0' and 'T'. Therefore we take the dp array as  $dp[n][T+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChangeUtil(vector<int>& arr, int ind, int T, vector<vector<long>>& dp){

    if(ind == 0){
        return (T%arr[0]==0);
    }

    if(dp[ind][T]!=-1)
        return dp[ind][T];

    long notTaken = countWaysToMakeChangeUtil(arr,ind-1,T,dp);

    long taken = 0;
    if(arr[ind] <= T)
        taken = countWaysToMakeChangeUtil(arr,ind,T-arr[ind],dp);

    return dp[ind][T] = notTaken + taken;
}

long countWaysToMakeChange(vector<int>& arr, int n, int T){

    vector<vector<long>> dp(n, vector<long>(T+1, -1));
    return countWaysToMakeChangeUtil(arr,n-1, T, dp);
}

int main() {

    vector<int> arr ={1,2,3};
    int target=4;

    int n =arr.size();

    cout<<"The total number of ways is " <<countWaysToMakeChange(arr,n,target);
}

```

### **Output:**

The total number of ways is 4

### **Time Complexity: O(N\*T)**

Reason: There are N\*W states therefore at max 'N\*T' new problems will be solved.

### **Space Complexity: O(N\*T) + O(N)**

Reason: We are using a recursion stack space(O(N)) and a 2D array ( O(N\*T)).

### **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

First, we need to initialize the base conditions of the recursive solution.

- At  $\text{ind}==0$ , we are considering the first element, if the target value is divisible by the first coin's value, we set the cell's value as 1 else 0.
- Next, we are done for the first row, so our 'ind' variable will move from 1 to  $n-1$ , whereas our 'target' variable will move from 0 to ' $T$ '. We will set the nested loops to traverse the dp array.
- Inside the nested loops we will apply the recursive logic to find the answer of the cell.
- When the nested loop execution has ended, we will return  $\text{dp}[n-1][T]$  as our answer.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChange(vector<int>& arr, int n, int T){

    vector<vector<long>> dp(n, vector<long>(T+1, 0));

    //Initializing base condition
    for(int i=0;i<=T;i++){
        if(i%arr[0]==0)
            dp[0][i]=1;
        // Else condition is automatically fulfilled,
        // as dp array is initialized to zero
    }

    for(int ind=1; ind<n;ind++){
        for(int target=0;target<=T;target++){
            long notTaken = dp[ind-1][target];

            long taken = 0;
            if(arr[ind]<=target)
                taken = dp[ind][target-arr[ind]];

            dp[ind][target] = notTaken + taken;
        }
    }

    return dp[n-1][T];
}

int main() {

    vector<int> arr ={1,2,3};
    int target=4;

    int n =arr.size();

    cout<<"The total number of ways is " <<countWaysToMakeChange(arr,n,target);
}

```

### **Output:**

The total number of ways is 4

### **Time Complexity: O(N\*T)**

Reason: There are two nested loops

### **Space Complexity: O(N\*T)**

Reason: We are using an external array of size 'N\*T'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][target] = dp[ind-1][target], dp[ind-1][target - arr[ind]]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

**Note:** We first need to initialize the first row as we had done in the tabulation approach.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChange(vector<int>& arr, int n, int T){

    vector<long> prev(T+1, 0);

    //Initializing base condition
    for(int i=0;i<=T;i++){
        if(i%arr[0]==0)
            prev[i]=1;
        // Else condition is automatically fulfilled,
        // as prev array is initialized to zero
    }

    for(int ind=1; ind<n;ind++){
        vector<long> cur(T+1, 0);
        for(int target=0;target<=T;target++){
            long notTaken = prev[target];

            long taken = 0;
            if(arr[ind]<=target)
                taken = cur[target-arr[ind]];

            cur[target] = notTaken + taken;
        }
        prev = cur;
    }

    return prev[T];
}

int main() {

    vector<int> arr ={1,2,3};
    int target=4;

    int n =arr.size();

    cout<<"The total number of ways is " <<countWaysToMakeChange(arr,n,target);
}

```

### **Output:**

The total number of ways is 4

### **Time Complexity: O(N\*T)**

Reason: There are two nested loops.

### **Space Complexity: O(T)**

Reason: We are using an external array of size 'T+1' to store two rows only.



# Unbounded Knapsack (DP-23)

 [takeuforward.org/data-structure/unbounded-knapsack-dp-23](https://takeuforward.org/data-structure/unbounded-knapsack-dp-23)

March 6, 2022

## Problem Link: Unbounded Knapsack

A thief wants to rob a store. He is carrying a bag of capacity W. The store has 'n' items of infinite supply. Its weight is given by the 'wt' array and its value by the 'val' array. He can either include an item in its knapsack or exclude it but can't partially have it as a fraction. We need to find the maximum value of items that the thief can steal. He can take a single item any number of times he wants and put it in his knapsack.

**Example:**

$N = 3$	$W = 10$
wt	2   4   6
val	5   11   13
<b>Answer : 27</b>	
The thief can put the items in the following way	
2 items of weight 4 and 1 item of weight 2.	
<b><math>11 + 11 + 5</math></b>	

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

Pre-req: 0/1 Knapsack

**Solution :**

## Why a Greedy Solution doesn't work?

The first approach that comes to our mind is greedy. A greedy solution will fail in this problem because there is no 'uniformity' in data. While selecting a local better choice we may choose an item that will in long term give less value.

As the greedy approach doesn't work, we will try to generate all possible combinations using **recursion** and select the combination which gives us the **maximum** value in the given constraints.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given 'n' items. Their weight is represented by the 'wt' array and value by the 'val' array. So clearly one parameter will be 'ind', i.e index upto which the array items are being considered.

There is one more parameter "W". We need the capacity of the knapsack to decide whether we can pick an array item or not in the knapsack.

So, we can say that initially, we need to find  $f(n-1, W)$  where  $W$  is the overall capacity given to us.  $f(n-1, W)$  means we are finding the maximum value of items that the thief can steal from items with index 0 to  $n-1$  capacity  $W$  of the knapsack.

**f(ind,W) ->Maximum value of items from index 0 to ind, with capacity of knapsack W**

### Base Cases:

If  $ind==0$ , it means we are at the first item. Now, in an unbounded knapsack we can pick an item any number of times we want. As there is only one item left, we will pick for  $W/wt[0]$  times because we ultimately want to maximize the value of items while respecting the constraint of weight of the knapsack. The value added will be the product of the number of items picked and value of the individual item.  
Therefore we return  $(W/wt[0]) * val[0]$ .

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index item. If we exclude the current item, the capacity of the bag will not be affected and the value added will be 0 for the current item. So we will call the recursive function  $f(ind-1, W)$

- **Include the current element in the subsequence:**

We will try to find a subsequence by considering the current item to the knapsack. As we have included the item, the capacity of the knapsack will be updated to  $W-wt[ind]$  and the current item's value ( $val[ind]$ ) will also be added to the further recursive call answer.

Now here is the catch, as there is an unlimited supply of coins, we want to again form a solution with the same item value. So we **will not** recursively call for  $f(ind-1, W-wt[ind])$  rather we will stay at that index only and call for  **$f(ind, W-wt[ind])$**  to find the answer.

**Note:** We will consider the current item in the subsequence only when the current element's weight is less than or equal to the capacity 'W' of the knapsack, if it isn't we will not be considering it.

```
f(ind,W) {
    if( ind == 0) {
        return (W/wt[0]) * val[0]
    }
}
```

```
f(ind,W) {
    if( ind == 0) {
        return (W/wt[0]) * val[0]
    }
    notTake = 0 + f(ind-1,W)
    take = INT_MIN
    if(wt[ind]<=W){
        take =  val[ind] + f(ind,W-val[ind])
    }
}
```

### Step 3: Return the maximum of take and notTake

As we have to return the maximum amount of value, we will return the max of take and notTake as our answer.

The final pseudocode after steps 1, 2, and 3:

```
f(ind,W) {  
    if( ind == 0) {  
        return (W/wt[0]) * val[0]  
    }  
    notTake = 0 + f(ind-1,W)  
    take = INT_MIN  
    if(wt[ind]<=W){  
        take =  val[ind] + f(ind,W-val[ind])  
    }  
    return max(notTake, take)  
}
```

#### Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][W+1]$ . The size of the input array is ‘N’, so the index will always lie between ‘0’ and ‘n-1’. The capacity can take any value between ‘0’ and ‘W’. Therefore we take the dp array as  $dp[n][W+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,target)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][target] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][target]$  to the solution we get.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int knapsackUtil(vector<int>& wt, vector<int>& val, int ind, int W, vector<vector<int>>& dp){

    if(ind == 0){
        return ((int)(W/wt[0])) * val[0];
    }

    if(dp[ind][W]!=-1)
        return dp[ind][W];

    int notTaken = 0 + knapsackUtil(wt,val,ind-1,W,dp);

    int taken = INT_MIN;
    if(wt[ind] <= W)
        taken = val[ind] + knapsackUtil(wt,val,ind,W-wt[ind],dp);

    return dp[ind][W] = max(notTaken,taken);
}

int unboundedKnapsack(int n, int W, vector<int>& val,vector<int>& wt) {

    vector<vector<int>> dp(n, vector<int>(W+1, -1));
    return knapsackUtil(wt, val, n-1, W, dp);
}

int main() {

    vector<int> wt = {2,4,6};
    vector<int> val = {5,11,13};
    int W=10;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<unboundedKnapsack
    (n,W,val,wt);
}

```

### **Output:**

The Maximum value of items, thief can steal is 27

### **Time Complexity: O(N\*W)**

Reason: There are  $N \times W$  states therefore at max ' $N \times W$ ' new problems will be solved.

### **Space Complexity: O(N\*W) + O(N)**

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (  $O(N \times W)$ ).

### **Steps to convert Recursive Solution to Tabulation one.**

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

First, we need to initialize the base conditions of the recursive solution.

- At  $\text{ind}==0$ , we are considering the first element, so we will assign its value as  $(i/\text{wt}[o]) * \text{val}[o]$ , where  $i$  will iterate from 0 to  $W$ .
- Next, we are done for the first row, so our ‘ $\text{ind}$ ’ variable will move from 1 to  $n-1$ , whereas our ‘ $\text{cap}$ ’ variable will move from 0 to ‘ $W$ ’. We will set the nested loops to traverse the dp array.
- Inside the nested loops we will apply the recursive logic to find the answer of the cell.
- When the nested loop execution has ended, we will return  $\text{dp}[n-1][W]$  as our answer.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int unboundedKnapsack(int n, int W, vector<int>& val, vector<int>& wt) {

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    //Base Condition

    for(int i=wt[0]; i<=W; i++){
        dp[0][i] = ((int) i/wt[0]) * val[0];
    }

    for(int ind =1; ind<n; ind++){
        for(int cap=0; cap<=W; cap++){

            int notTaken = 0 + dp[ind-1][cap];

            int taken = INT_MIN;
            if(wt[ind] <= cap)
                taken = val[ind] + dp[ind][cap - wt[ind]];

            dp[ind][cap] = max(notTaken, taken);
        }
    }

    return dp[n-1][W];
}

int main() {

    vector<int> wt = {2,4,6};
    vector<int> val = {5,11,13};
    int W=10;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<unboundedKnapsack
    (n,W,val,wt);
}

```

### **Output:**

The Maximum value of items, thief can steal is 27

### **Time Complexity: O(N\*W)**

Reason: There are two nested loops

### **Space Complexity: O(N\*W)**

Reason: We are using an external array of size 'N\*W'. Stack Space is eliminated.

### **Part 3: Space Optimization**

If we closely look the relation,

$$dp[ind][cap] = \max(dp[ind-1][cap], dp[ind][cap - wt[ind]])$$

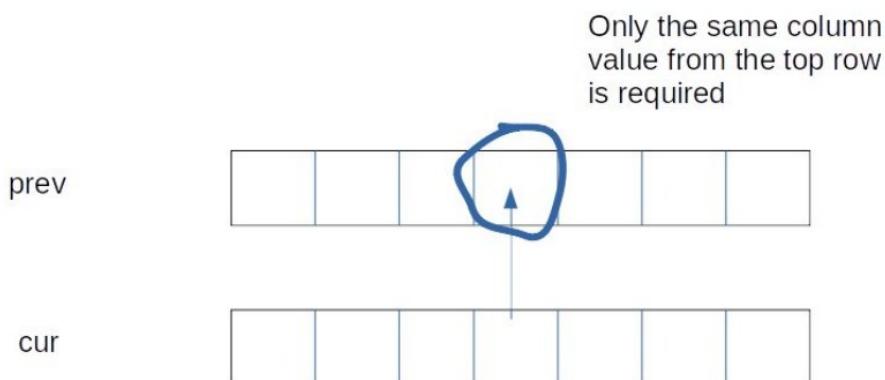
We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

We will be space optimizing this solution using **only one row**.

### Intuition:

If we clearly see the values required:  $dp[ind-1][cap]$  and  $dp[ind-1][cap - wt[ind]]$ , we can say that if we are at a column  $cap$ , we will only require the values shown in the blue box(of the same column) from the previous row and other values will be from the cur row itself. So why do we need to store an entire array for it?

## Space Optimization using Single Row



If we need only one value from the prev row, there is no need to store an entire row. We can work a bit smarter.

We can use the cur row itself to store the required value in the following way:

- We somehow make sure that the previous value( say preValue) is available to us in some manner ( we will discuss later how we got the value).
- Now, let us say that we want to find the value of cell  $cur[3]$ , by going through the relation we find that we need a preValue and one value from the cur row.
- We see that to calculate the  $cur[3]$  element, we need only a single variable (preValue). The catch is that we can initially place this preValue at the position  $cur[3]$  (before finding its updated value) and later while calculating for the current row's cell  $cur[3]$ , the value present there automatically serves as the preValue and we can use it to find the required  $cur[3]$  value. ( If there is any confusion please see the code).

- After calculating the cur[3] value we store it at the cur[3] position so this cur[3] will automatically serve as preValue for the next row. In this way we space optimize the tabulation approach by just using one row.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int unboundedKnapsack(int n, int W, vector<int>& val, vector<int>& wt) {

    vector<int> cur(W+1, 0);

    //Base Condition

    for(int i=wt[0]; i<=W; i++){
        cur[i] = ((int)i/wt[0]) * val[0];
    }

    for(int ind =1; ind<n; ind++){
        for(int cap=0; cap<=W; cap++){

            int notTaken = cur[cap];

            int taken = INT_MIN;
            if(wt[ind] <= cap)
                taken = val[ind] + cur[cap - wt[ind]];

            cur[cap] = max(notTaken, taken);
        }
    }

    return cur[W];
}

int main() {

    vector<int> wt = {2,4,6};
    vector<int> val = {5,11,13};
    int W=10;

    int n = wt.size();

    cout<<"The Maximum value of items, thief can steal is " <<unboundedKnapsack
    (n,W,val,wt);
}
```

### Output:

The Maximum value of items, thief can steal is 27

**Time Complexity: O(N\*W)**

Reason: There are two nested loops.

**Space Complexity: O(W)**

Reason: We are using an external array of size 'W+1' to store only one row.

# Rod Cutting Problem | (DP – 24)

 [takeuforward.org/data-structure/rod-cutting-problem-dp-24](https://takeuforward.org/data-structure/rod-cutting-problem-dp-24)

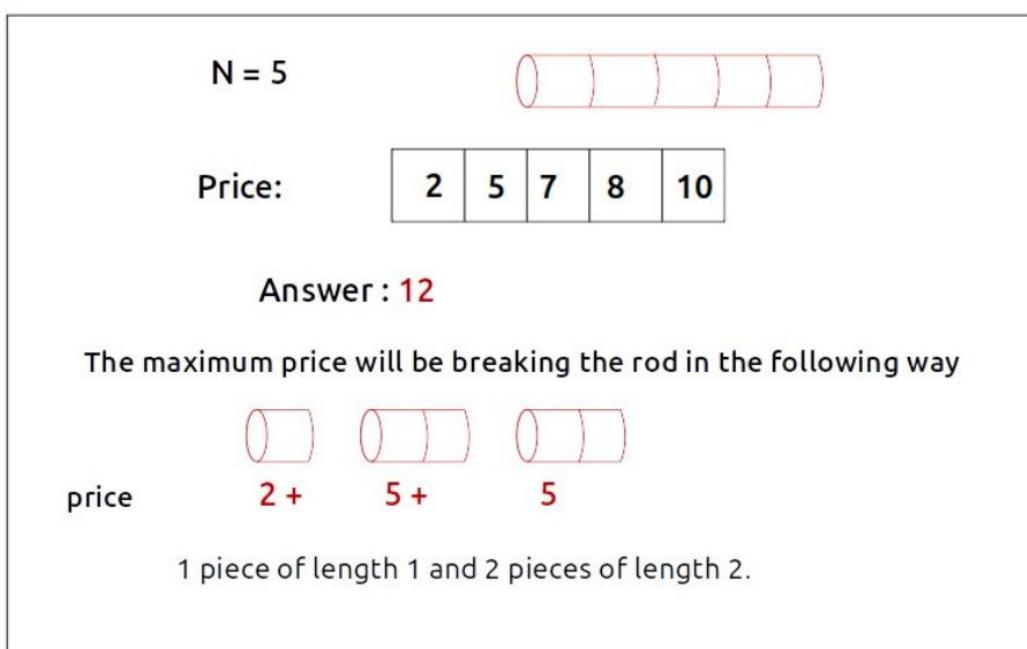
March 7, 2022

**Problem Statement:** Rod Cutting Problem

**Problem Link:** [Rod Cutting Problem](#)

We are given a rod of size 'N'. It can be cut into pieces. Each length of a piece has a particular price given by the price array. Our task is to find the maximum revenue that can be generated by selling the rod after cutting( if required) into pieces.

**Example:**



**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Pre-req:** [Unbounded Knapsack](#)

**Solution :**

**Intuition:**

We want to cut the rod into pieces to get the maximum price. We can have 0 cuts as well if the whole rod is giving us a better price.

Suppose we have a rod of length 3, we will have  $2^{3-1} = 4$  ways to cut the rod ( as we can cut the rod at (3-1) places and at every place we have 2 options, to cut it or not). The following figure shows all the ways:

### Rod of length 3



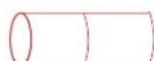
### Ways to cut



No cut – 1 piece



1 cut – 2 pieces



1 cut – 2 pieces



2 cuts – 3 pieces

The following two observations should be noted:

- We can't have a rod piece bigger than the original rod length (self-explanatory). Therefore the rod length N acts like a limiting factor to the size of pieces we can cut.
- If we cut the piece of length 'k', we can still cut another piece from the remaining length of size 'k' again.

From the above two points, after a little thinking, we can say that this problem can be solved by the same technique we used in solving the **Unbounded Knapsack**

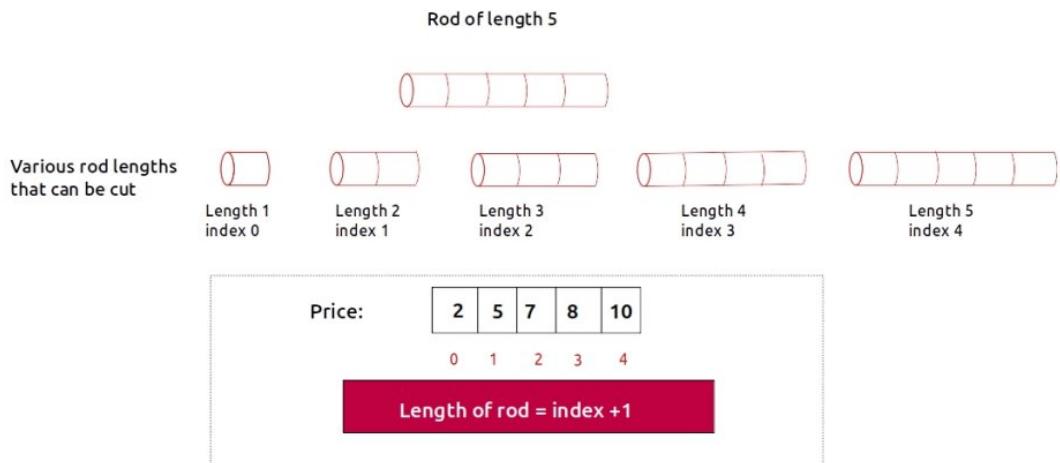
The rod pieces are equivalent to the items and the rod length is equivalent to the knapsack capacity. From where we will discuss the unbounded knapsack problem with the required modifications.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in Dynamic Programming Introduction.

**Step 1:** Express the problem in terms of indexes.

The price of individual lengths is given by the price array. Price[0] gives us the price of a rod with length 1, index 2 gives us a rod with length 2, and so on. We can cut a rod from index 0 to  $N-1$ . (where the length of the rod will be 'ind+1'). One parameter will be 'ind' which tells us the rod length that we want to cut from the original rod. Initially, we would want to consider the entire rod length.



We are given a rod with length 'N'. So clearly another parameter will be 'N', i.e the total rod length that is given to us so that we can know the maximum length of rods that we can cut.

Initially, we would want to find  $f(N-1, N)$ , i.e the maximum revenue generated by considering all rod lengths from index 0 to  $N-1$  (i.e from length 1 to length N) with the total rod length given as N.

We can generalize this as:

**$f(ind, N) \rightarrow$  The maximum revenue generated by considering rods from index 0 to index  $ind-1$ , with total rod length of N**

### Base Cases:

If  $ind==0$ , it means we are considering a rod of length 1. Its price is given by  $price[0]$ . Now for length N, the number of rod pieces of length 1 will be N ( $N/1$ ). Therefore we will return the maximum revenue generated, i.e ' $N * price[0]$ '.

**Step 2:** Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "[Recursion on Subsequences](#)".

We have two choices:

```
f(ind,N){
    if(ind==0)
        return N*val[0]
}
```

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index irod length. If we exclude the current item, the total length of the rod will not be affected and the revenue added will be 0 for the current item. So we will call the recursive function  $f(ind-1, N)$
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index rod length. As we have cut the rod, the total rod length will be updated to  $N-(ind+1)$ , where ‘ $ind+1$ ’ is the rod length, and the current rod piece price’s value ( $price[ind]$ ) will also be added to the further recursive call answer.

Now here is the catch, we can cut the rod piece of the same index length. So we **will not** recursively call for  $f(ind-1, N-(ind+1))$  rather we will stay at that index only and call for  **$f(ind, N-(ind+1))$**  to find the answer.

**Note:** We will consider the current item in the subsequence only when the current element’s length is less than or equal to the total rod length ‘ $N$ ’, if it isn’t we will not be considering it.

```

f(ind,N){
    if(ind==0)
        return N*val[0]

    notTake = 0 + f(ind-1,N)

    take = INT_MIN
    rodLength = ind+1
    if(rodLength <=N)
        take = price[ind] + f(ind,N-rodLength)

}

```

### Step 3: Return the maximum of take and notTake

As we have to return the maximum amount of price we can generate, we will return the maximum of take and notTake.

The final pseudocode after steps 1, 2, and 3:

```

f(ind,N){
    if(ind==0)
        return N*val[0]

    notTake = 0 + f(ind-1,N)

    take = INT_MIN
    rodLength = ind+1
    if(rodLength <=N)
        take = price[ind] + f(ind,N-rodLength)

    return max(take, notTake)
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[N][N+1]$ . The size of the price array is ‘N’, so the index will always lie between ‘0’ and ‘N-1’. The rod length can take any value between ‘0’ and ‘N’. Therefore we take the dp array as  $dp[N][N+1]$
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(ind,N)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[ind][N] \neq -1$ ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][N]$  to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int cutRodUtil(vector<int>& price, int ind, int N, vector<vector<int>>& dp){

    if(ind == 0){
        return N*price[0];
    }

    if(dp[ind][N]!=-1)
        return dp[ind][N];

    int notTaken = 0 + cutRodUtil(price,ind-1,N,dp);

    int taken = INT_MIN;
    int rodLength = ind+1;
    if(rodLength <= N)
        taken = price[ind] + cutRodUtil(price,ind,N-rodLength,dp);

    return dp[ind][N] = max(notTaken,taken);
}

int cutRod(vector<int>& price,int N) {

    vector<vector<int>> dp(N,vector<int>(N+1,-1));
    return cutRodUtil(price,N-1,N,dp);
}

int main() {

    vector<int> price = {2,5,7,8,10};

    int n = price.size();

    cout<<"The Maximum price generated is "<<cutRod(price,n);
}

```

### **Output:**

The Maximum price generated is 12

### **Time Complexity: O(N\*N)**

Reason: There are  $N*(N+1)$  states therefore at max ' $N*(N+1)$ ' new problems will be solved.

### **Space Complexity: O(N\*N) + O(N)**

Reason: We are using a recursion stack space( $O(N)$ ) and a 2D array (  $O(N*(N+1))$ ).

### **Steps to convert Recursive Solution to Tabulation one.**

---

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

- First we need to initialize the base conditions of the recursive solution. At  $ind == 0$ , we are considering a unit length rod, so we will assign its value as  $(i * price[0])$ , where  $i$  will iterate from 0 to N.
- Next, we are done for the first row, so our ‘ind’ variable will move from 1 to  $n-1$ , whereas our ‘length’ variable will move from 0 to ‘N’. We will set the nested loops to traverse the dp array.
- Inside the nested loops we will apply the recursive logic to find the answer of the cell.
- When the nested loop execution has ended, we will return  $dp[N-1][N]$  as our answer.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int cutRod(vector<int>& price, int N) {

    vector<vector<int>> dp(N, vector<int>(N+1, -1));

    for(int i=0; i<=N; i++){
        dp[0][i] = i*price[0];
    }

    for(int ind=1; ind<N; ind++){
        for(int length =0; length<=N; length++){

            int notTaken = 0 + dp[ind-1][length];

            int taken = INT_MIN;
            int rodLength = ind+1;
            if(rodLength <= length)
                taken = price[ind] + dp[ind][length-rodLength];

            dp[ind][length] = max(notTaken, taken);
        }
    }

    return dp[N-1][N];
}

int main() {
    vector<int> price = {2,5,7,8,10};

    int n = price.size();

    cout<<"The Maximum price generated is "<<cutRod(price,n);
}

```

### **Output:**

The Maximum price generated is 12

### **Time Complexity: O(N\*N)**

Reason: There are two nested loops

### **Space Complexity: O(N\*N)**

Reason: We are using an external array of size ‘N\*(N+1)’. Stack Space is eliminated.

## **Part 3: Space Optimization**

---

If we closely look the relation,

$$dp[ind][length] = \max(dp[ind-1][length], dp[ind][length-(ind+1)])$$

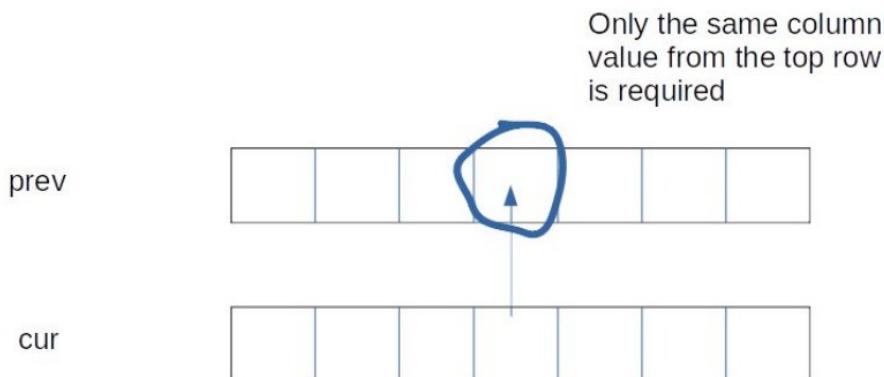
We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

We will be space optimizing this solution using **only one row**.

### Intuition:

If we clearly see the values required:  $dp[ind-1][cap]$  and  $dp[ind-1][cap - wt[ind]]$ , we can say that if we are at a column cap, we will only require the values shown in the blue box(of the same column) from the previous row and other values will be from the cur row itself. So why do we need to store an entire array for it?

## Space Optimization using Single Row



If we need only one value from the prev row, there is no need to store an entire row. We can work a bit smarter.

We can use the cur row itself to store the required value in the following way:

- We somehow make sure that the previous value( say preValue) is available to us in some manner ( we will discuss later how we got the value).
- Now, let us say that we want to find the value of cell  $cur[3]$ , by going through the relation we find that we need a preValue and one value from the cur row.
- We see that to calculate the  $cur[3]$  element, we need only a single variable (preValue). The catch is that we can initially place this preValue at the position  $cur[3]$  (before finding its updated value) and later while calculating for the current row's cell  $cur[3]$ , the value present there automatically serves as the preValue and we can use it to find the required  $cur[3]$  value. ( If there is any confusion please see the code).

- After calculating the cur[3] value we store it at the cur[3] position so this cur[3] will automatically serve as preValue for the next row. In this way we space optimize the tabulation approach by just using one row.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int cutRodUtil(vector<int>& price, int ind, int N, vector<vector<int>>& dp){

    if(ind == 0){
        return N*price[0];
    }

    if(dp[ind][N]!=-1)
        return dp[ind][N];

    int notTaken = 0 + cutRodUtil(price,ind-1,N,dp);

    int taken = INT_MIN;
    int rodLength = ind+1;
    if(rodLength <= N)
        taken = price[ind] + cutRodUtil(price,ind,N-rodLength,dp);

    return dp[ind][N] = max(notTaken,taken);
}

int cutRod(vector<int>& price,int N) {

    vector<int> cur (N+1,0);

    for(int i=0; i<=N; i++){
        cur[i] = i*price[0];
    }

    for(int ind=1; ind<=N; ind++){
        for(int length =0; length<=N; length++){

            int notTaken = 0 + cur[length];

            int taken = INT_MIN;
            int rodLength = ind+1;
            if(rodLength <= length)
                taken = price[ind] + cur[length-rodLength];

            cur[length] = max(notTaken,taken);
        }
    }

    return cur[N];
}

int main() {

    vector<int> price = {2,5,7,8,10};

    int n = price.size();
}

```

```
cout<<"The Maximum price generated is "<<cutRod(price,n);  
}
```

**Output:**

The Maximum price generated is 12

**Time Complexity: O(N\*N)**

Reason: There are two nested loops.

**Space Complexity: O(N)**

Reason: We are using an external array of size 'N+1' to store only one row.

# Longest Common Subsequence | (DP – 25)

 [takeuforward.org/data-structure/longest-common-subsequence-dp-25](https://takeuforward.org/data-structure/longest-common-subsequence-dp-25)

March 12, 2022

**Problem Statement:** Introduction to DP on Strings – Longest Common Subsequence

In the coming articles, we will discuss problems related to ‘Dynamic Programming on Strings’. We will discuss the problem of ‘**Longest Common Subsequence**’ in this article. Before proceeding further, let us understand what is the “Longest Common Subsequence”, or rather what is a “**subsequence**”?

A subsequence of a string is a list of characters of the string where some characters are deleted ( or not deleted at all) and they should be in the same order in the subsequence as in the original string.

For eg:

string : “abc”

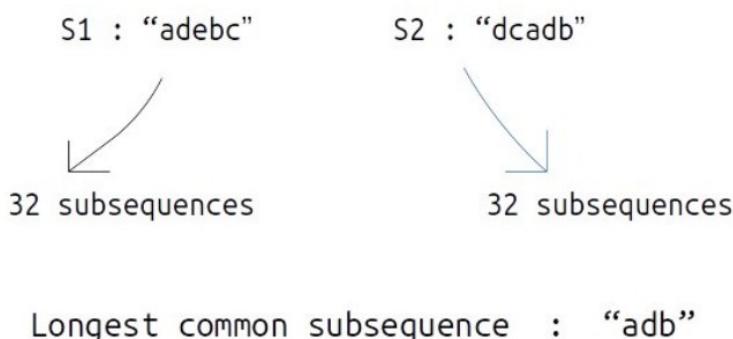
Subsequences: {“ ”, “a”, “b”, “c”, “ab”, “bc”, “ac”, “abc” }

Strings like “cab”, “bc” will not be called as a subsequence of “abc” as the characters are not coming in the same order.

**Note:** For a string of length n, the number of subsequences will be  $2^n$ .

Now we will look at “Longest Common Subsequence”. The longest Common Subsequence is defined for two strings. It is the common subsequence that has the greatest length.

For example:



**Problem Link:** [Longest Common Subsequence](#)

### Solution :

### **Approach 1: Using Brute Force**

We are given two strings, S<sub>1</sub>, and S<sub>2</sub> (suppose of same length n), the simplest approach will be to generate all the subsequences and store them, then manually find out the longest common subsequence.

This naive approach will give us the correct answer but to generate all the subsequences, we will require **exponential ( $2^n$ )** time. Therefore we will try some other approaches.

### **Approach 2: Using Dynamic Programming**

We would want to try something that can give us the longest common subsequence on the way of generating all subsequences. To generate all subsequences we will use recursion and in the recursive logic we will figure out a way to solve this problem.

### **Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two strings  $S_1$  and  $S_2$ :

S1 : “adabc”      S2 : “dcadbmn”  
                ↑                              ↑  
                ind1                            ind2

A single variable can't express both the strings at the same time, so we will use two variables  $\text{ind1}$  and  $\text{ind2}$ . They mean that we are considering string  $S_1$  from index  $0$  to  $\text{ind1}$  and string  $S_2$  from index  $0$  to  $\text{S2}$ . So our recursive function will look like this:

$f(ind1, ind2) \rightarrow$  Longest common subsequence of  $S1[0...ind1]$  and  $S2[0...ind2]$

**Step 2:** Explore all possibilities at a given index

## Intuition for Recursive Logic

In the function  $f(ind1,ind2)$ ,  $ind1$  and  $ind2$  are representing two characters from strings  $S1$  and  $S2$  respectively. For example:

$f(2,2)$

S1 : "acd"      S2 : "ced"  
  ↑               ↑  
  ind1           ind2

Now, there can be two possibilities,

**if( $S1[ind1] == S2[ind2]$ )** as in the figure below. In this case this common element will represent a unit length common subsequence, so we can say that we have found one character and we can shrink both the strings by 1 to find the longest common subsequence in the remaining pair of strings.

$f(2,2)$

S1 : "acd"      S2 : "ced"  
  ↑               ↑  
  ind1           ind2

$S1[ind1] == S2[ind2]$

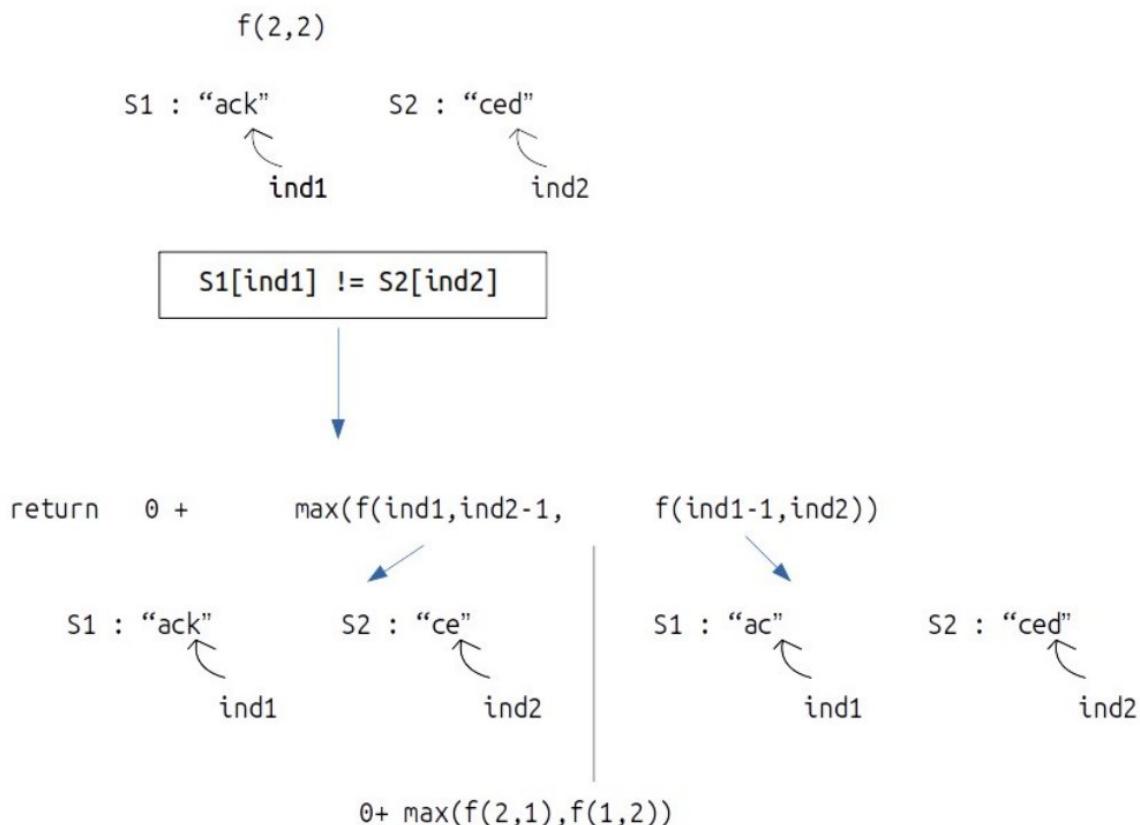


return 1 +  $f(ind1-1, ind2-1)$

S1 : "ac"      S2 : "ce"  
  ↑               ↑  
  ind1           ind2

$f(1,1)$

**if( $S_1[\text{ind1}] \neq S_2[\text{ind2}]$ )** as in the figure given below. In this case we know that the current characters represented by ind1 and ind2 will be different. So, we need to compare the ind1 character with shrunk  $S_2$  and ind2 with shrunk  $S_1$ . But how do we make this comparison ? If we make a single recursive call as we did above to  $f(\text{ind1}-1, \text{ind2}-1)$ , we may lose some characters of the subsequence. Therefore we make two recursive calls: one to  $f(\text{ind1}, \text{ind2}-1)$  (shrinking only  $S_1$ ) and one to  $f(\text{ind1}-1, \text{ind2})$  (shrinking only  $S_2$ ). Then when we return max of both the calls.

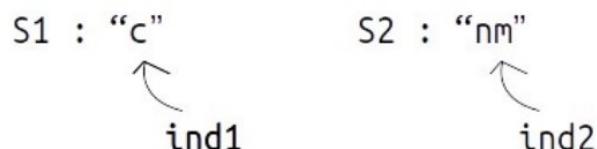


### Step 3: Return the maximum of the choices

In the first case, we have only one choice but in the second case we have two choices, as we have to return the longest common subsequences, we will return the maximum of both the choices in the second case.

#### Base Case:

For a case like this:



As  $S_1[\text{ind1}] \neq S_2[\text{ind2}]$

We will make a call to  $f(0-1,1)$ , i.e  $f(-1,1)$  but a negative index simply means that there are no more indexes to be explored, so we simply return 0. Same is the case when  $S_1[\text{ind1}] == S_2[\text{ind2}]$

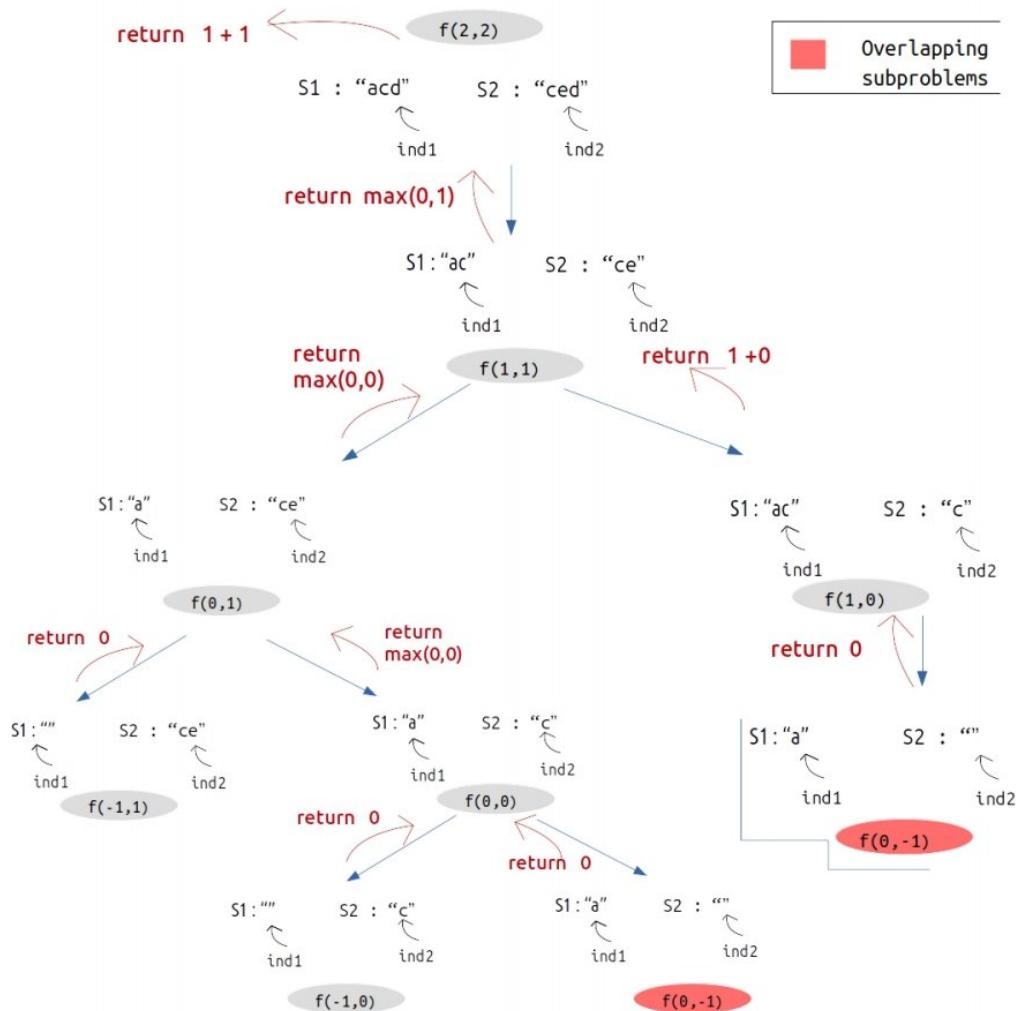
If ( $\text{ind1} < 0 \text{ || ind2} < 0$ ) return 0.

The final pseudocode after steps 1, 2, and 3:

```
f(ind1,ind2){  
    if(ind1<0 || ind2<0 )  
        return 0  
  
    if( S1[ind1] == S2[ind2] )  
        return 1 + f(ind1-1,ind2-1)  
  
    else  
  
        return 0 + max(f(ind1,ind2-1), f(ind1-1,ind2))  
}
```

## Recursive Tree

We will dry run this example:



### Steps to memoize a recursive solution:

As we see there are overlapping subproblems in the recursive tree, we can memorize the recursive code to reduce the time complexity.

1. Create a dp array of size  $[N][M]$  where N and M are lengths of S1 and S2 respectively. It will store all the possible different states that our recursive function will take.
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say  $f(\text{ind1}, \text{ind2})$ ), we first check whether the answer is already calculated using the dp array (i.e.  $\text{dp}[\text{ind1}][\text{ind2}] \neq -1$ ). If yes, simply return the value from the dp array.

- If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[ind][ind2]$  to the solution we get.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int lcsUtil(string& s1, string& s2, int ind1, int ind2, vector<vector<int>>& dp){

    if(ind1<0 || ind2<0)
        return 0;

    if(dp[ind1][ind2]!=-1)
        return dp[ind1][ind2];

    if(s1[ind1] == s2[ind2])
        return dp[ind1][ind2] = 1 + lcsUtil(s1,s2,ind1-1,ind2-1,dp);

    else
        return dp[ind1][ind2] = 0 + max(lcsUtil(s1,s2,ind1,ind2-1,dp),lcsUtil(s1,s2,ind1-1,ind2,dp));
}

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<vector<int>> dp(n, vector<int>(m, -1));
    return lcsUtil(s1,s2,n-1,m-1,dp);
}

int main() {

    string s1= "acd";
    string s2= "ced";

    cout<<"The Length of Longest Common Subsequence is "<<lcs(s1,s2);
}
```

**Output:** The Length of Longest Common Subsequence is 2

**Time Complexity: O(N\*M)**

Reason: There are  $N*M$  states therefore at max ' $N*M$ ' new problems will be solved.

## **Space Complexity: $O(N*M) + O(N+M)$**

Reason: We are using an auxiliary recursion stack space( $O(N+M)$ ) (see the recursive tree, in the worst case, we will go till  $N+M$  calls at a time) and a 2D array (  $O(N*M)$ ).

### **Steps to convert Recursive Solution to Tabulation one.**

---

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization.

#### **Initialization: Shifting of indexes**

---

In the recursive logic, we set the base case to if( $ind1 < 0 || ind2 < 0$ ) but we can't set the dp array's index to -1. Therefore a hack for this issue is to shift every index by 1 towards the right.

Recursive code indexes:      -1, 0, 1, ..., n

Shifted indexes :      0, 1, ..., n+1

- Therefore, now the base case will be if( $ind1 == 0 || ind2 == 0$ ).
- Similarly, we will implement the recursive code by keeping in mind the shifting of indexes, therefore  $S1[ind1]$  will be converted to  $S1[ind1-1]$ . Same for others.
- At last we will print  $dp[N][M]$  as our answer.

#### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, -1));
    for(int i=0;i<=n;i++){
        dp[i][0] = 0;
    }
    for(int i=0;i<=m;i++){
        dp[0][i] = 0;
    }

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                dp[ind1][ind2] = 1 + dp[ind1-1][ind2-1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1-1][ind2], dp[ind1][ind2-1]);
        }
    }

    return dp[n][m];
}

int main() {

    string s1= "acd";
    string s2= "ced";

    cout<<"The Length of Longest Common Subsequence is "<<lcs(s1,s2);
}

```

### **Output:**

The Length of Longest Common Subsequence is 2

### **Time Complexity: O(N\*M)**

Reason: There are two nested loops

### **Space Complexity: O(N\*M)**

Reason: We are using an external array of size ‘N\*M’. Stack Space is eliminated.

## **Part 3: Space Optimization**

---

If we closely we are using two rows: **dp[ind1-1][ ], dp[ind][ ]**,

So we are not required to contain an entire array, we can simply have two rows prev and cur where prev corresponds to  $dp[ind-1]$  and cur to  $dp[ind]$ .

After declaring prev and cur, replace  $dp[ind-1]$  to prev and  $dp[ind]$  with cur and after the inner loop executes, we will set prev = cur, so that the cur row can serve as prev for the next index.

### Code:

- C++Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<int> prev(m+1,0), cur(m+1,0);

    // Base Case is covered as we have initialized the prev and cur to 0.

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                cur[ind2] = 1 + prev[ind2-1];
            else
                cur[ind2] = 0 + max(prev[ind2],cur[ind2-1]);
        }
        prev= cur;
    }

    return prev[m];
}

int main() {

    string s1= "acd";
    string s2= "ced";

    cout<<"The Length of Longest Common Subsequence is "<<lcs(s1,s2);
}
```

### Output:

The Maximum price generated is 12

**Time Complexity: O(N\*M)**

Reason: There are two nested loops.

## **Space Complexity: O(M)**

Reason: We are using an external array of size 'M+1' to store only two rows.

# Print Longest Common Subsequence | (DP – 26)

 [takeuforward.org/data-structure/print-longest-common-subsequence-dp-26](https://takeuforward.org/data-structure/print-longest-common-subsequence-dp-26)

March 20, 2022

## Problem Statement:

### Print Longest Common Subsequence

In the previous article Longest Common Subsequence, we learned to print the length of the longest common subsequence of two strings. In this article, we will learn to print the actual string of the longest common subsequence.

**Prereq:** Longest Common Subsequence

## Intuition:

Let us consider the following example:

S1 : "abcde"

S2 : "bdgek"

**lcs String :** **bde**

**len(lcs) :** **3**

We will continue from where we left in the article DP-25. There in the tabulation approach, we declared a dp array and  $dp[n][m]$  will have the length of the longest common subsequence., i.e  $dp[n][m] = 3$ .

Now, with help of two nested loops, if we print the dp array, it will look like this:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	1	1	1	1	1
3	0	1	1	1	1	1
4	0	1	2	2	2	2
5	0	1	2	2	3	3

Here  $dp[5][5]$  gives us the length of the longest common subsequence: 3.

Now let us try to form the string itself. We know its length already. We give it the name str.

We will use the dp array to form the LCS string. For that, we need to think, about how did the dp array was originally filled. The tabulation approach used **1-based indexing**. We also write the characters corresponding to the indexes of the dp array:

### Forming the lcs string

str :  
size - 3

0	1	2

S1 : "abcde"

S2 : "bdgek"

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'b'	'd'	'g'	'e'	'k'
2	0	'b'	1	1	1	1
3	0	'c'	1	1	1	1
4	0	'd'	1	2	2	2
5	0	'e'	1	2	2	3

Now, let us see what were the conditions that we used while forming the dp array:

- **if( $S1[i-1] == S2[j-1]$ )**, then return  $1 + dp[i-1][j-1]$
- **if( $S1[i-1] != S2[j-1]$ )**, then return  $0 + \max(dp[i-1][j], dp[i][j-1])$

These two conditions along with the dp array give us all the required information required to print the LCS string.

### Approach:

The algorithm approach is stated below:

- We will fill the string str from the last by maintaining a pointer.
- We will start from the right-most cell of the dp array, initially  $i=n$  and  $j=m$ .
- At every cell, we will check if  $S1[i-1] == S2[j-1]$ , if it is then it means this character is a part of the longest common substring. So we will push it to the str (at last). Then we will move to the diagonally top-left() cell by assigning  $i$  to  $i-1$  and  $j$  to  $j-1$ .
- Else, this character is not a part of the longest common subsequence. It means that originally this cell got its value from its left cell ( $\leftarrow$ ) or from its top cell ( $\uparrow$ ). Whichever cell's value will be more of the two, we will move to that cell.

- We will continue till  $i > o$  and  $j > o$ , failing it we will break from the loop.
- At last we will get our lcs string in “str”.

### Dry Run:

---

S1 : "abcde"

S2 : "bdgek"

(i)

S1 : "abcde"

S2 : "bdgek"

(ii)

0      1      2      3      4      5

0	0	0	0	0	0	
1	0	'b'	'd'	'g'	'e'	'k'
2	0	'a'	0	0	0	0
3	0	'b'	1	1	1	1
4	0	'c'	1	1	1	1
5	0	'd'	1	2	2	2
	0	'e'	1	2	2	3

S1[4] != S2[4]

0      1      2      3      4      5

0	0	0	0	0	0	
1	0	'b'	'd'	'g'	'e'	'k'
2	0	'a'	0	0	0	0
3	0	'b'	1	1	1	1
4	0	'c'	1	1	1	1
5	0	'd'	1	2	2	2
	0	'e'	1	2	2	3

S1[4] == S2[3]

str :  
size - 3

0	1	2

str :  
size - 3

		e
0	1	2

---

At starting i=5 and j=5.

- As  $S1[4] \neq S2[4]$ , we move to the left side ( $\leftarrow$ ) as it's value is greater than the top value( $\uparrow$ ), therefore  $i=5$  and  $j=4$
- As  $S1[4] == S2[3]$ , we add the current character to the str string(at last) and move to  $i-1$  and  $j-1$  cell i.e top-left(), therefore  $i=4$  and  $j=3$ .

S1 : "abcde" | S2 : "bdgek"

(iii)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'a'	0	0	0	0
2	0	'b'	1	1	1	1
3	0	'c'	1	1	1	1
4	0	'd'	1	2	2	2
5	0	'e'	1	2	2	3

S1[3] != S2[2]

str : 

0	1	2

S1 : "abcde" | S2 : "bdgek"

(iv)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'a'	0	0	0	0
2	0	'b'	1	1	1	1
3	0	'c'	1	1	1	1
4	0	'd'	1	2	2	2
5	0	'e'	1	2	2	3

S1[3] == S2[1]

str : 

	d	e
0	1	2

- As S1[3] != S2[2], we move to the left cell ( $\leftarrow$ ) as its value is larger than the top cell( $\uparrow$ ), i becomes 4 and j becomes 2.
- As S1[3] == S2[1], we will add this character to str string and we will move to the top-left cell () i becomes 3 and j becomes 1

S1 : "abcde" | S2 : "bdgek"

(v)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'a'	0	0	0	0
2	0	'b'	1	1	1	1
3	0	'c'	1	1	1	1
4	0	'd'	1	2	2	2
5	0	'e'	1	2	2	3

S1[2] != S2[1]

str : 

	d	e
0	1	2

S1 : "abcde" | S2 : "bdgek"

(vi)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'a'	0	0	0	0
2	0	'b'	1	1	1	1
3	0	'c'	1	1	1	1
4	0	'd'	1	2	2	2
5	0	'e'	1	2	2	3

S1[1] == S2[0]

str : 

b	d	e
0	1	2

- As  $S1[2] \neq S2[1]$ , we will move to the top cell ( $\uparrow$ ) as its value is greater than the left cell ( $\leftarrow$ ). Now i becomes 2 and j remains 1.
- As  $S1[1] == S2[0]$ , we will add this character to str string and we will move to the top-left cell () i becomes 1 and j becomes 1 and str becomes o.
- As j is zero, we have hit the exit condition so we will break out of the loop and str contains the longest common subsequence.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

void lcs(string s1, string s2) {

    int n = s1.size();
    int m = s2.size();

    vector < vector < int >> dp(n + 1, vector < int > (m + 1, 0));
    for (int i = 0; i <= n; i++) {
        dp[i][0] = 0;
    }
    for (int i = 0; i <= m; i++) {
        dp[0][i] = 0;
    }

    for (int ind1 = 1; ind1 <= n; ind1++) {
        for (int ind2 = 1; ind2 <= m; ind2++) {
            if (s1[ind1 - 1] == s2[ind2 - 1])
                dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
        }
    }

    int len = dp[n][m];
    int i = n;
    int j = m;

    int index = len - 1;
    string str = "";
    for (int k = 1; k <= len; k++) {
        str += "$"; // dummy string
    }

    while (i > 0 && j > 0) {
        if (s1[i - 1] == s2[j - 1]) {
            str[index] = s1[i - 1];
            index--;
            i--;
            j--;
        } else if (s1[i - 1] > s2[j - 1]) {
            i--;
        } else j--;
    }
    cout << str;
}

int main() {

    string s1 = "abcde";
    string s2 = "bdgek";

    cout << "The Longest Common Subsequence is ";
}

```

```
    lcs(s1, s2);  
}
```

**Output:** The Longest Common Subsequence is bde

**Time Complexity: O(N\*M)**

Reason: There are two nested loops

**Space Complexity: O(N\*M)**

Reason: We are using an external array of size 'N\*M'. Stack Space is eliminated.

# Longest Common Substring | (DP – 27)

---

 [takeuforward.org/data-structure/longest-common-substring-dp-27](https://takeuforward.org/data-structure/longest-common-substring-dp-27)

March 23, 2022

## Problem Statement: Longest Common Substring

A substring of a string is a subsequence in which all the characters are **consecutive**. Given two strings, we need to find the longest common substring.

**Example:**

S1: "abcjklp"

S2: "acjkp"

Longest Common Substring: "cjk"

We need to print the **length** of the longest common substring.

**Problem Link:** [Longest Common Substring](#)

**Solution :**

---

**Pre-req:** [Longest Common Subsequence](#), [Print Longest Common Subsequence](#)

**Approach:**

We can modify the approach we used in the article [Longest Common Subsequence](#), in order to find the longest common substring. The main distinguishing factor between the two is the consecutiveness of the characters.

While finding the longest common subsequence, we were using two pointers (ind1 and ind2) to map the characters of the two strings. We will again have the same set of conditions for finding the longest common substring, with slight modifications to what we do when the condition becomes true.

We will try to form a solution in the bottom-up (tabulation) approach. We will set two nested loops with loop variables i and j.

**Thinking in terms of consecutiveness of characters**

We have two conditions:

- if( $S1[i-1] \neq S2[j-1]$ ), the characters don't match, therefore the consecutiveness of characters is broken. So we set the cell value ( $dp[i][j]$ ) as 0.

- if( $S_1[i-1] == S_2[j-1]$ ), then the characters match and we simply set its value to  $1 + dp[i-1][j-1]$ . We have done so because  $dp[i-1][j-1]$  gives us the longest common substring till the last cell character (current strings -{matching character}). As the current cell's character is matching we are adding 1 to the consecutive chain.

**Note:**  $dp[n][m]$  will **not** give us the answer; rather the maximum value in the entire  $dp$  array will give us the length of the longest common substring. This is because there is no restriction that the longest common substring is present at the end of both the strings.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int lcs(string &s1, string &s2){

    int n = s1.size();
    int m = s2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    int ans = 0;

    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(s1[i-1]==s2[j-1]){
                int val = 1 + dp[i-1][j-1];
                dp[i][j] = val;
                ans = max(ans, val);
            }
            else
                dp[i][j] = 0;
        }
    }

    return ans;
}

int main() {

    string s1= "abcjklp";
    string s2= "acjkp";

    cout<<"The Length of Longest Common Substring is "<<lcs(s1,s2);
}
```

### Output:

The Length of Longest Common Substring is 3

## **Time Complexity: O(N\*M)**

Reason: There are two nested loops

## **Space Complexity: O(N\*M)**

Reason: We are using an external array of size 'N\*M'. Stack Space is eliminated.

## **Space Optimization**

---

If we look closely, we need values from the previous row: dp[ind-1][ ]

So we are not required to contain an entire array, we can simply have two rows prev and cur where prev corresponds to dp[ind-1] and cur to dp[ind].

After declaring prev and cur, replace dp[ind-1] to prev and dp[ind] with cur and after the inner loop executes, we will set prev = cur, so that the cur row can serve as prev for the next index.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string &s1, string &s2){
    //      Write your code here.

    int n = s1.size();
    int m = s2.size();

    vector<int> prev(m+1, 0), cur(m+1, 0);

    int ans = 0;

    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(s1[i-1]==s2[j-1]){
                int val = 1 + prev[j-1];
                cur[j] = val;
                ans = max(ans, val);
            }
            else
                cur[j] = 0;
        }
        prev=cur;
    }

    return ans;
}

int main() {

    string s1= "abcjklp";
    string s2= "acjkp";

    cout<<"The Length of Longest Common Substring is "<<lcs(s1,s2);
}

```

### **Output:**

The Length of Longest Common Substring is 3

### **Time Complexity: O(N\*M)**

Reason: There are two nested loops.

### **Space Complexity: O(M)**

Reason: We are using an external array of size 'M+1' to store only two rows.

# Longest Palindromic Subsequence | (DP-28)

---

 [takeuforward.org/data-structure/longest-palindromic-subsequence-dp-28](https://takeuforward.org/data-structure/longest-palindromic-subsequence-dp-28)

March 23, 2022

**Problem Statement:** Longest Palindromic Subsequence

A palindromic string is a string that is equal to its reverse. For example: “nitin” is a palindromic string. Now the question states to find the length of the longest palindromic subsequence of a string. It is that palindromic subsequence of the given string with the greatest length.

**Example:**

str: “bbbab”

Longest Palindromic Subsequence: “bbbb”

There can be many subsequences like “b”, “ba”, “bbb” but “bbbb” is the subsequence that is a palindrome and has the greatest length.

We need to print the **length** of the longest palindromic subsequence.

**Problem Link:** [Longest Palindromic Subsequence](#)

**Solution :**

---

**Approach 1: Using Brute Force**

We are given a string S, the simplest approach will be to generate all the subsequences and store them, then manually find out the longest palindromic subsequence.

This naive approach will give us the correct answer but to generate all the subsequences, we will require **exponential (  $2^n$  )** time. Therefore we will try some other approaches.

**Approach 2: Using Dynamic Programming**

**Pre-req:** [Longest Common Subsequence](#)

We can use the approach discussed in the article [Longest Common Subsequence](#), to find the Longest Palindromic Subsequence.

**Intuition:**

Let us say that we are given the following string.

The longest palindromic subsequence will be:  
“babcbab”.

str: “bbabcabcab”

What is special about this string is that it is  
**palindromic (equal to its reverse) and of the longest length.**

Now let us write the reverse of str next to it and please think about the highlighted characters.

If we look closely at the highlighted characters, they are nothing but the longest common subsequence of the two strings.

str : “bbabcabcab”  
rev(str) : “bacbcbabb”

Now, we have taken the reverse of the string for the following two reasons:

- The longest palindromic subsequence being a palindrome will remain the same when the entire string is reversed.
- The length of the palindromic subsequence will also remain the same when the entire string is reversed.

From the above discussion we can conclude:

**The longest palindromic subsequence of a string is the longest common subsequence of the given string and its reverse.**

### Approach:

The algorithm is stated as follows:

- We are given a string (say s), make a copy of it and store it( say string t).
- Reverse the original string s.
- Find the longest common subsequence as discussed in [dp-25](#).

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, -1));
    for(int i=0;i<=n;i++){
        dp[i][0] = 0;
    }
    for(int i=0;i<=m;i++){
        dp[0][i] = 0;
    }

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                dp[ind1][ind2] = 1 + dp[ind1-1][ind2-1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1-1][ind2], dp[ind1][ind2-1]);
        }
    }

    return dp[n][m];
}

int longestPalindromeSubsequence(string s){
    string t = s;
    reverse(s.begin(), s.end());
    return lcs(s,t);
}

int main() {

    string s= "bbabcbcab";

    cout<<"The Length of Longest Palindromic Subsequence is "<<
    longestPalindromeSubsequence(s);
}

```

**Output:** The Length of Longest Palindromic Subsequence is 7

**Time Complexity:** O(N\*N)

Reason: There are two nested loops

**Space Complexity:** O(N\*N)

Reason: We are using an external array of size '(N\*N)'. Stack Space is eliminated.

## Space Optimization

---

If we closely we are using two rows: **dp[ind1-1][ ]**, **dp[ind][ ]**,

So we are not required to contain an entire array, we can simply have two rows prev and cur where prev corresponds to dp[ind-1] and cur to dp[ind].

After declaring prev and cur, replace dp[ind-1] to prev and dp[ind] with cur and after the inner loop executes, we will set prev = cur, so that the cur row can serve as prev for the next index.

### Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<int> prev(m+1,0), cur(m+1,0);

    // Base Case is covered as we have initialized the prev and cur to 0.

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                cur[ind2] = 1 + prev[ind2-1];
            else
                cur[ind2] = 0 + max(prev[ind2],cur[ind2-1]);
        }
        prev= cur;
    }

    return prev[m];
}

int longestPalindromeSubsequence(string s){
    string t = s;
    reverse(s.begin(),s.end());
    return lcs(s,t);
}

int main() {

    string s= "bbabcbcab";

    cout<<"The Length of Longest Palindromic Subsequence is "<<
    longestPalindromeSubsequence(s);
}
```

**Output:**

The Length of Longest Palindromic Subsequence is 7

**Time Complexity: O(N\*N)**

Reason: There are two nested loops.

**Space Complexity: O(N)**

Reason: We are using an external array of size 'N+1' to store only two rows.

# Minimum insertions to make string palindrome | DP-29

 [takeuforward.org/data-structure/minimum-insertions-to-make-string-palindrome-dp-29](https://takeuforward.org/data-structure/minimum-insertions-to-make-string-palindrome-dp-29)

March 27, 2022

**Problem Statement:** Minimum insertions required to make a string palindrome

A palindromic string is a string that is the same as its reverse. For example: “nitin” is a palindromic string. Now the question states that we are given a string, we need to find the minimum insertions that we can make in that string to make it a palindrome.

**Example:**

str: “abcaa”

Minimum Insertions required to make str palindrome: 2

“abcaa”  “abca**cba**”

2 Insertions

**Problem Link:** [Minimum Insertions to make a string palindrome](#)

**Solution :**

**Pre-req:** [Longest Common Subsequence](#), [Longest Palindromic Subsequence](#).

**Intuition:**

We need to find the minimum insertions required to make a string palindrome. Let us keep the “minimum” criteria aside and think, how can we make any given string palindrome by inserting characters?

The easiest way is to add the reverse of the string at the back of the original string as shown below. This will make any string palindrome.

str: "abcaa"

Palindromic string

abcaa **aacba** ←

Inserting the reverse of the string at the end  
of the original string to make it palindrome

Here the number of characters inserted will be equal to n (length of the string). This is the maximum number of characters we can insert to make strings palindrome.

The problem states us to find the **minimum** of insertions. Let us try to figure it out:

To minimize the insertions, we will first try to refrain from adding those characters again which are already making the given string palindrome. For the given example, "aaa", "aba", "aca", any of these are themselves palindromic components of the string. We can take any of them( as all are of equal length) and keep them intact. (let's say "aaa").

Now, there are two characters('b' and 'c') remaining which prevent the string from being a palindrome. We can reverse their order and add them to the string to make the entire string palindrome.

str: "abcaa"

Keep "aaa" intact as it is itself palindromic.

abcaa

str: "abcaa"

Keep "aaa" intact as it is itself palindromic.

Adding Reverse of the remaining characters to the string

ab**cacba**

We can do this by taking some other components (like "aca") as well.

In order to minimize the insertions, we need to find the length of the longest palindromic component or in other words, the longest palindromic subsequence.

**Minimum Insertion required =  $n(\text{length of the string}) - \text{length of longest palindromic subsequence.}$**

**Approach:**

The algorithm is stated as follows:

- We are given a string (say s), store its length as n.
- Find the length of the longest palindromic subsequence ( say k) as discussed in [dp\\_28](#)
- Return n-k as answer.

**Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, -1));
    for(int i=0;i<=n;i++){
        dp[i][0] = 0;
    }
    for(int i=0;i<=m;i++){
        dp[0][i] = 0;
    }

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                dp[ind1][ind2] = 1 + dp[ind1-1][ind2-1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1-1][ind2],dp[ind1][ind2-1]);
        }
    }

    return dp[n][m];
}

int longestPalindromeSubsequence(string s){
    string t = s;
    reverse(s.begin(),s.end());
    return lcs(s,t);
}

int minInsertion(string s){
    int n = s.size();
    int k = longestPalindromeSubsequence(s);
    return n-k;
}

int main() {

    string s= "abcaa";
    cout<<"The Minimum insertions required to make string palindrome: "<<
    minInsertion(s);
}

```

### **Output:**

The Minimum insertions required to make string palindrome: 2

**Time Complexity: O(N\*N)**

Reason: There are two nested loops

## **Space Complexity: O(N\*N)**

Reason: We are using an external array of size (N\*N). Stack Space is eliminated.

## **Space Optimization**

---

If we closely we are using two rows: **dp[ind-1][ ]**, **dp[ind][ ]**,

So we are not required to contain an entire array, we can simply have two rows prev and cur where prev corresponds to dp[ind-1] and cur to dp[ind].

After declaring prev and cur, replace dp[ind-1] to prev and dp[ind] with cur and after the inner loop executes, we will set prev = cur, so that the cur row can serve as prev for the next index.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;
int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<int> prev(m+1, 0), cur(m+1, 0);

    // Base Case is covered as we have initialized the prev and cur to 0.

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                cur[ind2] = 1 + prev[ind2-1];
            else
                cur[ind2] = 0 + max(prev[ind2],cur[ind2-1]);
        }
        prev= cur;
    }

    return prev[m];
}

int longestPalindromeSubsequence(string s){
    string t = s;
    reverse(s.begin(),s.end());
    return lcs(s,t);
}

int minInsertion(string s){
    int n = s.size();
    int k = longestPalindromeSubsequence(s);
    return n-k;
}

int main() {

    string s= "abcaa";

    cout<<"The Minimum insertions required to make string palindrome: "<<
    minInsertion(s);
}

```

### **Output:**

The Minimum insertions required to make string palindrome: 2

**Time Complexity: O(N\*M)**

Reason: There are two nested loops.

## **Space Complexity: O(M)**

Reason: We are using an external array of size 'M+1' to store only two rows.

# Minimum Insertions/Deletions to Convert String | (DP-30)

---

 [takeuforward.org/data-structure/minimum-insertions-deletions-to-convert-string-dp-30](https://takeuforward.org/data-structure/minimum-insertions-deletions-to-convert-string-dp-30)

March 27, 2022

**Problem Statement:** Minimum Insertions/Deletions to Convert String A to String B

We are given two strings, str1 and str2. We are allowed the following operations:

- Delete any number of characters from string str1.
- Insert any number of characters in string str1.

We need to tell the minimum operations required to convert str1 to str2.

**Example:**

str1: "abcd"

str2: "anc"

Minimum Insertions/Deletions to convert str1 to str2 : 3(2+1)

Delete 2 characters from str1 : abcd

Insert 1 character to str1 : anc

**Problem Link: Can You Make**

**Solution :**

---

**Pre-req:** Longest Common Subsequence

**Intuition:**

---

We need to find the minimum operations required to convert string str1 to str2. Let us keep the “minimum” criteria aside and think, what maximum operations will be required for this conversion?

The easiest way is to remove all the characters of str1 and then insert all the characters of str2. In this way, we will convert str1 to str2 with ‘n+m’ operations. (Here n and m are the length of strings str1 and str2 respectively).

str1: "abcd"

Delete all characters from str1 : ~~abcd~~ 4 operations

Insert all characters of str2 to str1 : anc 3 operations

Number of Insertions required: 7 (4+3)

The problem states us to find the **minimum** of insertions. Let us try to figure it out:

To minimize the operations, we will first try to refrain from deleting those characters which are already present in str2. More extensively, we refrain from deleting those characters which are common and come in the same order. To minimize the operations, we would like to keep the maximum common characters coming in the same order intact. These maximum characters are the characters of the **longest common subsequence**.

We will first keep the longest common subsequence of the str1 and str2 intact in str1 and delete all other characters from str1.

str1: "abcd" str2: "anc"

Longest common subsequence: "ac"

Keeping lcs intact and deleting all other characters from str1: abcd

## 2 operations

Next, we will insert all the remaining characters of str2 to str1.

str1: "abcd"

str2: "anc"

Longest common subsequence: "ac"

Keeping lcs intact and deleting all other characters from str1: abcd

2 operations

Inserting the remaining characters in str1:

anc

1 operation

Minimum operations required: 3(2+1)

In order to minimize the operations, we need to find the length of the longest common subsequence.

**Minimum Operations required =  $(n - k) + (m - k)$**

Here n and m are the length of str1 and str2 respectively and k is the length of the longest common subsequence of str1 and str2.

### Approach:

The algorithm is stated as follows:

- Let n and m be the length of str1 and str2 respectively.
- Find the length of the longest common subsequence ( say k ) of str1 and str2 as discussed in [Longest Common Subsequence](#).
- Return  $(n-k) + (m-k)$  as answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, -1));
    for(int i=0;i<=n;i++){
        dp[i][0] = 0;
    }
    for(int i=0;i<=m;i++){
        dp[0][i] = 0;
    }

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                dp[ind1][ind2] = 1 + dp[ind1-1][ind2-1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1-1][ind2],dp[ind1][ind2-1]);
        }
    }

    return dp[n][m];
}

int canYouMake(string str1, string str2){

    int n = str1.size();
    int m = str2.size();

    int k = lcs(str1,str2);

    return (n-k)+(m-k);
}

int main() {

    string str1= "abcd";
    string str2= "anc";

    cout<<"The Minimum operations required to convert str1 to str2: "<<
    canYouMake(str1,str2);
}

```

### **Output:**

The Minimum operations required to convert str1 to str2: 3

**Time Complexity: O(N\*M)**

Reason: There are two nested loops

## **Space Complexity: O(N\*M)**

Reason: We are using an external array of size (N\*M). Stack Space is eliminated.

## **Space Optimization**

If we closely we are using two rows: **dp[ind1-1][ ]**, **dp[ind][ ]**,

So we are not required to contain an entire array, we can simply have two rows prev and cur where prev corresponds to dp[ind-1] and cur to dp[ind].

After declaring prev and cur, replace dp[ind-1] to prev and dp[ind] with cur and after the inner loop executes, we will set prev = cur, so that the cur row can serve as prev for the next index.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int lcs(string s1, string s2) {

    int n=s1.size();
    int m=s2.size();

    vector<int> prev(m+1,0), cur(m+1,0);

    // Base Case is covered as we have initialized the prev and cur to 0.

    for(int ind1=1;ind1<=n;ind1++){
        for(int ind2=1;ind2<=m;ind2++){
            if(s1[ind1-1]==s2[ind2-1])
                cur[ind2] = 1 + prev[ind2-1];
            else
                cur[ind2] = 0 + max(prev[ind2],cur[ind2-1]);
        }
        prev= cur;
    }

    return prev[m];
}

int canYouMake(string str1, string str2){

    int n = str1.size();
    int m = str2.size();

    int k = lcs(str1,str2);

    return (n-k)+(m-k);
}

int main() {

    string str1= "abcd";
    string str2= "anc";

    cout<<"The Minimum operations required to convert str1 to str2: "<<
    canYouMake(str1,str2);
}

```

### **Output:**

The Minimum operations required to convert str1 to str2: 3

### **Time Complexity: O(N\*M)**

Reason: There are two nested loops.

### **Space Complexity: O(M)**

Reason: We are using an external array of size ‘M+1’ to store only two rows.

# Shortest Common Supersequence | (DP – 31)

 [takeuforward.org/data-structure/shortest-common-supersequence-dp-31](https://takeuforward.org/data-structure/shortest-common-supersequence-dp-31)

March 31, 2022

**Problem Statement:** Shortest Common Supersequence

We are given two strings ‘S1’ and ‘S2’. We need to return their shortest common supersequence. A supersequence is defined as the string which contains both the strings S1 and S2 as subsequences.

**Example:**

S1: “brute”

S2: “groot”

Shortest Common Supersequence: **bgruote**



We need to return the length of the longest common subsequence and the string as well.

**Problem Link:** [Shortest Supersequence](#)

**Solution :**

**Pre-req:** [Longest Common Subsequence](#), [Print Longest Common Subsequence](#)

**Intuition:**

If we keep the “shortest” criteria aside, what can be a way to generate a supersequence given two strings. One easy way is to concat the given strings (write one after the other), this will always give us a supersequence for any pair of given strings.

S1: "brute"

S2: "groot"

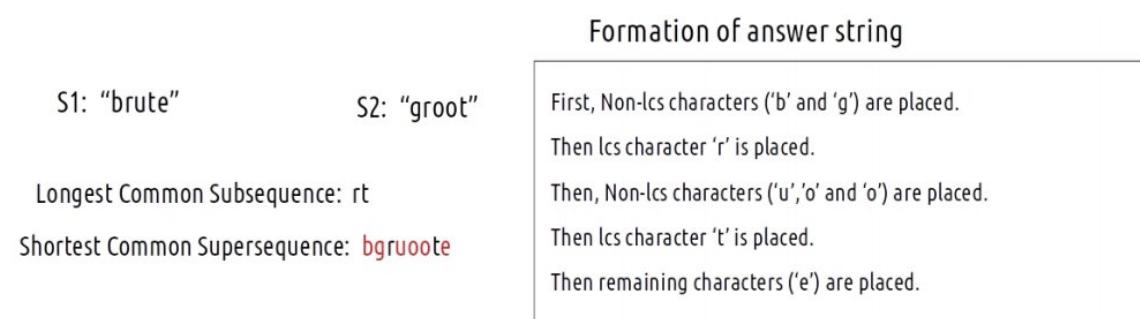
Supersequence(concatenating both Strings): **brutegroot**

This can be said as the worst case with time complexity of  $O(n+m)$ , where n and m are the lengths of strings S1 and S2 respectively.

How can we improve from this naive approach?

If we think a little, there are some common characters that we can avoid writing for both the strings separately. These common characters can't be all the common characters. They are the characters that **are common and come in the same order**. In other words, they are the characters of the **longest common subsequence**.

In an optimum solution, the characters of the longest common subsequence are written only once and other characters are placed around them. For every character that belongs to the longest common subsequence, the non-lcs characters coming before them in the strings S1 and S2 are placed before the lcs-character in the answer string. The below figure explains this:



## Length of Shortest Common Supersequence?

From the explanation above, we can see that characters of lcs need to be covered only once. Therefore, the length of the shortest Common supersequence =  $n + m - k$ , where (n and m are lengths of S1 and S2, and k is the length of the lcs string).

## Finding the supersequence string

Now, instead of the length, we are interested in finding the shortest supersequence string itself. Readers are highly advised to read the article [Print Longest Common Subsequence](#).

## Intuition:

When we form the DP table to calculate the longest common subsequence (as done in [Print Longest Common Subsequence](#)) we have all the information of characters that are coming in the lcs string and characters that don't. We use this same DP table to form the shortest common supersequence.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	'g'	'r'	'o'	'o'	't'	
2	'b'	0	0	0	0	0
3	'r'	0	1	1	1	1
4	'u'	0	1	1	1	1
5	't'	0	1	1	1	2
6	'e'	0	1	1	1	2

To frame the string, we need to understand how the dp table was formed and work in the reverse process.

Now, let us see what were the conditions that we used while forming the dp array:

- **if( $S1[i-1] == S2[j-1]$ )**, then return  $1 + dp[i-1][j-1]$
- **if( $S1[i-1] != S2[j-1]$ )**, then return  $0 + \max(dp[i-1][j], dp[i][j-1])$

## Approach:

We will start from the right-most cell of the dp array, initially  $i=n$  and  $j=m$

To form the string, we will work in a reverse manner.

- **if( $S1[i-1] == S2[j-1]$ )**, this means the character is an lcs character and needs to be included only once from both the strings, so we add it to the ans string and reduce both  $i$  and  $j$  by 1. We reduce them simultaneously to make sure the character is counted only once.

- if( $S_1[i-1] \neq S_2[j-1]$ ), this means that the character is a non-lcs character and then we move the pointer to the top cell or left cell depending on which is greater. This way non-lcs characters will be included separately in the right order.

The algorithm steps are stated below:

- We start from cell  $dp[n][m]$ . Initially  $i=n$  and  $j=m$ .
  - At every cell, we will check if  $S1[i-1] == S2[j-1]$ , if it is then it means this character is a part of the longest common subsequence. So we will push it to the ans string str. Then we will move to the diagonally top-left() cell by assigning  $i$  to  $i-1$  and  $j$  to  $j-1$ .
  - Else, this character is not a part of the longest common subsequence so we include it in ans string. Originally this cell got its value from its left cell ( $\leftarrow$ ) or from its top cell ( $\uparrow$ ). Whichever cell's value will be more of the two, we will move to that cell.
  - We will continue till  $i>0$  and  $j>0$ , failing it we will break from the loop.
  - After breaking, either  $i>0$  or  $j>0$  (only one condition will fail to break from the while loop), if( $i>0$ ) we push all the characters from  $S1$  to ans string, else if( $j>0$ ), we push all the remaining characters from  $S2$ .
  - At last, we reverse the ‘ans’ string and we get our answer.

## Dry Run:

At starting i=5 and j=5.

		(i)				
		0	1	2	3	4
0	0	0	0	0	0	0
	0	'g'	'r'	'o'	'o'	't'
	0	'b'	0	0	0	0
	0	'r'	0	1	1	1
	0	'u'	0	1	1	1
	0	't'	0	1	1	2
5	0	'e'	0	1	1	2

1000

515.

S1: "brute"		S2: "groot"			
		(ii)			
0	1	2	3	4	5
0	0	0	0	0	0
1	'g'	'r'	'o'	'o'	't'
2	'b'	0	0	0	0
3	'r'	0	1	1	1
4	'u'	0	1	1	1
5	't'	0	1	1	2
6	'e'	0	1	1	2

S1[3] == S2[4]

ans: et

(i) As  $S1[4] \neq S2[4]$ , we move to the top cell( $\uparrow$ ) as its value is greater than the left cell( $\leftarrow$ ) but before moving as we are leaving row 5( $i=5$ ) and will not return to it so we add  $S1[4]$  to our ans string.

(ii) As  $S1[3] == S2[4]$ , this is an lcs-character. We add the current character to the ans string (and move to  $i-1$  and  $j-1$  cell) i.e top-left(). **Reducing i and j simultaneously helps in adding the lcs character only once in the ans string.**

S1: "brute"		S2: "groot"	
		(iii)	
0	0	'g'	'r'
1	0	'o'	'o'
2	0	't'	't'
3	0	'e'	
4	0		
5	0		

S1[2] != S2[3]

ans: **eto**

S1: "brute"		S2: "groot"	
		(iv)	
0	0	'g'	'r'
1	0	'o'	'o'
2	0	't'	't'
3	0	'e'	
4	0		
5	0		

S1[2] != S2[2]

ans: **etoo**

(iii) As S1[2] != S2[3], we move to the left cell(←) as its value is greater than or equal to the top cell(↑) but before moving as we are leaving column 4 (j=4) and will not return to it so we add S2[3] to our ans string.

(iv) As S1[2] != S2[2], we move to the left cell(←) as its value is greater than or equal to the top cell(↑) but before moving as we are leaving column 3 (j=3) and will not return to it so we add S2[2] to our ans string.

S1: "brute"		S2: "groot"	
		(v)	
0	0	'g'	'r'
1	0	'o'	'o'
2	0	't'	't'
3	0	'e'	
4	0		
5	0		

S1[2] != S2[1]

ans: **etoo**

S1: "brute"		S2: "groot"	
		(vi)	
0	0	'g'	'r'
1	0	'o'	'o'
2	0	't'	't'
3	0	'e'	
4	0		
5	0		

S1[1] == S2[1]

ans: **etour**

(v) As S1[2] != S2[1], we move to the top cell(↑) as its value is greater than the left cell(←) but before moving as we are leaving row 3(i=3) and will not return to it so we add S1[2] to our ans string.

(vi) As S1[1] == S2[1], this is an lcs-character. We add the current character to the ans string(and move to i-1 and j-1 cell) i.e top-left().

S1: "brute"

S2: "groot"

(vii)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'g'	'r'	'o'	'o'	't'
2	0	'b'	0	0	0	0
3	0	'r'	0	1	1	1
4	0	'u'	0	1	1	1
5	0	't'	0	1	1	2
6	0	'e'	0	1	1	2

S1[0] != S2[0]

ans: etoourg

S1: "brute"

S2: "groot"

(viii)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	'g'	'r'	'o'	'o'	't'
2	0	'b'	0	0	0	0
3	0	'r'	0	1	1	1
4	0	'u'	0	1	1	1
5	0	't'	0	1	1	2
6	0	'e'	0	1	1	2

Break while loop and add remaining characters to ans String

ans: etoourgb

(vii) As S1[2] != S2[2], we move to the left cell(←) as its value is greater than or equal to the top cell(↑) but before moving as we are leaving column 1 ( $j=1$ ) and will not return to it so we add S2[0] to our ans string.

(viii) As  $j=0$ , we have reached the exit condition of the while loop, so we will break from it but still there are some characters left in the other string. We will simply add them in the ans string.

(ix) At last, we will return the **reverse of the ans string** as the final answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

string shortestSupersequence(string s1, string s2){

    int n = s1.size();
    int m = s2.size();

    vector < vector < int >> dp(n + 1, vector < int > (m + 1, 0));
    for (int i = 0; i <= n; i++) {
        dp[i][0] = 0;
    }
    for (int i = 0; i <= m; i++) {
        dp[0][i] = 0;
    }

    for (int ind1 = 1; ind1 <= n; ind1++) {
        for (int ind2 = 1; ind2 <= m; ind2++) {
            if (s1[ind1 - 1] == s2[ind2 - 1])
                dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
            else
                dp[ind1][ind2] = 0 + max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
        }
    }

    int len = dp[n][m];
    int i = n;
    int j = m;

    int index = len - 1;
    string ans = "";

    while (i > 0 && j > 0) {
        if (s1[i - 1] == s2[j - 1]) {
            ans += s1[i-1];
            index--;
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            ans += s1[i-1];
            i--;
        } else {
            ans += s2[j-1];
            j--;
        }
    }

    //Adding Remaining Characters - Only one of the below two while loops will run

    while(i>0){
        ans += s1[i-1];
        i--;
    }
    while(j>0){
        ans += s2[j-1];
    }
}

```

```
        j--;
    }

    reverse(ans.begin(),ans.end());

    return ans;
}

int main() {

    string s1 = "brute";
    string s2 = "groot";

    cout << "The Longest Common Supersequence is "<<shortestSupersequence(s1,s2);
}
```

### **Output:**

The Longest Common Supersequence is bgruoote

### **Time Complexity: O(N\*M)**

Reason: There are two nested loops

### **Space Complexity: O(N\*M)**

Reason: We are using an external array of size (N\*M).

# Distinct Subsequences| (DP-32)

---

 [takeuforward.org/data-structure/distinct-subsequences-dp-32](https://takeuforward.org/data-structure/distinct-subsequences-dp-32)

March 31, 2022

**Problem Statement:** Distinct Subsequences

**Problem Link:** [Subsequence Counting](#)

We are given two strings S1 and S2, we want to know how many distinct subsequences of S2 are present in S1.

**Example:**

S1: "babgbag"

S2: "bag"

Distinct Subsequences: 5

String S2("bag") is present as 5 distinct subsequences.

babgbag      babgbag      babgbag

babgbag      babgbag

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

**Note:** Please modulo the answer if asked.

**Solution :**

---

We have to find distinct subsequences of S2 in S1. As there is no uniformity in data, there is no other way to find out than to **try out all possible ways**. To do so we will need to use **recursion**.

**Steps to form the recursive solution:**

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two strings. We can represent them with the help of two indexes i and j. Initially,  $i=n-1$  and  $j=m-1$ , where n and m are lengths of strings S1 and S2. Initially, we will call  $f(n-1, j-1)$ , which means the count of all subsequences of string  $S2[0...m-1]$  in string  $S1[0...n-1]$ . We can generalize it as follows:

**$f(i,j) \rightarrow$  Count of all distinct subsequences of  $S2[0...j-1]$  in the string  $S1[0...i-1]$ .**

**Step 2:** Try out all possible choices at a given index.

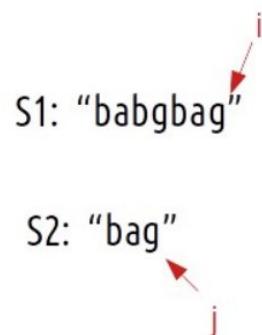
Now, i and j represent two characters from strings S1 and S2 respectively. We want to find distinct subsequences. There are only two options that make sense: either the characters represented by i and j match or they don't.

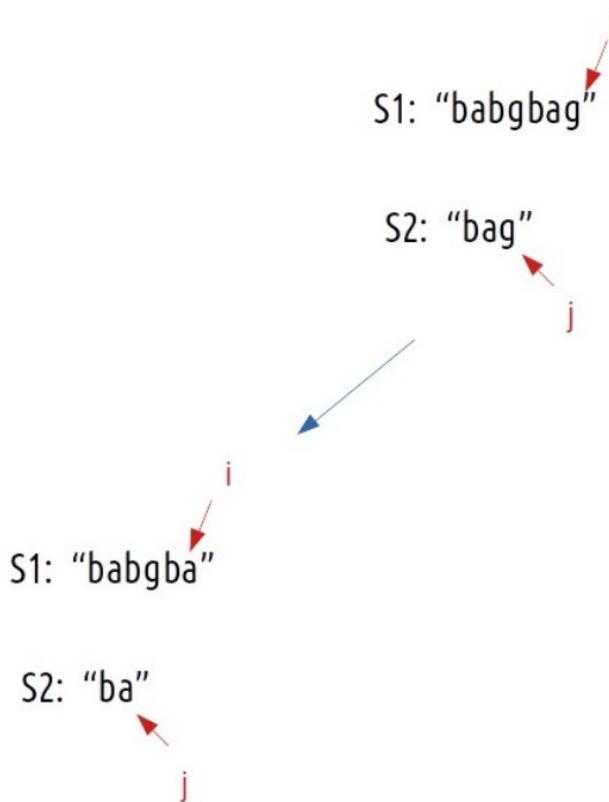
### **Case 1: When the characters match**

**if( $S1[i]==S2[j]$ ),** let's understand it with the following example:

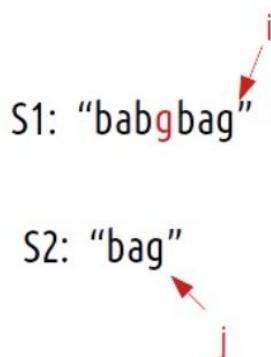
$S1[i] == S2[j]$ , now as the characters at i and j match, we would want to check the possibility of the remaining characters of S2 in S1 therefore we reduce the length of both the strings by 1 and call the function recursively.

S1: "babgbag"  
S2: "bag"



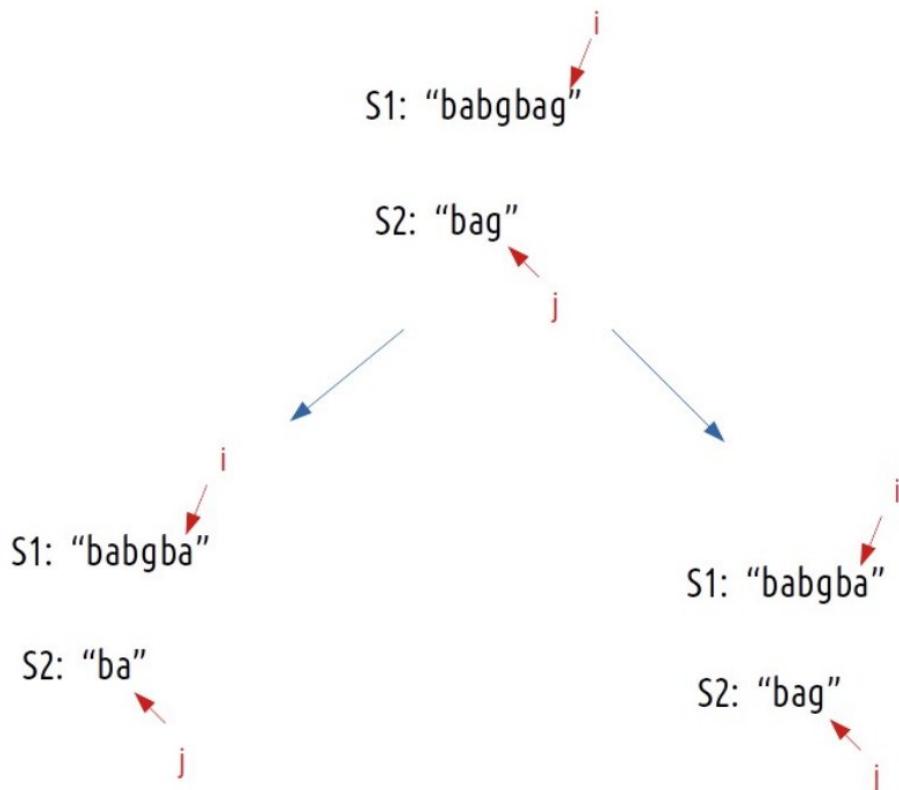


Now, if we only make the above single recursive call, we are rejecting the opportunities to find more than one subsequences because it can happen that the  $j$ th character may match with more characters in  $S1[0...i-1]$ , for example where there are more occurrences of 'g' in  $S1$  from which also an answer needs to be explored.



**Answer from g (marked in red) also needs to be explored**

To explore all such possibilities, we make another recursive call in which we reduce the length of the  $S1$  string by 1 but keep the  $S2$  string the same, i.e we call  $f(i-1,j)$ .



### Case 2: When the characters don't match

**if( $S1[i] \neq S2[j]$ )**, it means that we don't have any other choice than to try the next character of  $S1$  and match it with the current character  $S2$ .

This can be summarized as :

- **if( $S1[i]==S2[j]$ )**, call  $f(i-1,j-1)$  and  $f(i-1,j)$ .
- **if( $S1[i]\neq S2[j]$ )**, call  $f(i-1,j)$ .

### Step 3: Return the sum of choices

As we have to return the total count, we will return the **sum** of  $f(i-1,j-1)$  and  $f(i-1,j)$  in case 1 and simply return  $f(i-1,j)$  in case 2.

### Base Cases:

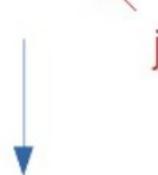
We are reducing  $i$  and  $j$  in our recursive relation, there can be two possibilities, either  $i$  becomes -1 or  $j$  becomes -1.

- if  $j < 0$ , it means we have matched all characters of  $S2$  with characters of  $S1$ , so we return 1.
- if  $i < 0$ , it means we have checked all characters of  $S1$  but we are not able to match all characters of  $S2$ , therefore we return 0.

The final pseudocode after steps 1, 2, and 3:

S1: "babgbak"  
i

S2: "bag"



S1: "babgbba"  
i

S2: "bag"  
j

```

f(i,j) {

    if( j<0)
        return 1
    if(i<0)
        return 0

    if( S1[i] == S2[j]){
        return f(i-1,j-1) + f(i-1,j)

    else
        return f(i-1,j)

}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size  $[n][m]$ . The size of  $S_1$  and  $S_2$  are  $n$  and  $m$  respectively, so the variable  $i$  will always lie between ‘0’ and ‘ $n-1$ ’ and the variable  $j$  between ‘0’ and ‘ $m-1$ ’.
2. We initialize the dp array to -1.
3. Whenever we want to find the answer to particular parameters (say  $f(i,j)$ ), we first check whether the answer is already calculated using the dp array(i.e  $dp[i][j] \neq -1$  ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set  $dp[i][j]$  to the solution we get.

### **Code:**

- C++Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int prime = 1e9+7;

int countUtil(string s1, string s2, int ind1, int ind2, vector<vector<int>>& dp) {
    if(ind2<0)
        return 1;
    if(ind1<0)
        return 0;

    if(dp[ind1][ind2]!=-1)
        return dp[ind1][ind2];

    if(s1[ind1]==s2[ind2]){
        int leaveOne = countUtil(s1,s2,ind1-1,ind2-1,dp);
        int stay = countUtil(s1,s2,ind1-1,ind2,dp);

        return dp[ind1][ind2] = (leaveOne + stay)%prime;
    }

    else{
        return dp[ind1][ind2] = countUtil(s1,s2,ind1-1,ind2,dp);
    }
}

int subsequenceCounting(string &t, string &s, int lt, int ls) {
    // Write your code here.

    vector<vector<int>> dp(lt, vector<int>(ls, -1));
    return countUtil(t,s,lt-1,ls-1,dp);
}

int main() {

    string s1 = "babgbag";
    string s2 = "bag";

    cout << "The Count of Distinct Subsequences is "
    << subsequenceCounting(s1,s2,s1.size(),s2.size());
}

```

### **Output:**

The Count of Distinct Subsequences is 5

### **Time Complexity: O(N\*M)**

Reason: There are N\*M states therefore at max 'N\*M' new problems will be solved.

### **Space Complexity: O(N\*M) + O(N+M)**

Reason: We are using a recursion stack space(O(N+M)) and a 2D array ( O(N\*M)).

## **Steps to convert Recursive Solution to Tabulation one.**

---

In the recursive logic, we set the base case too if( $i < 0$ ) and if( $j < 0$ ) but we can't set the dp array's index to -1. Therefore a hack for this issue is to shift every index by 1 towards the right.

Recursive code indexes:      -1, 0, 1, ..., n

Shifted indexes :      0, 1, ..., n+1

- First we initialize the dp array of size  $[n+1][m+1]$  as zero.
- Next, we set the base condition (keep in mind 1-based indexing), we set the first column's value as 1 and the first row as 1.
- Similarly, we will implement the recursive code by keeping in mind the shifting of indexes, therefore  $S1[i]$  will be converted to  $S1[i-1]$ . Same for  $S2$ .
- At last, we will print  $dp[N][M]$  as our answer.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int prime = 1e9+7;

int subsequenceCounting(string &s1, string &s2, int n, int m) {
    // Write your code here.

    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    for(int i=0;i<n+1;i++){
        dp[i][0]=1;
    }
    for(int i=1;i<m+1;i++){
        dp[0][i]=0;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<m+1;j++){

            if(s1[i-1]==s2[j-1])
                dp[i][j] = (dp[i-1][j-1] + dp[i-1][j])%prime;
            else
                dp[i][j] = dp[i-1][j];
        }
    }

    return dp[n][m];
}

```

```

int main() {

    string s1 = "babgbag";
    string s2 = "bag";

    cout << "The Count of Distinct Subsequences is " <<
    subsequenceCounting(s1,s2,s1.size(),s2.size());
}

```

### **Output:**

The Count of Distinct Subsequences is 5

### **Time Complexity: O(N\*M)**

Reason: There are two nested loops

### **Space Complexity: O(N\*M)**

Reason: We are using an external array of size 'N\*M'. Stack Space is eliminated.

### Part 3: Space Optimization

---

If we closely look the relation,

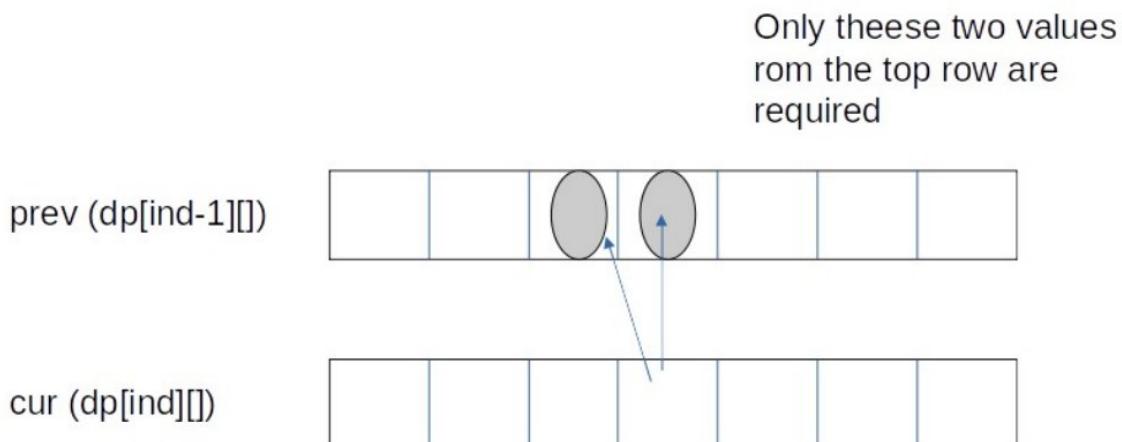
$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j] \text{ or } dp[i][j] = dp[i-1][j]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

We will be space optimizing this solution using **only one row**.

#### Intuition:

If we clearly see the values required:  $dp[i-1][j-1]$  and  $dp[i-1][j]$ , we can say that if we are at a column  $j$ , we will only require the values shown in the grey box from the previous row and other values will be from the cur row itself. So why do we need to store an entire array for it?



If we need only two values from the prev row, there is no need to store an entire row. We can work a bit smarter.

We can use the cur row itself to store the required value in the following way:

- We take a single row 'prev'.
- We initialize it to the base condition.
- Whenever we want to compute a value of the cell  $prev[j]$ , we take the already existing value ( $prev[j]$  before new computation) and  $prev[j-1]$  (if required, in case of character match).
- We perform the above step on all the indexes.
- So we see how we can space optimize using a single row itself.

#### Code:

- C++ Code

- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int prime = 1e9+7;

int subsequenceCounting(string &s1, string &s2, int n, int m) {
    // Write your code here.

    vector<int> prev(m+1, 0);

    prev[0]=1;

    for(int i=1;i<n+1;i++){
        for(int j=m;j>=1;j--){ // Reverse direction

            if(s1[i-1]==s2[j-1])
                prev[j] = (prev[j-1] + prev[j])%prime;
            else
                prev[j] = prev[j]; //can omit this statement
        }
    }

    return prev[m];
}

int main() {

    string s1 = "babgbag";
    string s2 = "bag";

    cout << "The Count of Distinct Subsequences is " <<
    subsequenceCounting(s1,s2,s1.size(),s2.size());
}
```

**Output:**

The Count of Distinct Subsequences is 5

**Time Complexity: O(N\*M)**

Reason: There are two nested loops.

**Space Complexity: O(M)**

Reason: We are using an external array of size 'M+1' to store only one row.

# Edit Distance | (DP-33)

---

 [takeuforward.org/data-structure/edit-distance-dp-33](https://takeuforward.org/data-structure/edit-distance-dp-33)

April 10, 2022

## Problem Statement: Edit Distance

We are given two strings 'S1' and 'S2'. We need to convert S1 to S2. The following three operations are allowed:

1. Deletion of a character.
2. Replacement of a character with another one.
3. Insertion of a character.

We have to return the **minimum** number of operations required to convert S1 to S2 as our answer.

**Example:**

S1: "horse"

S2: "ros"

Minimum number of operations required: 3

Step 1: Replace 'h' at index 0 with 'r' of S1.

rorse

Step 2: Delete 'r' at index 2 of S1.

~~r~~orse

Step 3: Replace 'e' at index 4 of S1.

~~r~~ose

## Problem Link: [Shortest Supersequence](#)

**Disclaimer:** Don't jump directly to the solution, try it out yourself first.

## Intuition:

---

For every index of string S1, we have three options to match that index with string S2, i.e replace the character, remove the character or insert some character at that index.

Therefore, we can think in terms of string matching path as we have done already in previous questions.

As there is no uniformity in data, there is no other way to find out than to **try out all possible ways**. To do so we will need to use **recursion**.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in the Dynamic Programming Introduction.

**Step 1:** Express the problem in terms of indexes.

We are given two strings. We can represent them with the help of two indexes i and j. Initially,  $i=n-1$  and  $j=m-1$ , where n and m are lengths of strings S1 and S2. Initially, we will call  $f(n-1, m-1)$ , which means the minimum number of operations required to convert string  $S1[0...n-1]$  to string  $S2[0...m-1]$ .

We can generalize this as follows:

**$f(i,j) \rightarrow$  Minimum number of operations required to convert string  $S1[0...i]$  to  $S2[0...j]$  using three given operations.**

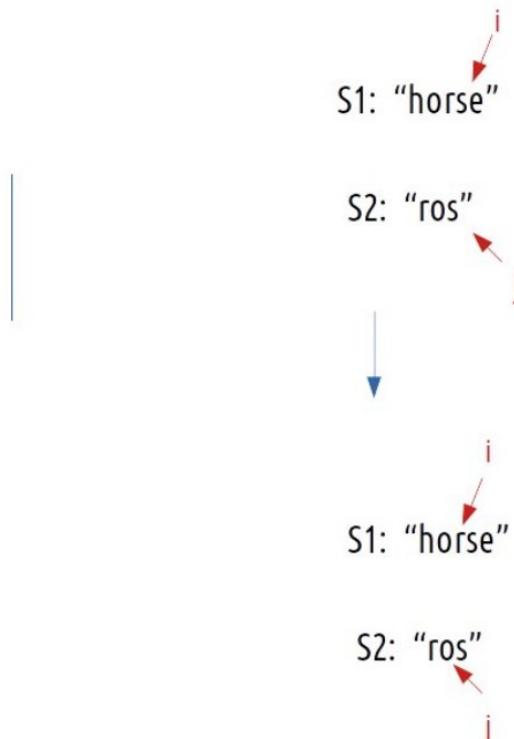
**Step 2:** Try out all possible choices at a given index.

Now, i and j represent two characters from strings S1 and S2 respectively. There are only two options that make sense: either the characters represented by i and j match or they don't.

#### (i) When the characters match

**if( $S1[i]==S2[j]$ ),**

If this is true, now as the characters at i and j match, we would not want to do any operations to make them match, so we will just decrement both i and j by 1 and recursively find the answer for the remaining string portion. We return  **$0+f(i-1,j-1)$** . The following figure makes it clear.



## (ii) When the characters don't match

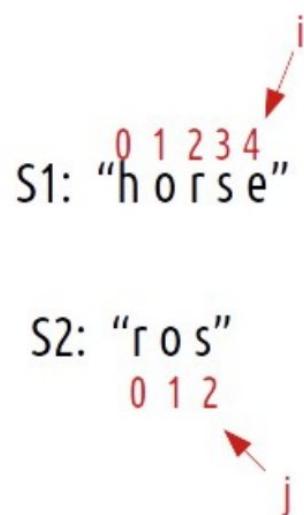
**if( $S1[i] \neq S2[j]$ )** is true, then we have to do any of three operations to match the characters. We have three options, we will analyze each of them one by one.

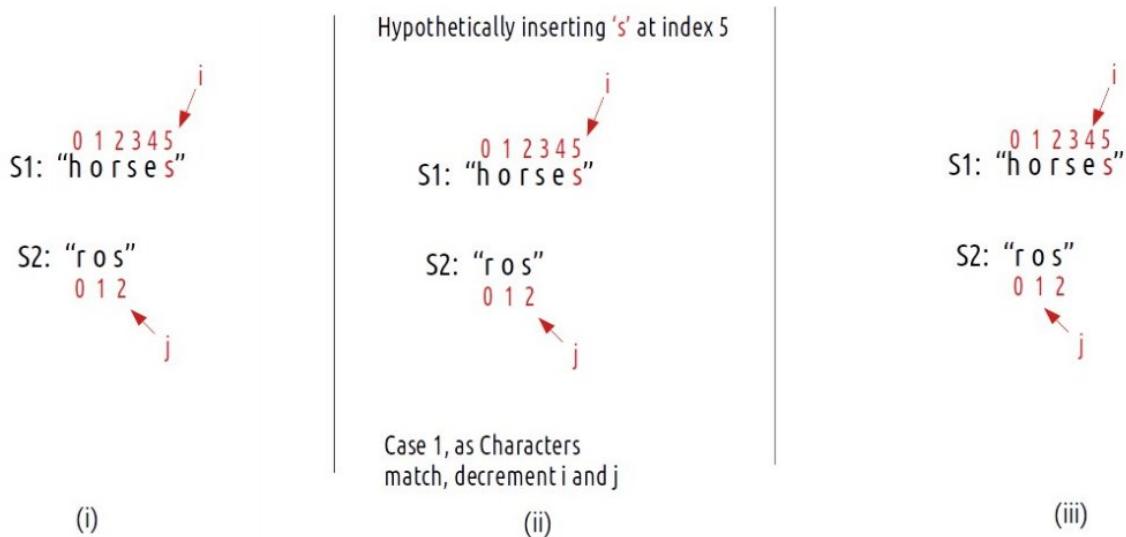
### Case 1: Inserting a character

Consider this example,

Now if we have to match the strings by insertions, what would we do?:

- We would have placed an 's' at index 5 of S1.
- Suppose i now point to s at index 5 of S1 and j points are already pointing to s at index j of S2.
- Now, we hit the condition, where characters do match. (as mentioned in case 1).
- Therefore, we will decrement i and j by 1. They will now point to index 4 and 1 respectively.





Now, the number of operations we did were only 1 (inserting **s** at index 5) but do we need to really insert the '**s**' at index 5 and modify the string? The answer is simply **NO**. As we see that inserting a character (here '**s**' at index 5), we will eventually get to the third step. So we can just **return  $1 + f(i, j-1)$**  as **i** remains there only after insertion and **j** decrements by 1. We can say that we have **hypothetically** inserted character **s**.

### Case 2: Deleting a character

Consider the same example,

We can simply delete the character at index 4 and check from the next index.

Now, **j** remains at its original index and we decrement **i** by 1. We perform 1 operation, therefore we will recursively call  **$1 + f(i-1, j)$** .

### Case 3: Replacing a character

Consider the same example,

If we replace the character '**e**' at index 4 of **S1** with '**s**', we have matched both the characters ourselves. We again hit the case of character matching, therefore we decrement **both i and j** by 1. As the number of operations performed is 1, we will return  **$1 + f(i-1, j-1)$** .

To summarise, these are the three choices we have in case characters don't match:

- **return  $1 + f(i-1, j)$**  // Insertion of character.
- **return  $1 + f(i, j-1)$**  // Deletion of character.
- **return  $1 + f(i-1, j-1)$**  // Replacement of character.

### Step 3: Return the minimum of all choices.

As we have to return the minimum number of operations,  
we will return the minimum of all operations.

### Base Cases:

We are reducing i and j in our recursive relation, there can be two possibilities, either i becomes -1 or j becomes -1., i,e we exhaust either S1 or S2 respectively.

i  
S1: "horse"

S2: "ros"

j

i  
0 1 2 3 4  
S1: "h o r s e"

S2: "r o s"

0 1 2

j

S1 gets exhausted

i  
-1  
S1: " "

S2: " r o "  
0 1  
j

Insert j+1 characters

S2 gets exhausted

i  
0 1 2  
S1: "h o r "

S2: " "  
-1  
j

Delete i+1 characters

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {
    if(i<0)
        return j+1
    if(j<0)
        return i+1

    if(S1[i]==S2[j])
        return 0 + f(i-1,j-1)

    else
        return 1+min(f(i-1,j-1),min(f(i-1,j),f(i,j-1)))
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][m]. The size of S1 and S2 are n and m respectively, so the variable i will always lie between ‘0’ and ‘n-1’ and the variable j between ‘0’ and ‘m-1’.
2. We initialize the dp array to -1.
3. Whenever we want to find the answer to particular parameters (say f(i,j)), we first check whether the answer is already calculated using the dp array(i.e dp[i][j] != -1 ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[i][j] to the solution we get.

### **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

int editDistanceUtil(string& S1, string& S2, int i, int j, vector<vector<int>>& dp){

    if(i<0)
        return j+1;
    if(j<0)
        return i+1;

    if(dp[i][j]!=-1) return dp[i][j];

    if(S1[i]==S2[j])
        return dp[i][j] = 0+editDistanceUtil(S1,S2,i-1,j-1,dp);

    // Minimum of three choices
    else return dp[i][j] = 1+min(editDistanceUtil(S1,S2,i-1,j-1,dp),
        min(editDistanceUtil(S1,S2,i-1,j,dp),editDistanceUtil(S1,S2,i,j-1,dp)));
}

int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<vector<int>> dp(n, vector<int>(m, -1));
    return editDistanceUtil(S1,S2,n-1,m-1,dp);
}

int main() {

    string s1 = "horse";
    string s2 = "ros";

    cout << "The minimum number of operations required is: " << editDistance(s1,s2);
}

```

**Output:** The minimum number of operations required is: 3

**Time Complexity: O(N\*M)**

Reason: There are N\*M states therefore at max 'N\*M' new problems will be solved.

**Space Complexity: O(N\*M) + O(N+M)**

Reason: We are using a recursion stack space(O(N+M)) and a 2D array ( O(N\*M)).

**Steps to convert Recursive Solution to Tabulation one.**

In the recursive logic, we set the base case too if( $i < 0$ ) and if( $j < 0$ ) but we can't set the dp array's index to -1. Therefore a hack for this issue is to shift every index by 1 towards the right.

Recursive code indexes:      -1, 0, 1, ..., n

Shifted indexes :      0, 1, ..., n+1

- First we initialise the dp array of size [n+1][m+1] as zero.
- Next, we set the base condition (keep in mind 1-based indexing), we set the first column's value as i and the first row as j( 1-based indexing).
- Similarly, we will implement the recursive code by keeping in mind the shifting of indexes, therefore S1[i] will be converted to S1[i-1]. Same for S2.
- At last, we will print dp[N][M] as our answer.

#### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    for(int i=0;i<=n;i++){
        dp[i][0] = i;
    }
    for(int j=0;j<=m;j++){
        dp[0][j] = j;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<m+1;j++){
            if(S1[i-1]==S2[j-1])
                dp[i][j] = 0+dp[i-1][j-1];

            else dp[i][j] = 1+min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]));
        }
    }

    return dp[n][m];
}

int main() {

    string s1 = "horse";
    string s2 = "ros";

    cout << "The minimum number of operations required is: " << editDistance(s1,s2);
}

```

**Output:** The minimum number of operations required is: 3

**Time Complexity: O(N\*M)**

Reason: There are two nested loops

**Space Complexity: O(N\*M)**

Reason: We are using an external array of size ‘N\*M’. Stack Space is eliminated.

### Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

## Approach:

---

We will space optimize in the following way:

- We take two rows 'prev' and 'cur'.
- We initialize it to the base condition. Now, at starting the prev row needs to be initialized with its column value. Moreover, the cur variable whenever declared should have its first cell as a row value. (See the code).
- Next, we implement the memoization logic. We replace  $dp[i-1]$  with prev and  $dp[i]$  by cur.
- After every inner loop execution, we set  $prev=cur$ , for the next iteration.
- At last, we return  $prev[m]$  as our answer.

## Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<int> prev(m+1, 0);
    vector<int> cur(m+1, 0);

    for(int j=0;j<=m;j++){
        prev[j] = j;
    }

    for(int i=1;i<n+1;i++){
        cur[0]=i;
        for(int j=1;j<m+1;j++){
            if(S1[i-1]==S2[j-1])
                cur[j] = 0+prev[j-1];

            else cur[j] = 1+min(prev[j-1],min(prev[j],cur[j-1]));
        }
        prev = cur;
    }

    return prev[m];
}

int main() {

    string s1 = "horse";
    string s2 = "ros";

    cout << "The minimum number of operations required is: " << editDistance(s1,s2);
}

```

**Output:** The minimum number of operations required is: 3

**Time Complexity: O(N\*M)**

Reason: There are two nested loops.

**Space Complexity: O(M)**

Reason: We are using an external array of size ‘M+1’ to store two rows.

# Wildcard Matching | (DP-34)

 [takeuforward.org/data-structure/wildcard-matching-dp-34](https://takeuforward.org/data-structure/wildcard-matching-dp-34)

April 20, 2022

## Problem Statement: Wildcard Matching

We are given two strings ‘S1’ and ‘S2’. String S1 can have the following two special characters:

1. ‘?’ can be matched to a single character of S2.
2. ‘\*’ can be matched to any sequence of characters of S2. (sequence can be of length zero or more).

We need to check whether strings S1 and S2 match or not.

**Example:**

S1: "?ay"  
S2: "ray"      **Matches**

We can replace ‘?’ of S1 at index 0 with ‘r’.

S1: "\*\*abcd"  
S2: "abcd"      **Matches**

We can replace ‘\*’ of S1 at index 0 and 1 with “(sequence of length 0).

S1: "ab\*cd"  
S2: "abcdefcd"      **Matches**

We can replace ‘\*’ of S1 at index 2 with ‘def’.

S1: "ab?d"  
S2: "abcc"      **Not Match**

Character ‘d’ at index 3 of S1 doesn’t match with character ‘c’ at index 3 of S2.

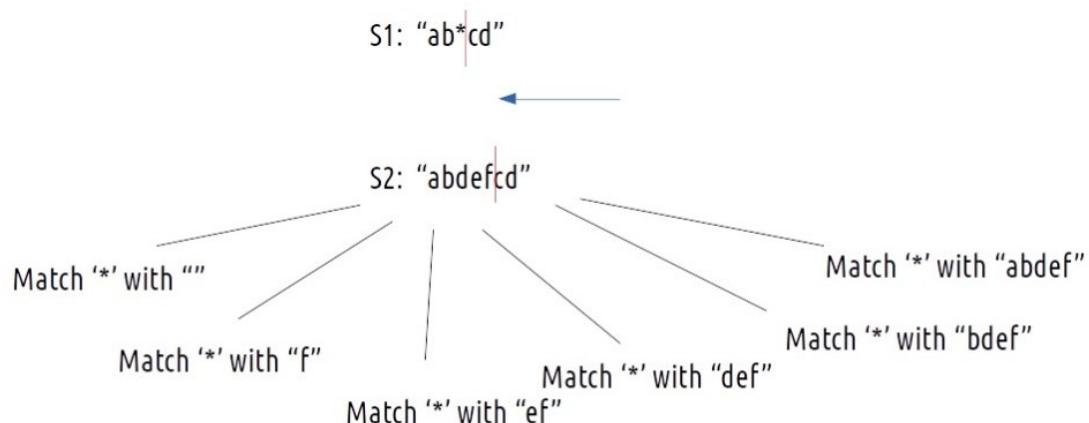
## Problem Link: [Wildcard Pattern Matching](#)

**Disclaimer:** Don’t jump directly to the solution, try it out yourself first.

## Intuition:

For every index of string S1, we have different options to match that index with string S2. Therefore, we can think in terms of string matching path as we have done already in previous questions.

- Either the characters match already.
- Or, if there is a '?', we can explicitly match a single character.
- For a '\*', the following figure explains the scenario.



As there is no uniformity in data, there is no other way to find out than to **try out all possible ways**. To do so we will need to use **recursion**.

### Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

**Step 1:** Express the problem in terms of indexes.

We are given two strings. We can represent them with the help of two indexes i and j. Initially,  $i=n-1$  and  $j=m-1$ , where n and m are lengths of strings S1 and S2. Initially, we will call  $f(n-1, m-1)$ , which means whether string  $S1[0...n-1]$  matches with string  $S2[0...m-1]$ .

We can generalize this as follows:

**$f(i,j) \rightarrow$  Return whether String  $S1[0...i]$  matches with String  $S2[0...j]$ .**

**Step 2:** Try out all possible choices at a given index.

Now, i and j represent two characters from strings S1 and S2 respectively. There are only two options that make sense: either the characters represented by i and j match or they don't.

#### (i) When the characters match

**if( $S_1[i] == S_2[j]$ ),**

If this is true, the characters at  $i$  and  $j$  match, we can simply move to the next characters of both the strings. So we will just decrement both  $i$  and  $j$  by 1 and recursively find the answer for the remaining string portions. We return  $f(i-1, j-1)$ . The following figure makes it clear.

### (ii) When the characters don't match

If the characters don't match, there are three possible scenarios:

1.  $S_1[i] == ?$
2.  $S_1[i] == *$
3.  $S_1[i]$  is some other character

Let us discuss them one by one:

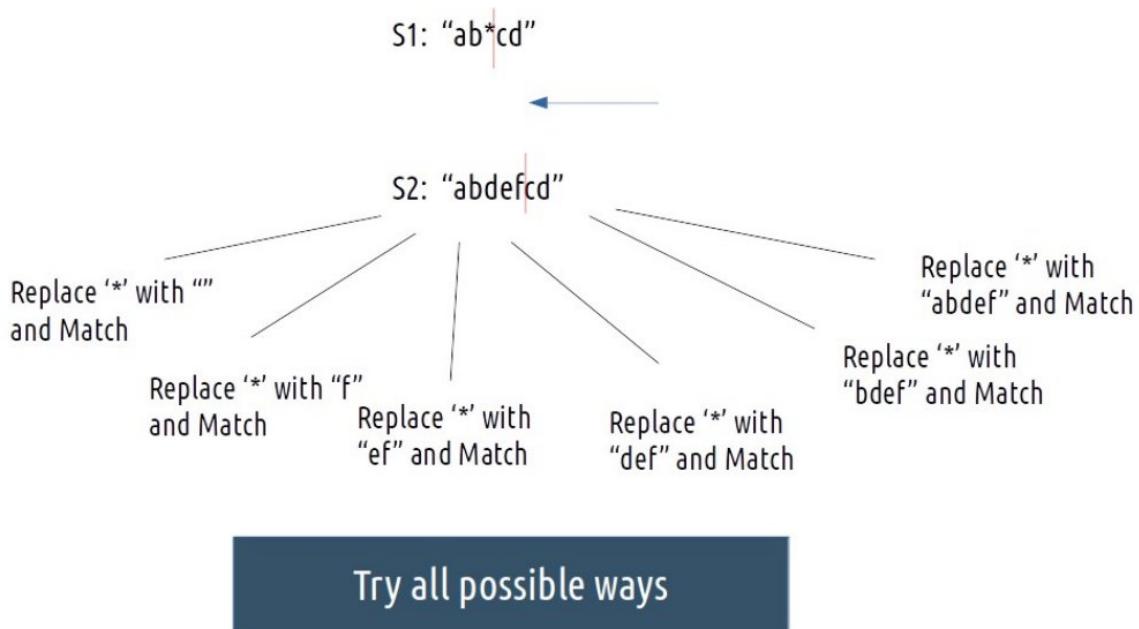
#### (i) If $S_1[i] == ?$

In this case, we can explicitly match ‘?’ at index  $i$  of  $S_1$  with the corresponding character at index  $j$  of  $S_2$ . And then recursively call  $f(i-1, j-1)$  to check for the remaining string.

#### (ii) If $S_1[i] == *$

This is an interesting case as now “\*” can be replaced with any sequence of characters( of length 0 or more) from  $S_2$ .

We will revisit this example:



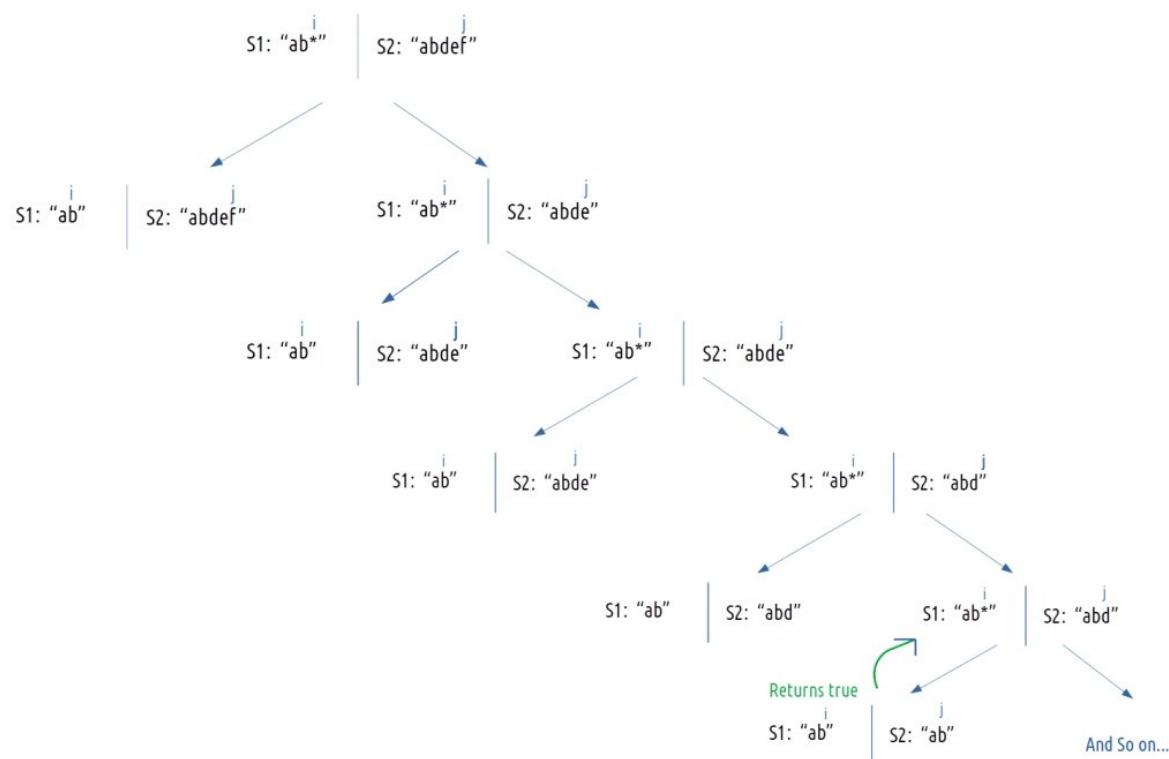
If any of these cases return true, we can say that the characters do match. The next question is **how to try all possible ways?**

We are using two pointers  $i$  and  $j$  to represent characters of strings  $S_1$  and  $S_2$ . We can surely write a for loop to compare characters from  $0$  to  $j$  of  $S_2$  for the above scenario. **Can we do it more smartly?** Yes, we can. Please understand the approach explained below.

We are using a recursive function  $f(i,j)$ . If we do only the following two recursive calls:

- Call **f(i-1,j)**. i.e **replace “\*” with nothing** and act as if it was not present.
  - Call **f(i,j-1)**. i.e **replace “\*” with a single character** at index j and make the i pointer to still point at index i. In this, we matched it with a single character (one of the many options that need to be tried) and in the next recursive call, as i still point to “\*”, we get the exact two recursive calls again.

The following recursive tree will help us to understand the recursion better.



So we see how we can tackle all the required cases associated with `**` by using recursion.

**(iii) If  $S1[i]$  is neither '?' nor '\*', then we can say as the characters at i and j don't match then the strings don't match, so we return false.**

To summarise:

1. If  $S1[i] == '?'$ , return  $f(i-1, j)$
  2. Else if  $S1[i] == '*'$ , return  $f(i-1, j) || f(i, j-1)$

3. Else return **false**

### **Step 3: Return logical OR (||) of all the choices**

If any of the cases return true, we can say that strings do match. We can use OR operator (||) with the recursive calls.

#### **Base Cases:**

We are reducing i and j in our recursive relation, there can be two possibilities, either i becomes -1 or j becomes -1., i.e we exhaust either S1 or S2 respectively.

##### **(i) When S1 is exhausted:**

When S1 is exhausted ( $i < 0$ ), we know that in order for the strings to match, String S2 should also exhaust simultaneously. If it does, we return true, else we return false.

We can say:

- if( $i < 0 \&\& j < 0$ ), return true.
- if( $i < 0 \&\& j \geq 0$ ), return false.

##### **(ii) When S2 is exhausted:**

When S2 is exhausted( $j < 0$ ) and S1 has not, there is only one pattern that can account for true(matching of strings). It is if S1 is like this “\*”, “\*\*\*\*”, “\*\*\*”, i.e: S1 contains only stars. Then we can replace every star with a sequence of length 0 and say that the string match.

If S1 is all-stars, we return true, else return false.

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {
    if(i<0 && j<0)    return true
    if(i<0 && j>=0)   return false
    if(j<0 && i>=0)
        return isAllStars(S1,i)
    if(S1[i]==S2[j] || S1[i]=='?')
        return f(i-1,j-1)
    else {
        if( S1[i] == '*' )
            return f(i-1,j) || f(i,j-1)
        else return false
    }
}

```

### **Steps to memoize a recursive solution:**

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][m]. The size of S1 and S2 are n and m respectively, so the variable i will always lie between ‘0’ and ‘n-1’ and the variable j between ‘0’ and ‘m-1’.
2. We initialize the dp array to -1.
3. Whenever we want to find the answer to particular parameters (say f(i,j)), we first check whether the answer is already calculated using the dp array(i.e dp[i][j]!= -1 ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[i][j] to the solution we get.

### **Code:**

- C++ Code

- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {
    for (int j = 0; j <= i; j++) {
        if (S1[j] != '*')
            return false;
    }
    return true;
}

bool wildcardMatchingUtil(string & S1, string & S2, int i, int j, vector < vector
< bool >> & dp) {

    //Base Conditions
    if (i < 0 && j < 0)
        return true;
    if (i < 0 && j >= 0)
        return false;
    if (j < 0 && i >= 0)
        return isAllStars(S1, i);

    if (dp[i][j] != -1) return dp[i][j];

    if (S1[i] == S2[j] || S1[i] == '?')
        return dp[i][j] = wildcardMatchingUtil(S1, S2, i - 1, j - 1, dp);

    else {
        if (S1[i] == '*')
            return wildcardMatchingUtil(S1, S2, i - 1, j, dp) ||
wildcardMatchingUtil(S1, S2, i, j - 1, dp);
        else return false;
    }
}

bool wildcardMatching(string & S1, string & S2) {

    int n = S1.size();
    int m = S2.size();

    vector < vector < bool >> dp(n, vector < bool > (m, -1));
    return wildcardMatchingUtil(S1, S2, n - 1, m - 1, dp);
}

int main() {

    string S1 = "ab*cd";
    string S2 = "abcdefcd";

    if (wildcardMatching(S1, S2))
        cout << "String S1 and S2 do match";
    else cout << "String S1 and S2 do not match";
}

```

## **Output:**

String S1 and S2 do match

## **Time Complexity: O(N\*M)**

Reason: There are  $N*M$  states therefore at max ' $N*M$ ' new problems will be solved.

## **Space Complexity: O(N\*M) + O(N+M)**

Reason: We are using a recursion stack space( $O(N+M)$ ) and a 2D array (  $O(N*M)$ ).

## **Steps to convert Recursive Solution to Tabulation one.**

In the recursive logic, we set the base case too if( $i < 0$ ) and if( $j < 0$ ) but we can't set the dp array's index to -1. Therefore a hack for this issue is to shift every index by 1 towards the right.

Recursive code indexes:      -1, 0, 1, ..., n

Shifted indexes :      0, 1, ..., n+1

- First we initialise the dp array of size  $[n+1][m+1]$  as zero.
- Next, we set the base condition (keep in mind 1-based indexing), we set the top-left cell as 'true', then we set the first column's value as 'false'; and for the first row, we will run `isAllStars()` for every cell value.
- Similarly, we will implement the recursive code by keeping in mind the shifting of indexes, therefore  $S1[i]$  will be converted to  $S1[i-1]$ . Same for  $S2$ .
- At last we will print  $dp[n][m]$  as our answer.

## **Code:**

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {

    // S1 is taken in 1-based indexing
    for (int j = 1; j <= i; j++) {
        if (S1[j - 1] != '*')
            return false;
    }
    return true;
}

bool wildcardMatching(string & S1, string & S2) {

    int n = S1.size();
    int m = S2.size();

    vector < vector < bool >> dp(n + 1, vector < bool > (m, false));

    dp[0][0] = true;

    for (int j = 1; j <= m; j++)
        dp[0][j] = false;
    }

    for (int i = 1; i <= n; i++) {
        dp[i][0] = isAllStars(S1, i);
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {

            if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
                dp[i][j] = dp[i - 1][j - 1];

            else {
                if (S1[i - 1] == '*')
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];

                else dp[i][j] = false;
            }
        }
    }

    return dp[n][m];
}

int main() {

    string S1 = "ab*cd";
    string S2 = "abdefcd";

    if (wildcardMatching(S1, S2))
        cout << "String S1 and S2 do match";
}

```

```
    else cout << "String S1 and S2 do not match";
}
```

### Output:

String S1 and S2 do match

### Time Complexity: O(N\*M)

Reason: There are two nested loops

### Space Complexity: O(N\*M)

Reason: We are using an external array of size 'N\*M'. Stack Space is eliminated.

## Part 3: Space Optimization

If we closely look the relation,

$$dp[i][j] = dp[i-1][j-1], dp[i][j] = dp[i-1][j] \mid\mid dp[i][j-1]$$

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev) and current row's previous columns values. So, we don't need to store an entire array. Hence we can space optimise it.

### Approach:

We will space optimize in the following way:

- We take two rows 'prev' and 'cur'.
- We initialize it to the base condition. We first initialize the prev row. Its first value needs to be true. Rest all the values of the prev row needs to be false.
- Moreover, the cur variable whenever declared should have its first cell's value given by isAllStarts() function.
- Next, we implement the memoization logic. We replace  $dp[i-1]$  with prev and  $dp[i]$  by cur.
- After every inner loop execution, we set  $prev=cur$ , for the next iteration.
- At last, we return  $prev[m]$  as our answer.

### Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {

    // S1 is taken in 1-based indexing
    for (int j = 1; j <= i; j++) {
        if (S1[j - 1] != '*')
            return false;
    }
    return true;
}

bool wildcardMatching(string & S1, string & S2) {

    int n = S1.size();
    int m = S2.size();

    vector < bool > prev(m + 1, false);
    vector < bool > cur(m + 1, false);

    prev[0] = true;

    for (int i = 1; i <= n; i++) {
        cur[0] = isAllStars(S1, i);
        for (int j = 1; j <= m; j++) {

            if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
                cur[j] = prev[j - 1];

            else {
                if (S1[i - 1] == '*')
                    cur[j] = prev[j] || cur[j - 1];

                else cur[j] = false;
            }
        }
        prev = cur;
    }

    return prev[m];
}

int main() {

    string S1 = "ab*cd";
    string S2 = "abcdefcd";

    if (wildcardMatching(S1, S2))
        cout << "String S1 and S2 do match";
    else cout << "String S1 and S2 do not match";
}

```

### **Output:**

String S1 and S2 do match

**Time Complexity: O(N\*M)**

Reason: There are two nested loops.

**Space Complexity: O(M)**

Reason: We are using an external array of size 'M+1' to store two rows.