0/1 Knapsack Problem

Last Updated : 18 Mar, 2024

Prerequisites: Introduction to Knapsack Problem, its Types and How to solve them

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Examples:

Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
Output: *3*
Explanation: *There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
Output: *0*

Recommended Problem

0 - 1 Knapsack Problem

Dynamic Programming Algorithms Flipkart Morgan Stanley +9 more

Solve Problem

Submission count: 3.8L

Recursion Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.

Optimal Substructure: *To consider all subsets of items, there can be two cases for every item.*

- ***Case 1:*** *The item is included in the optimal subset.*
- ***Case 2:*** *The item is not included in the optimal set.*

Follow the below steps to solve the problem:

The maximum value obtained from 'N' items is the max of the following two values.

- Case 1 (include the N^{th} item): Value of the N^{th} item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the N^{th} item).
- Case 2 (exclude the N^{th} item): Maximum value obtained by N-1 items and W weight.
- If the weight of the 'N^{th}' item is greater than 'W', then the Nth item cannot be included and **Case 2** is the only possibility.

Below is the implementation of the above approach:

C++ C Java Python C# Javascript PHP

```cpp
/* A Naive recursive implementation of
 0-1 Knapsack problem */
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{

    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more
    // than Knapsack capacity W, then
    // this item cannot be included
    // in the optimal solution
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << knapSack(W, weight, profit, n);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output

```
220
```

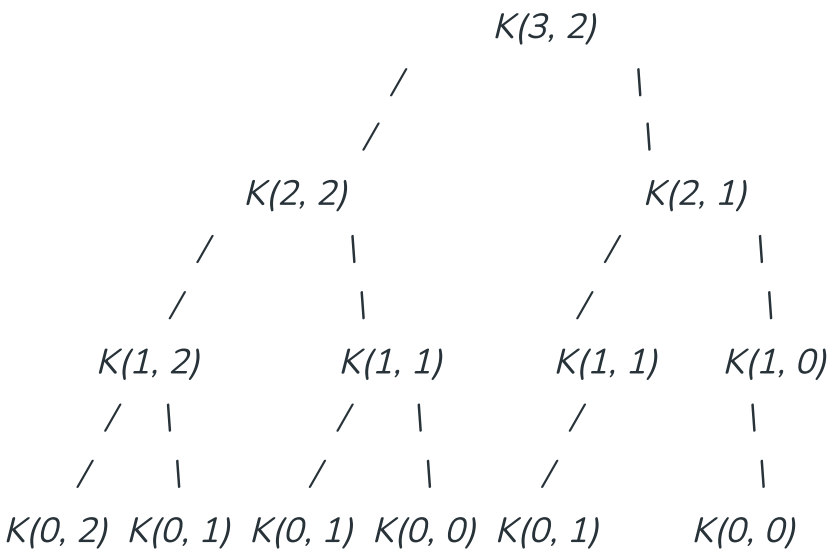
Time Complexity: $O(2^N)$
Auxiliary Space: O(N), Stack space required for recursion

Dynamic Programming Approach for 0/1 Knapsack Problem

Memoization Approach for 0/1 Knapsack Problem:

Note: It should be noted that the above function using recursion computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice.

*In the following recursion tree, **K()** refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.*
The recursion tree is for following sample inputs.
weight[] = {1, 1, 1}, W = 2, profit[] = {10, 20, 30}



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem.

> *If we get a subproblem the first time, we can solve this problem by creating a 2-D array that can store a particular state (n, w). Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time.*

Below is the implementation of the above approach:

C++ Java C# Javascript Python3

```cpp
// Here is the top-down approach of
// dynamic programming
#include <bits/stdc++.h>
using namespace std;

// Returns the value of maximum profit
int knapSackRec(int W, int wt[], int val[], int index, int** dp)
{
    // base condition
    if (index < 0)
        return 0;
    if (dp[index][W] != -1)
        return dp[index][W];

    if (wt[index] > W) {

        // Store the value of function call
        // stack in table before return
        dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
        return dp[index][W];
    }
    else {
        // Store value in a table before return
        dp[index][W] = max(val[index]
                           + knapSackRec(W - wt[index], wt, val,
                                         index - 1, dp),
                           knapSackRec(W, wt, val, index - 1, dp));

        // Return value of table after storing
        return dp[index][W];
    }
}

int knapSack(int W, int wt[], int val[], int n)
{
```

```
        // double pointer to declare the
        // table dynamically
        int** dp;
        dp = new int*[n];

        // loop to create the table dynamically
        for (int i = 0; i < n; i++)
            dp[i] = new int[W + 1];

        // loop to initially filled the
        // table with -1
        for (int i = 0; i < n; i++)
            for (int j = 0; j < W + 1; j++)
                dp[i][j] = -1;
        return knapSackRec(W, wt, val, n - 1, dp);
    }

    // Driver Code
    int main()
    {
        int profit[] = { 60, 100, 120 };
        int weight[] = { 10, 20, 30 };
        int W = 50;
        int n = sizeof(profit) / sizeof(profit[0]);
        cout << knapSack(W, weight, profit, n);
        return 0;
    }
```

Output

```
220
```

Time Complexity: O(N * W). As redundant calculations of states are avoided.
Auxiliary Space: O(N * W) + O(N). The use of a 2D array data structure for storing intermediate states and O(N) auxiliary stack space(ASS) has been used for recursion stack

Bottom-up Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:

> *Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0/1 Knapsack problem has both*

properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computation of the same subproblems can be avoided by constructing a temporary array K[] [] in a bottom-up manner.

Illustration:

Below is the illustration of the above approach:

*Let, **weight[] = {1, 2, 3}, profit[] = {10, 15, 40}, Capacity = 6***

- *If no element is filled, then the possible profit is 0.*

weight⇢ item⇣/	***0***	***1***	***2***	***3***	***4***	***5***	***6***
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1							
2							
3							

- ***For filling the first item in the bag:** If we follow the above mentioned procedure, the table will look like the following.*

weight⇢ item⇣/	***0***	***1***	***2***	***3***	***4***	***5***	***6***
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*10*	*10*	*10*	*10*	*10*	*10*

weight⇢ item⇣/	0	1	2	3	4	5	6
2							
3							

- ***For filling the second item:***
  When jthWeight = 2, then maximum possible profit is max (10, DP[1][2-2] + 15) = max(10, 15) = 15.
  When jthWeight = 3, then maximum possible profit is max(2 not put, 2 is put into bag) = max(DP[1][3], 15+DP[1][3-2]) = max(10, 25) = 25.

weight⇢ item⇣/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- ***For filling the third item:***
  When jthWeight = 3, the maximum possible profit is max(DP[2][3], 40+DP[2][3-3]) = max(25, 40) = 40.
  When jthWeight = 4, the maximum possible profit is max(DP[2][4], 40+DP[2][4-3]) = max(25, 50) = 50.
  When jthWeight = 5, the maximum possible profit is max(DP[2][5], 40+DP[2][5-3]) = max(25, 55) = 55.
  When jthWeight = 6, the maximum possible profit is max(DP[2][6], 40+DP[2][6-3]) = max(25, 65) = 65.

weight⇢ item⇣/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Below is the implementation of the above approach:

C++ C Java C# Javascript PHP Python3

```cpp
// A dynamic programming based
// solution for 0-1 Knapsack problem
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    vector<vector<int> > K(n + 1, vector<int>(W + 1));

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1]
                              + K[i - 1][w - wt[i - 1]],
                          K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}
```

```
// Driver Code
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);

    cout << knapSack(W, weight, profit, n);

    return 0;
}

// This code is contributed by Debojyoti Mandal
```

Output

```
220
```

Time Complexity: O(N * W). where ‘N’ is the number of elements and ‘W’ is capacity.
Auxiliary Space: O(N * W). The use of a 2-D array of size ‘N*W’.

Space optimized Approach for 0/1 Knapsack Problem using Dynamic Programming:

To solve the problem follow the below idea:

> *For calculating the current row of the dp[] array we require only previous row, but if we start traversing the rows from right to left then it can be done with a single row only*

Below is the implementation of the above approach:

C++ Java C# Javascript Python3

Last Updated : 20 Dec, 2022

Given a boolean expression with the following symbols.

```
Symbols
    'T' ---> true
    'F' ---> false
```

And following operators filled between symbols

```
Operators
    &   ---> boolean AND
    |   ---> boolean OR
    ^   ---> boolean XOR
```

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.
Let the input be in form of two arrays one contains the symbols (T and F) in order and the other contains operators (&, | and ^}

Examples:

```
Input: symbol[]    = {T, F, T}
       operator[]  = {^, &}
Output: 2
The given expression is "T ^ F & T", it evaluates true
in two ways "((T ^ F) & T)" and "(T ^ (F & T))"

Input: symbol[]    = {T, F, F}
       operator[]  = {^, |}
Output: 2
The given expression is "T ^ F | F", it evaluates true
in two ways "( (T ^ F) | F )" and "( T ^ (F | F) )".

Input: symbol[]    = {T, T, F, T}
       operator[]  = {|, &, ^}
Output: 4
The given expression is "T | T & F ^ T", it evaluates true
in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)
and (T|((T&F)^T)).
```

Recommended Problem

Boolean Parenthesization

Solve Problem

Dynamic Programming Algorithms Amazon Microsoft +2 more

Submission count: 1.1L

Solution:

Let **T(i, j)** represent the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

$$T(i,j) = \sum_{k=i}^{j-1} \left\{ \begin{array}{ccccc} T(i,k) * T(k+1,j) & if & operator & [k] & is'\&' \\ Total(i,k) * Total(k+1,j) - F(i,k) * F(k+1,j) & if & operator & [k] & is'|' \\ T(i,k) * F(k+1,j) + F(i,k) * T(k+1,j) & if & operator & [k] & is'\oplus' \end{array} \right\} Total(i,j) = T(i,j) + F(i,j)$$

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ Total(i,k) * Total(k+1,j) - F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * F(k+1,j) + F(i,k) * T(k+1) & \text{If operator[k] is '\^'} \end{cases}$$

$$\text{Total(i, j)} = \text{T(i, j)} + \text{F(i, j)}$$

Let **F(i, j)** represent the number of ways to parenthesize the symbols between i and j (inclusive) such that the subexpression between i and j evaluates to false.

$$F(i,j) = \sum_{k=i}^{j-1} \left\{ \begin{array}{ccc} Total(i,k) * Total(k+1,j) - T(i,k) * T(k+1,j) & if & operator[k] \quad is'\&' \\ F(i,k) * F(k+1,j) & if & operator[k] \quad is'|' \\ T(i,k) * T(k+1,j) + F(i,k) * F(k+1,j) & if & operator[k] \quad is'\oplus' \end{array} \right\} Total(i,j) = T(i,j) + F(i,j)$$

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} Total(i,k) * Total(k+1,j) - T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * T(k+1,j) + F(i,k) * F(k+1) & \text{If operator[k] is '\^'} \end{cases}$$

$$\text{Total(i, j)} = \text{T(i, j)} + \text{F(i, j)}$$

Base Cases:

```
T(i, i) = 1 if symbol[i] = 'T'
T(i, i) = 0 if symbol[i] = 'F'
```

```
F(i, i) = 1 if symbol[i] = 'F'
F(i, i) = 0 if symbol[i] = 'T'
```

If we draw the recursion tree of the above recursive solution, we can observe that it has many overlapping subproblems. Like other dynamic programming problems, it can be solved by filling a table in a bottom-up manner. Following is the implementation of a dynamic programming solution.

C++

```
#include <cstring>
#include <iostream>
using namespace std;

// Returns count of all possible
// parenthesizations that lead
// to result true for a boolean
// expression with symbols like
// true and false and operators
// like &, | and ^ filled
// between symbols
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diagonal entries first
    // All diagonal entries in
    // T[i][i] are 1 if symbol[i]
    // is T (true).  Similarly,
    // all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++) {
        F[i][i] = (symb[i] == 'F') ? 1 : 0;
        T[i][i] = (symb[i] == 'T') ? 1 : 0;
    }

    // Now fill T[i][i+1],
    // T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2],
    // F[i][i+3]... in order
    for (int gap = 1; gap < n; ++gap)
    {
        for (int i = 0, j = gap;
             j < n; ++i, ++j)
        {
```

```
                T[i][j] = F[i][j] = 0;
                for (int g = 0;
                     g < gap; g++)
                {
                    // Find place of parenthesization using
                    // current value of gap
                    int k = i + g;

                    // Store Total[i][k]
                    // and Total[k+1][j]
                    int tik = T[i][k] + F[i][k];
                    int tkj = T[k + 1][j]
                      + F[k + 1][j];

                    // Follow the recursive formulas
                    // according
                    // to the current operator
                    if (oper[k] == '&') {
                        T[i][j] += T[i][k]
                                * T[k + 1][j];
                        F[i][j] += (tik * tkj
                                    - T[i][k]
                                    * T[k + 1][j]);
                    }
                    if (oper[k] == '|') {
                        F[i][j] += F[i][k]
                          * F[k + 1][j];
                        T[i][j] += (tik * tkj
                                    - F[i][k]
                                    * F[k + 1][j]);
                    }
                    if (oper[k] == '^') {
                        T[i][j] += F[i][k]
                          * T[k + 1][j]
                          + T[i][k]
                          * F[k + 1][j];
                        F[i][j] += T[i][k]
                                * T[k + 1][j]
                                + F[i][k] * F[k + 1][j];
                    }
                }
            }
        }
        return T[0][n - 1];
    }

    // Driver code
    int main()
    {
```

```
    char symbols[] = "TTFT";
    char operators[] = "|&^";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and
    // (T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}
```

Java

```
import java.io.*;
import java.util.*;

class GFG {

    // Returns count of all possible
    // parenthesizations that lead to
    // result true for a boolean
    // expression with symbols like true
    // and false and operators like &, |
    // and ^ filled between symbols
    static int countParenth(char symb[], char oper[], int n)
    {
        int F[][] = new int[n][n];
        int T[][] = new int[n][n];

        // Fill diagonal entries first
        // All diagonal entries in T[i][i]
        // are 1 if symbol[i] is T (true).
        // Similarly, all F[i][i] entries
        // are 1 if symbol[i] is F (False)
        for (int i = 0; i < n; i++) {
            F[i][i] = (symb[i] == 'F') ? 1 : 0;
            T[i][i] = (symb[i] == 'T') ? 1 : 0;
        }

        // Now fill T[i][i+1], T[i][i+2],
        // T[i][i+3]... in order And F[i][i+1],
        // F[i][i+2], F[i][i+3]... in order
        for (int gap = 1; gap < n; ++gap) {
            for (int i = 0, j = gap; j < n; ++i, ++j) {
                T[i][j] = F[i][j] = 0;
                for (int g = 0; g < gap; g++)

                {
```

Output

```
4
```

Time Complexity: $O(n^3)$, as we are using nested loops to traverse n^3 times. Where n is the length of the symbols string.
Auxiliary Space: $O(n^2)$, as we are using extra space for the DP matrix. Where n is the length of the symbols string.

Approach 2:

We can also use the recursive approach (Top-Down DP), this approach uses memoization.

C++

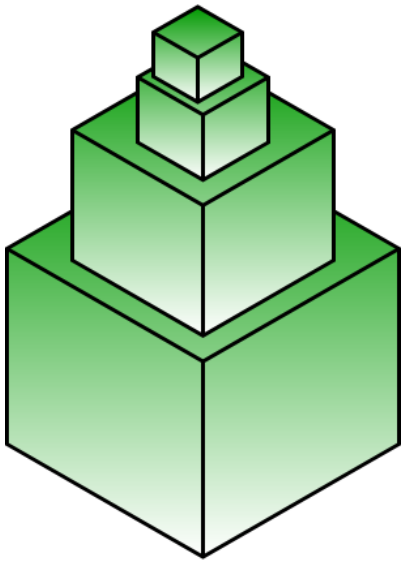
```
#include <bits/stdc++.h>
using namespace std;

int dp[101][101][2];
int parenthesis_count(string s,
                      int i,
                      int j,
                      int isTrue)
{
    // Base Condition
    if (i > j)
        return false;
    if (i == j) {
        if (isTrue == 1)
            return s[i] == 'T';
        else
            return s[i] == 'F';
    }

    if (dp[i][j][isTrue] != -1)
        return dp[i][j][isTrue];
    int ans = 0;
    for (int k = i + 1
         ; k <= j - 1; k = k + 2)
    {
        int leftF, leftT, rightT, rightF;
        if (dp[i][k - 1][1] == -1)
        {
            leftT = parenthesis_count(s, i, k - 1, 1);
```

Box Stacking Problem | DP-22

Last Updated : 20 Mar, 2023

You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Source: http://people.csail.mit.edu/bdean/6.046/dp/. The link also has a video for an explanation of the solution.



The Box Stacking problem is a variation of LIS problem. We need to build a maximum height stack.

Following are the key points to note in the problem statement:
1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
2) We can rotate boxes such that width is smaller than depth. For example, if there is a box with dimensions {1x2x3} where 1 is height, 2×3 is base, then there can be three possibilities, {1x2x3}, {2x1x3} and {3x1x2}
3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.
Following is the **solution** based on DP solution of LIS problem.

Method 1 : dynamic programming using tabulation

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of the original array. For simplicity, we consider width as always smaller than or equal to depth.
2) Sort the above generated 3n boxes in decreasing order of base area.
3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.
MSH(i) = Maximum possible Stack Height with box i at top of stack
MSH(i) = { Max (MSH(j)) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).
If there is no such j then MSH(i) = height(i)
4) To get overall maximum height, we return max(MSH(i)) where 0 < i < n
Following is the implementation of the above solution.

```
            Box box = rot[i];
            int val = 0;

            for(int j = 0; j < i; j++){
                Box prevBox = rot[j];
                if(box.w < prevBox.w && box.d < prevBox.d){
                    val = Math.max(val, msh[j]);
                }
            }
            msh[i] = val + box.h;
        }

        int max = -1;

        /* Pick maximum of all msh values */
        for(int i = 0; i < count; i++){
            max = Math.max(max, msh[i]);
        }

        return max;
    }

    /* Driver program to test above function */
    public static void main(String[] args) {

        Box[] arr = new Box[4];
        arr[0] = new Box(4, 6, 7);
        arr[1] = new Box(1, 2, 3);
        arr[2] = new Box(4, 5, 6);
        arr[3] = new Box(10, 12, 32);

        System.out.println("The maximum possible "+
                           "height of stack is " +
                           maxStackHeight(arr,4));
    }
}

// This code is contributed by Divyam
```

Python3

```
# Dynamic Programming implementation
# of Box Stacking problem
class Box:

    # Representation of a box
    def __init__(self, h, w, d):
        self.h = h
```

```
            self.w = w
            self.d = d

        def __lt__(self, other):
            return self.d * self.w < other.d * other.w

    def maxStackHeight(arr, n):

        # Create an array of all rotations of
        # given boxes. For example, for a box {1, 2, 3},
        # we consider three instances{{1, 2, 3},
        # {2, 1, 3}, {3, 1, 2}}
        rot = [Box(0, 0, 0) for _ in range(3 * n)]
        index = 0

        for i in range(n):

            # Copy the original box
            rot[index].h = arr[i].h
            rot[index].d = max(arr[i].d, arr[i].w)
            rot[index].w = min(arr[i].d, arr[i].w)
            index += 1

            # First rotation of the box
            rot[index].h = arr[i].w
            rot[index].d = max(arr[i].h, arr[i].d)
            rot[index].w = min(arr[i].h, arr[i].d)
            index += 1

            # Second rotation of the box
            rot[index].h = arr[i].d
            rot[index].d = max(arr[i].h, arr[i].w)
            rot[index].w = min(arr[i].h, arr[i].w)
            index += 1

        # Now the number of boxes is 3n
        n *= 3

        # Sort the array 'rot[]' in non-increasing
        # order of base area
        rot.sort(reverse = True)

        # Uncomment following two lines to print
        # all rotations
        # for i in range(n):
        #     print(rot[i].h, 'x', rot[i].w, 'x', rot[i].d)

        # Initialize msh values for all indexes
        # msh[i] --> Maximum possible Stack Height
```

```
    # with box i on top
    msh = [0] * n

    for i in range(n):
        msh[i] = rot[i].h

    # Compute optimized msh values
    # in bottom up manner
    for i in range(1, n):
        for j in range(0, i):
            if (rot[i].w < rot[j].w and
                rot[i].d < rot[j].d):
                if msh[i] < msh[j] + rot[i].h:
                    msh[i] = msh[j] + rot[i].h

    maxm = -1
    for i in range(n):
        maxm = max(maxm, msh[i])

    return maxm

# Driver Code
if __name__ == "__main__":
    arr = [Box(4, 6, 7), Box(1, 2, 3),
           Box(4, 5, 6), Box(10, 12, 32)]
    n = len(arr)
    print("The maximum possible height of stack is",
           maxStackHeight(arr, n))

# This code is contributed by vibhu4agarwal
```

C#

```
using System;

class Box
{
    public int h, w, d, area;
    public Box(int h, int w, int d)
    {
        this.h = h;
        this.w = w;
        this.d = d;
    }

    public bool IsSmallerThan(Box other)
    {
        return this.w * this.d < other.w * other.d;
```

```
    if (maxHeight < maxStackHeight[i]) {
      maxHeight = maxStackHeight[i];
    }
  }

  return maxHeight;
}

// Example usage
const boxes = [
  new Box(4, 6, 7),
  new Box(1, 2, 3),
  new Box(4, 5, 6),
  new Box(10, 12, 32),
];
const ans = maxStackHeight(boxes);
console.log(ans); // Output: 60
```

Output

```
The maximum possible height of stack is 60
```

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

```
10 x 12 x 32
12 x 10 x 32
32 x 10 x 12
4 x 6 x 7
4 x 5 x 6
6 x 4 x 7
5 x 4 x 6
7 x 4 x 6
6 x 4 x 5
1 x 2 x 3
2 x 1 x 3
3 x 1 x 2
```

The height 60 is obtained by boxes { {**3**, 1, 2}, {**1**, 2, 3}, {**6**, 4, 5}, {**4**, 5, 6}, {**4**, 6, 7}, {**32**, 10, 12}, {**10**, 12, 32}}

Count all combinations of coins to make a given value sum (Coin Change II)

Last Updated : 05 Mar, 2024

Given an integer array of **coins[ ]** of size **N** representing different types of denominations and an integer **sum**, the task is to count all combinations of coins to make a given value **sum**.

Note: Assume that you have an infinite supply of each type of coin.

Examples:

Input: *sum = 4, coins[] = {1,2,3},*
Output: *4*
Explanation: *there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}.*

Input: *sum = 10, coins[] = {2, 5, 3, 6}*

Output: *5*

Explanation: *There are five solutions:*

{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.

Recommended Problem

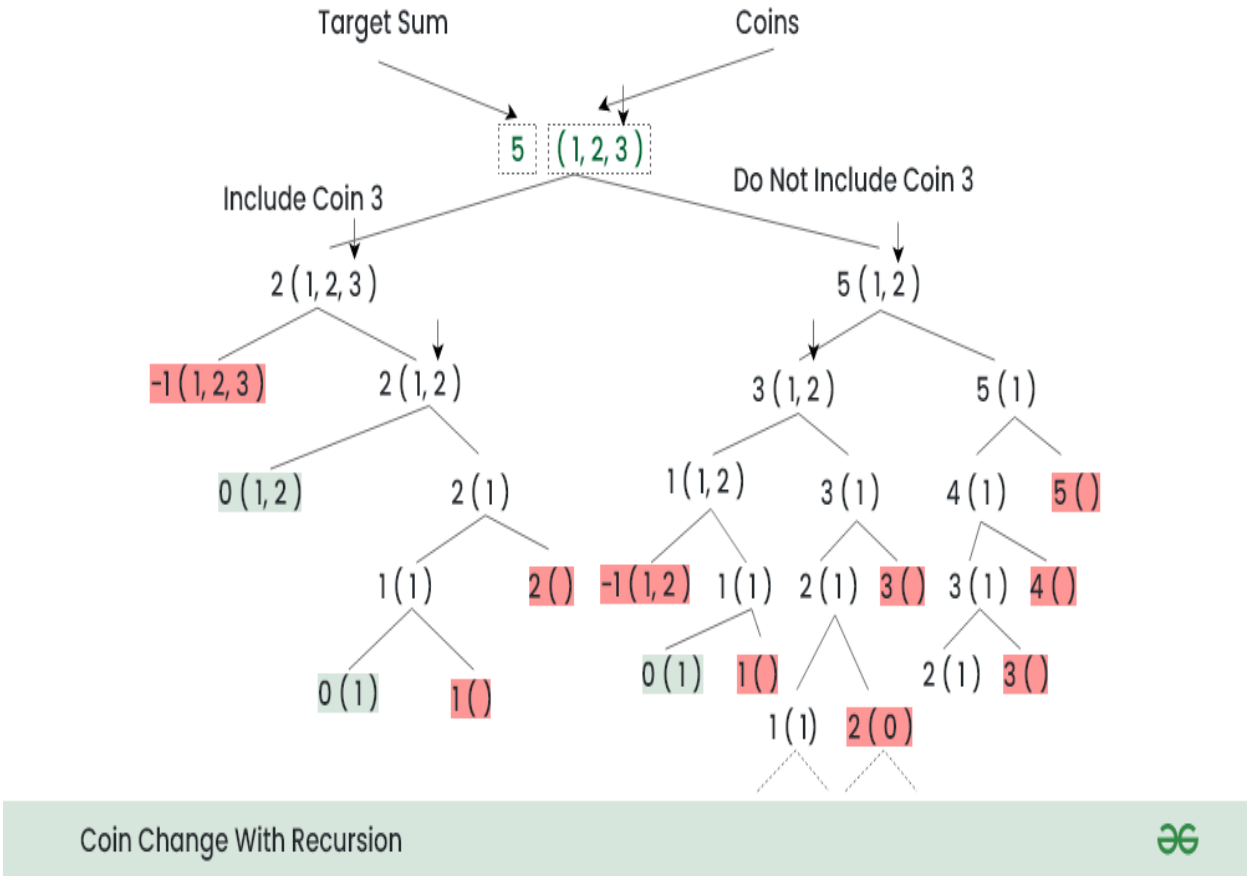
Coin Change

Arrays Dynamic Programming +2 more Paytm Flipkart +7 more

Solve Problem

Submission count: 2.6L

Count all combinations of coins to make a given value sum using Recursion:



Coin Change Using Recursion

Recurrence Relation:

count(coins,n,sum) = count(coins,n,sum-count[n-1]) + count(coins,n-1,sum)

For each coin, there are 2 options.

- **Include the current coin**: Subtract the current coin's denomination from the target sum and call the count function recursively with the updated sum and the same set of coins i.e., count(coins, n, sum – coins[n-1])
- **Exclude the current coin**: Call the count function recursively with the same sum and the remaining coins. i.e., count(coins, n-1,sum).

The final result will be the sum of both cases.

Base case:

- If the target sum (sum) is 0, there is only one way to make the sum, which is by not selecting any coin. So, count(0, coins, n) = 1.
- If the target sum (sum) is negative or no coins are left to consider (n == 0), then there are no ways to make the sum, so count(sum, coins, 0) = 0.

Below is the Implementation of the above approach.

C++

```
// Recursive C++ program for
// coin change problem.
#include <bits/stdc++.h>
using namespace std;

// Returns the count of ways we can
// sum coins[0...n-1] coins to get sum "sum"
int count(int coins[], int n, int sum)
{

    // If sum is 0 then there is 1 solution
    // (do not include any coin)
    if (sum == 0)
        return 1;

    // If sum is less than 0 then no
    // solution exists
```

```
    if (sum < 0)
        return 0;

    // If there are no coins and sum
    // is greater than 0, then no
    // solution exist
    if (n <= 0)
        return 0;

    // count is sum of solutions (i)
    // including coins[n-1] (ii) excluding coins[n-1]
    return count(coins, n, sum - coins[n - 1])
           + count(coins, n - 1, sum);
}

// Driver code
int main()
{
    int i, j;
    int coins[] = { 1, 2, 3 };
    int n = sizeof(coins) / sizeof(coins[0]);
    int sum = 5;

    cout << " " << count(coins, n, sum);

    return 0;
}
```

C

```
// Recursive C program for
// coin change problem.
#include <stdio.h>

// Returns the count of ways we can
// sum coins[0...n-1] coins to get sum "sum"
int count(int coins[], int n, int sum)
{
    // If sum is 0 then there is 1 solution
    // (do not include any coin)
    if (sum == 0)
        return 1;

    // If sum is less than 0 then no
    // solution exists
    if (sum < 0)
        return 0;
```

Time Complexity: $O(2^{sum})$

Auxiliary Space: O(sum)

Count all combinations of coins to make a given value sum

Dynamic Programming (Memoization):

> *The above recursive solution has Optimal Substructure and Overlapping Subproblems so Dynamic programming (Memoization) can be used to solve the problem. So 2D array can be used to store results of previously solved subproblems.*

Follow the below steps to Implement the idea:

- Create a 2D dp array to store the results of previously solved subproblems.
- dp[i][j] will represent the number of distinct ways to make the sum **j** by using the first **i** coins.
- During the recursion call, if the same state is called more than once, then we can directly return the answer stored for that state instead of calculating again.

Below is the implementation using the Memoization:

C++

```
#include <bits/stdc++.h>
using namespace std;

// Recursive function to count the numeber of distinct ways
// to make the sum by using n coins

int count(vector<int>& coins, int n, int sum,
          vector<vector<int> >& dp)
{
    // Base Case
    if (sum == 0)
        return dp[n][sum] = 1;

    // If number of coins is 0 or sum is less than 0 then
    // there is no way to make the sum.
```

```
    if (n == 0 || sum < 0)
        return 0;

    // If the subproblem is previously calculated then
    // simply return the result
    if (dp[n][sum] != -1)
        return dp[n][sum];

    // Two options for the current coin
    return dp[n][sum]
           = count(coins, n, sum - coins[n - 1], dp)
             + count(coins, n - 1, sum, dp);
}
int32_t main()
{
    int tc = 1;
    // cin >> tc;
    while (tc--) {
        int n, sum;
        n = 3, sum = 5;
        vector<int> coins = { 1, 2, 3 };
        // 2d dp array to store previously calculated
        // results
        vector<vector<int> > dp(n + 1,
                                vector<int>(sum + 1, -1));
        int res = count(coins, n, sum, dp);
        cout << res << endl;
    }
}
```

Java

```
// Java program for the above approach
import java.util.*;

class GFG {
    // Recursive function to count the numeber of distinct
    // ways to make the sum by using n coins
    static int count(int[] coins, int sum, int n,
                     int[][] dp)
    {
      // Base Case
        if (sum == 0)
            return dp[n][sum] = 1;

      // If number of coins is 0 or sum is less than 0 then
        // there is no way to make the sum.
        if (n == 0 || sum<0)
```

```
            return 0;

        // If the subproblem is previously calculated then
        // simply return the result
        if (dp[n][sum] != -1)
            return dp[n][sum];

       // Two options for the current coin
        return dp[n][sum]
            = count(coins, sum - coins[n - 1], n, dp)
              + count(coins, sum, n - 1, dp);
    }

    // Driver code
    public static void main(String[] args)
    {
        int tc = 1;
        while (tc != 0) {
            int n, sum;
            n = 3;
            sum = 5;
            int[] coins = { 1, 2, 3 };
            int[][] dp = new int[n + 1][sum + 1];
            for (int[] row : dp)
                Arrays.fill(row, -1);
            int res = count(coins, sum, n, dp);
            System.out.println(res);
            tc--;
        }
    }
}
```

C#

```
// C# program for the above approach
using System;
public class GFG {
    // Recursive function to count the numeber of distinct
    // ways to make the sum by using n coins
    static int count(int[] coins, int sum, int n,
                     int[, ] dp)
    {
      // Base Case
        if (sum == 0)
            return dp[n, sum] = 1;

      // If number of coins is 0 or sum is less than 0 then
        // there is no way to make the sum.
```

```
    # If the subproblem is previously calculated then simply return the result
    if (dp[n][sum] != -1):
        return dp[n][sum]

      # Two options for the current coin

    dp[n][sum] = count(coins, sum - coins[n - 1], n, dp) + \
        count(coins, sum, n - 1, dp)

    return dp[n][sum]


# Driver code
if __name__ == '__main__':
    tc = 1
    while (tc != 0):
        n = 3
        sum = 5
        coins = [1, 2, 3]
        dp = [[-1 for i in range(sum+1)] for j in range(n+1)]
        res = count(coins, sum, n, dp)
        print(res)
        tc -= 1
```

Output

```
5
```

Time Complexity: O(N*sum), where N is the number of coins and sum is the target sum.
Auxiliary Space: O(N*sum)

Count all combinations of coins to make a given value sum using Dynamic Programming (Tabulation):

We can use the following steps to implement the dynamic programming(tabulation) approach for Coin Change.

- Create a 2D dp array with rows and columns equal to the number of coin denominations and target sum.
- `dp[0][0]` will be set to `1` which represents the base case where the **target sum** is **0**, and there is only one way to make the change by not selecting any

coin.

- Iterate through the rows of the **dp** array (i from 1 to `n`), representing the current coin being considered.
  - The inner loop iterates over the target sums (`j` from 0 to `sum`).
    - Add the number of ways to make change without using the current coin, i.e., `dp[i][j] += dp[i-1][j]`.
    - Add the number of ways to make change using the current coin, i.e., `dp[i][j] += dp[i][j-coins[i-1]]`.
- `dp[n][sum]` will contain the total number of ways to make change for the given target sum using the available coin denominations.

Below is the implementation of the above approach:

C++

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

using namespace std;

// Returns total distinct ways to make sum using n coins of
// different denominations
int count(vector<int>& coins, int n, int sum)
{
    // 2d dp array where n is the number of coin
    // denominations and sum is the target sum
    vector<vector<int> > dp(n + 1, vector<int>(sum + 1, 0));

    // Represents the base case where the target sum is 0,
    // and there is only one way to make change: by not
    // selecting any coin
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {

            // Add the number of ways to make change without
            // using the current coin,
            dp[i][j] += dp[i - 1][j];

            if ((j - coins[i - 1]) >= 0) {

                // Add the number of ways to make change
                // using the current coin
                dp[i][j] += dp[i][j - coins[i - 1]];
```

```
            }
        }
    }
    return dp[n][sum];
}

// Driver Code
int main()
{
    vector<int> coins{ 1, 2, 3 };
    int n = 3;
    int sum = 5;
    cout << count(coins, n, sum);
    return 0;
}
```

Java

```
import java.util.*;

public class CoinChangeWays {
    // Returns total distinct ways to make sum using n coins of
    // different denominations
    static int count(List<Integer> coins, int n, int sum) {
        // 2D dp array where n is the number of coin
        // denominations and sum is the target sum
        int[][] dp = new int[n + 1][sum + 1];

        // Represents the base case where the target sum is 0,
        // and there is only one way to make change: by not
        // selecting any coin
        dp[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= sum; j++) {
                // Add the number of ways to make change without
                // using the current coin
                dp[i][j] += dp[i - 1][j];

                if ((j - coins.get(i - 1)) >= 0) {
                    // Add the number of ways to make change
                    // using the current coin
                    dp[i][j] += dp[i][j - coins.get(i - 1)];
                }
            }
        }
        return dp[n][sum];
    }
```

```
    # Represents the base case where the target sum is 0, and there is only one wa
    dp[0][0] = 1
    for i in range(1, n + 1):
        for j in range(target_sum + 1):
            # Add the number of ways to make change without using the current coir
            dp[i][j] += dp[i - 1][j]

            if j - coins[i - 1] >= 0:
                # Add the number of ways to make change using the current coin
                dp[i][j] += dp[i][j - coins[i - 1]]

    return dp[n][target_sum]

# Driver Code
if __name__ == "__main__":
    coins = [1, 2, 3]
    n = 3
    target_sum = 5
    print(count(coins, n, target_sum))
```

Output

```
5
```

Time complexity : O(N*sum)
Auxiliary Space : O(N*sum)

Count all combinations of coins to make a given value sum Dynamic Programming (Space Optimized):

In the above tabulation approach we are only using dp[i-1][j] and dp[i][j] etc, so we can do space optimization by only using a 1d dp array.

Follow the below steps to Implement the idea:

- Create a 1D dp array, **`dp[i]`** represents the number of ways to make the sum **`i`** using the given coin denominations.
- The outer loop iterates over the coins, and the inner loop iterates over the target sums. For each **`dp[j]`**, it calculates the number of ways to make

Count number of ways to cover a distance

Last Updated : 09 Apr, 2024

Given a distance ‘dist’, count total number of ways to cover the distance with 1, 2 and 3 steps.

Examples:

Input: *n = 3*
Output: *4*
Explanation: *Below are the four ways*
=> 1 step + 1 step + 1 step
=> 1 step + 2 step
=> 2 step + 1 step
=> 3 step

Input: *n = 4*
Output: *7*
Explanation: *Below are the four ways*
=> 1 step + 1 step + 1 step + 1 step
=> 1 step + 2 step + 1 step
=> 2 step + 1 step + 1 step
=> 1 step + 1 step + 2 step
=> 2 step + 2 step
=> 3 step + 1 step
=> 1 step + 3 step

Count number of ways to cover a distance using Recursion:

*There are n stairs, and a person is allowed to next step, skip one position or skip two positions. So there are n positions. The idea is standing at the ith position the person can move by **i+1, i+2, i+3** position. So a recursive function can be formed where at current index **i** the function is recursively called for **i+1, i+2** and **i+3** positions.*
*There is another way of forming the recursive function. To reach position **i,** a person has to jump either from **i-1, i-2 or i-3** position where **i** is the starting position.*

Step by step approach:

- Create a recursive function (*count(int n)*) which takes only one parameter.
- Check the base cases. If the value of **n** is less than **0** then return **0**, and if value of **n** is equal to zero then return **1** as it is the starting position.

- Call the function recursively with values **n-1, n-2** and **n-3** and sum up the values that are returned, i.e. ***sum = count(n-1) + count(n-2) + count(n-3).***
- Return the value of ***sum***.

Below are the implementation of the above approach:

C++ Java Python3 C# Javascript

```cpp
// A naive recursive C++ program to count number of ways to cover
// a distance with 1, 2 and 3 steps
#include<iostream>
using namespace std;

// Returns count of ways to cover 'dist'
int printCountRec(int dist)
{
    // Base cases
    if (dist<0)      return 0;
    if (dist==0)  return 1;

    // Recur for all previous 3 and add the results
    return printCountRec(dist-1) +
           printCountRec(dist-2) +
           printCountRec(dist-3);
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountRec(dist);
    return 0;
}
```

Output:

```
7
```

Time Complexity: $O(3^n)$, the time complexity of the above solution is exponential, a close upper bound is $O(3^n)$. From each state 3, a recursive function is called. So the upper bound for n states is $O(3^n)$.
Auxiliary Space: O(1), No extra space is required.

Count number of ways to cover a distance using Dynamic Programming (Memoization):

The problem have overlapping subproblems, meaning that the same subproblems are encountered multiple times during the recursive computation. For example, when calculating the number of ways to cover distance 'dist', we need to calculate the number of ways to cover distances 'dist-1', 'dist-2', and 'dist-3'. These subproblems are also encountered when calculating the number of ways to cover distances 'dist-2' and 'dist-3'.

***Memoization** solve this issue by storing the results of previously computed subproblems in a data structure, typically an array or a hash table.*

*Before making a recursive call, we first check if the result for the current distance has already been computed and stored in the **memo[]** array. If it has, we directly return the stored result, avoiding the recursive call. This significantly reduces the number of recursive calls and improves the performance of the algorithm, especially for larger distances. Once the result for the current distance is computed, we store it in the **memo[]** array for future reference.*

Below is the implementation of the above approach:

C++ Java Python JavaScript

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Returns count of ways to cover 'dist' using memoization
int printCountRecMemo(int dist, vector<int>& memo)
{
    // Base cases
    if (dist < 0)
        return 0;
    if (dist == 0)
        return 1;
```

```
    // Check if the value for 'dist' is already computed
    if (memo[dist] != -1)
        return memo[dist];

    // Recur for all previous 3 and add the results
    int ways = printCountRecMemo(dist - 1, memo)
               + printCountRecMemo(dist - 2, memo)
               + printCountRecMemo(dist - 3, memo);

    // Memoize the result for 'dist' for future use
    memo[dist] = ways;

    return ways;
}

// Function to calculate the count of ways with memoization
int countWays(int dist)
{
    vector<int> memo(
        dist + 1,
        -1); // Initialize memoization array with -1
    return printCountRecMemo(dist, memo);
}

// Driver program
int main()
{
    int dist = 4;
    cout << countWays(dist);
    return 0;
}
```

Output

7

Time Complexity: O(n)
Auxiliary Space: O(n)

Count number of ways to cover a distance using Dynamic Programming (Tabluation):

We start by initializing the base cases for covering distances 0, 1, and 2. For distance 0, there is only one way to cover it (do nothing). For distance 1, there is also only one way to cover it (take a step of size 1). For

distance 2, there are two ways to cover it (take two steps of size 1 or take a step of size 2).

*We use a bottom-up approach to fill in the **count[]** array. For each entry **count[i]**, we compute the number of ways to cover distance **i** by adding the number of ways to cover distances **i-1, i-2**, and **i-3.***

*Finally, we return the value of **count[dist],** which represents the number of ways to cover the given distance.*

Below is the implementation of the above approach:

C++ Java Python3 C# Javascript PHP

```cpp
// A Dynamic Programming based C++ program to count number of ways
// to cover a distance with 1, 2 and 3 steps
#include<iostream>
using namespace std;

int printCountDP(int dist)
{
    int count[dist+1];

    // Initialize base values. There is one way to cover 0 and 1
    // distances and two ways to cover 2 distance
     count[0] = 1;
     if(dist >= 1)
            count[1] = 1;
     if(dist >= 2)
              count[2] = 2;

    // Fill the count array in bottom up manner
    for (int i=3; i<=dist; i++)
       count[i] = count[i-1] + count[i-2] + count[i-3];

    return count[dist];
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountDP(dist);
    return 0;
}
```

Output

```
7
```

Time Complexity: O(n), Only one traversal of the array is needed.
Auxiliary Space: O(n), To store the values in a DP O(n) extra space is needed.

Count number of ways to cover a distance using Contant Space O(1):

> *From the above solution, we can observe that only three previous states are required to compute the value of current state. So, creates an array of size 3, where each element represents the number of ways to reach a certain distance. The base cases are initialized to 1 for distance 0 and 1, and 2 for distance 2. The remaining elements are filled using the recurrence relation, which states that the number of ways to reach a certain distance is the sum of the number of ways to reach the previous three distances. The final answer is the number of ways to reach the given distance, which is stored in the last element of the array.*

Below is the implementation of the above approach:

C++ Java Python3 C# Javascript

```cpp
// A Dynamic Programming based C++ program to count number of ways
#include<iostream>
using namespace std;

int printCountDP(int dist)
{
        //Create the array of size 3.
        int  ways[3] , n = dist;

        //Initialize the bases cases
        ways[0] = 1;
        ways[1] = 1;
        ways[2] = 2;

        //Run a loop from 3 to n
```

```
        //Bottom up approach to fill the array
        for(int i=3 ;i<=n ;i++)
            ways[i%3] = ways[(i-1)%3] + ways[(i-2)%3] + ways[(i-3)%3];

        return ways[n%3];
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountDP(dist);
    return 0;
}
```

Output

```
7
```

Time Complexity: O(n).
Auxiliary Space: O(1)

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great investment, the passion Sandeep sir has towards DSA/teaching is what made the huge difference." - **Gaurav | Placed at Amazon**

Before you move on to the world of development, **master the fundamentals of DSA** on which every advanced algorithm is built upon. Choose your preferred language and start learning today:

DSA In JAVA/C++
DSA In Python
DSA In JavaScript
Trusted by Millions, Taught by One- Join the best DSA Course Today!

68 Suggest improvement

Cutting a Rod | DP-13

Last Updated : 13 Apr, 2024

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as the following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 1   5   8   9  10  17  17  20
```

And if the prices are as follows, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 3   5   8   9  10  17  17  20
```

Recommended Problem

Solve Problem

Rod Cutting

Arrays Dynamic Programming +2 more Google

Solve Problem

Submission count: 1.1L

Method 1: A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently be solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.
Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.
cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

C++ Java C# Javascript Python3

```cpp
// A recursive solution for Rod cutting problem
#include <bits/stdc++.h>
#include <iostream>
#include <math.h>
using namespace std;

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }
```

```
/* Returns the best obtainable price for a rod of length n
   and price[] as prices of different pieces */
int cutRod(int price[], int index, int n)
{
    // base case
    if (index == 0) {
        return n * price[0];
    }
  // if n is 0 we cannot cut the rod anymore.
     if (n ==0) {
       return 0;}

    //At any index we have 2 options either
      //cut the rod of this length or not cut
      //it
    int notCut = cutRod(price,index - 1,n);
    int cut = INT_MIN;
    int rod_length = index + 1;

    if (rod_length <= n)
        cut = price[index]
               + cutRod(price,index,n - rod_length);

    return max(notCut, cut);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int size = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum Obtainable Value is "
         << cutRod(arr, size - 1, size);
    getchar();
    return 0;
}

//This code is contributed by Sanskar
```

Output

```
Maximum Obtainable Value is 22
```

Time Complexity: $O(2^n)$ where n is the length of the price array.

Space Complexity: O(n) where n is the length of the price array.

2) Overlapping Subproblems:

The following is a simple recursive implementation of the Rod Cutting problem.
The implementation simply follows the recursive structure mentioned above.

C++ Java C# Javascript Python3

```
// A memoization solution for Rod cutting problem
#include <bits/stdc++.h>
#include <iostream>
#include <math.h>
using namespace std;

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

/* Returns the best obtainable price for a rod of length n
   and price[] as prices of different pieces */
int cutRod(int price[], int index, int n,
           vector<vector<int> >& dp)
{
    // base case
    if (index == 0) {
        return n * price[0];
    }
    if (dp[index][n] != -1)
        return dp[index][n];
    // At any index we have 2 options either
    // cut the rod of this length or not cut
    // it
    int notCut = cutRod(price, index - 1, n,dp);
    int cut = INT_MIN;
    int rod_length = index + 1;

    if (rod_length <= n)
        cut = price[index]
              + cutRod(price, index, n - rod_length,dp);

    return dp[index][n]=max(notCut, cut);

}
/* Driver program to test above functions */
int main()
{
    int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int size = sizeof(arr) / sizeof(arr[0]);
    vector<vector<int> > dp(size,
                            vector<int>(size + 1, -1));
    cout << "Maximum Obtainable Value is "
         << cutRod(arr, size - 1, size, dp);
    getchar();
    return 0;
```

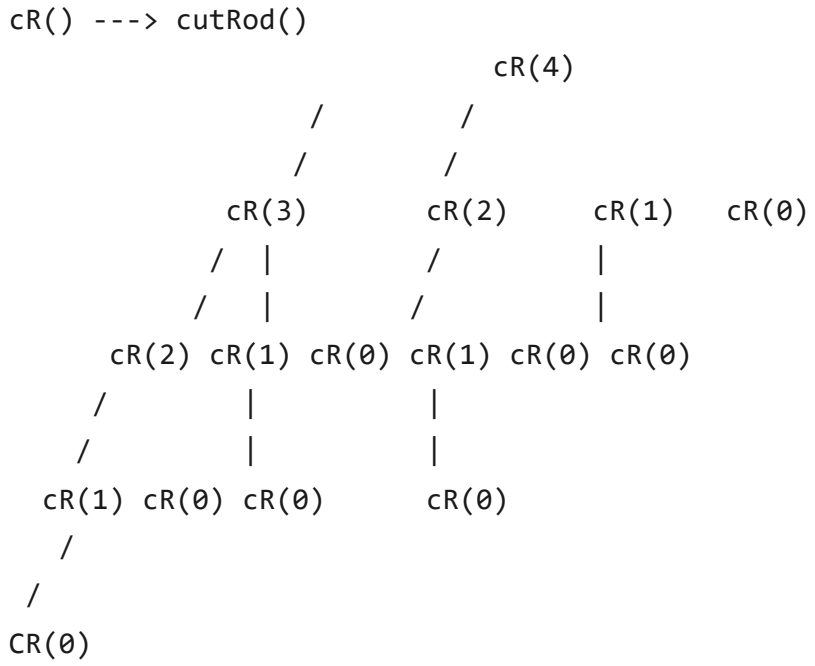
```
        }

        // This code is contributed by Sanskar
```

Output

```
Maximum Obtainable Value is 22
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(n^2)+O(n)$

Considering the above implementation, the following is the recursion tree for a Rod of length 4.



In the above partial recursion tree, cR(2) is solved twice. We can see that there are many subproblems that are solved again and again. Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Rod Cutting problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array val[] in a bottom-up manner.

C++ C Java C# Javascript PHP Python3

```cpp
// A Dynamic Programming solution for Rod cutting problem
#include<iostream>
#include <bits/stdc++.h>
#include<math.h>
using namespace std;

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
   int val[n+1];
   val[0] = 0;
   int i, j;

   // Build the table val[] in bottom up manner and return the last entry
   // from the table
   for (i = 1; i<=n; i++)
   {
       int max_val = INT_MIN;
       for (j = 0; j < i; j++)
         max_val = max(max_val, price[j] + val[i-j-1]);
       val[i] = max_val;
   }

   return val[n];
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    cout <<"Maximum Obtainable Value is "<<cutRod(arr, size);
    getchar();
    return 0;
```

```
    }

    // This code is contributed by shivanisinghss2110
```

Output

```
Maximum Obtainable Value is 22
```

The Time Complexity of the above implementation is O(n^2), which is much better than the worst-case time complexity of Naive Recursive implementation.

Space Complexity: O(n) as **val** array has been created.

3) Using the idea of Unbounded Knapsack.

This problem is very similar to the Unbounded Knapsack Problem, where there are multiple occurrences of the same item. Here the pieces of the rod. Now I will create an analogy between Unbounded Knapsack and the Rod Cutting Problem.

Unbounded Knapsack	Rod Cutting Problem
SIze of the array(N)	Length of the Rod(Length)
Value of the item(Val)	Price of the piece of Rod(price)
Weight array(Wt[])	Length Array(length[])
Maximum weight that the Knapsack can have(W)	The length of the main rod(W)

C++ C Java C# Javascript Python3

```cpp
// CPP program for above approach
#include <iostream>
using namespace std;

// Global Array for
// the purpose of memoization.
int t[9][9];

// A recursive program, using ,
// memoization, to implement the
// rod cutting problem(Top-Down).
int un_kp(int price[], int length[],
                    int Max_len, int n)
{

    // The maximum price will be zero,
    // when either the length of the rod
    // is zero or price is zero.
    if (n == 0 || Max_len == 0)
    {
        return 0;
    }

    // If the length of the rod is less
    // than the maximum length, Max_lene will
    // consider it.Now depending
    // upon the profit,
    // either Max_lene  we will take
    // it or discard it.
    if (length[n - 1] <= Max_len)
    {
        t[n][Max_len]
            = max(price[n - 1]
                      + un_kp(price, length,
                           Max_len - length[n - 1], n),
                  un_kp(price, length, Max_len, n - 1));
    }

    // If the length of the rod is
    // greater than the permitted size,
    // Max_len we will  not consider it.
    else
    {
        t[n][Max_len]
            = un_kp(price, length,
                              Max_len, n - 1);
    }

    // Max_lene Max_lenill return the maximum
    // value obtained, Max_lenhich is present
    // at the nth roMax_len and Max_length column.
    return t[n][Max_len];
}

/* Driver program to
test above functions */
int main()
```

```
{
    int price[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int n = sizeof(price) / sizeof(price[0]);
    int length[n];
    for (int i = 0; i < n; i++) {
        length[i] = i + 1;
    }
    int Max_len = n;

    // Function Call
    cout << "Maximum obtained value  is "
         << un_kp(price, length, n, Max_len) << endl;
}
```

Output

```
Maximum obtained value  is 22
```

Time Complexity: $O(n^2)$
Auxiliary Space: O(n), since n extra space has been taken.

4) Dynamic Programming Approach Iterative Solution

We will divide the problem into smaller sub-problems. Then using a 2-D matrix, we will calculate the maximum price we can achieve for any particular weight

C++ Java C# Javascript Python3

```
#include <algorithm>
#include <iostream>
using namespace std;

int cutRod(int prices[], int n)
{
    int mat[n + 1][n + 1];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                mat[i][j] = 0;
```

```
                }
                else {
                    if (i == 1) {
                        mat[i][j] = j * prices[i - 1];
                    }
                    else {
                        if (i > j) {
                            mat[i][j] = mat[i - 1][j];
                        }
                        else {
                            mat[i][j] = max(prices[i - 1]
                                            + mat[i][j - i],
                                        mat[i - 1][j]);
                        }
                    }
                }
            }
        }

        return mat[n][n];
    }

    int main()
    {
        int prices[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
        int n = sizeof(prices) / sizeof(prices[0]);

        cout << "Maximum obtained value is "
             << cutRod(prices, n) << endl;
    }
```

Output

```
Maximum obtained value is 22
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n^2)$

Dice Throw | DP-30

Last Updated : 06 Oct, 2023

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.

Recommended Problem

Dice throw

Mathematical Algorithms +2 more Amazon Microsoft

Solve Problem

Submission count: 30.1K

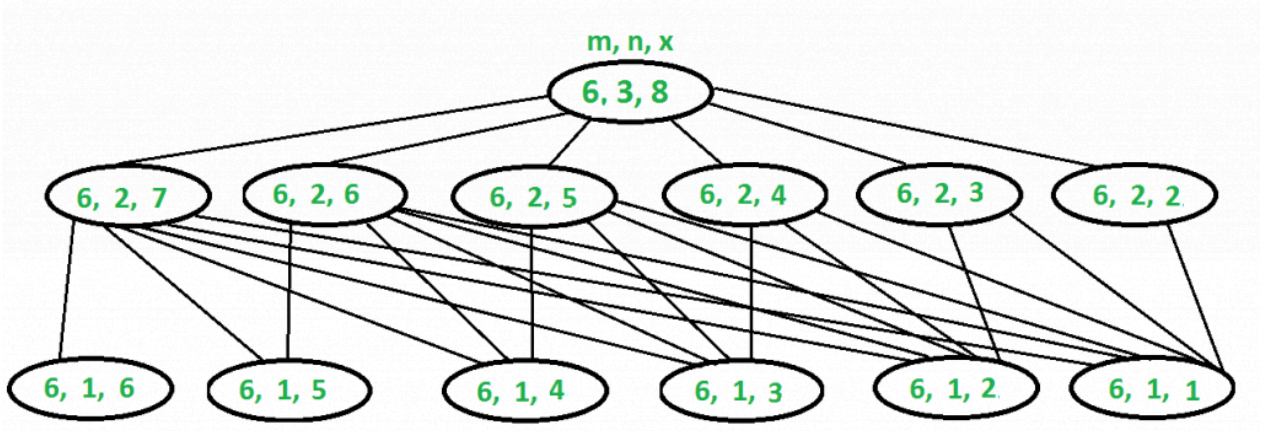
The **Naive approach** is to find all the possible combinations of values from n dice and keep on counting the results that sum to X.


```
Let the function to find X from n dice is: Sum(m, n, X)
The function can be represented as:
Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth
dice
               + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth
dice
               + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth
dice
                  ..................................................
                  ..................................................
                  ..................................................
              + Finding Sum (X - m) from (n - 1) dice plus m from nth
dice

So we can recursively write Sum(m, n, x) as following
Sum(m, n, X) = Sum(m, n - 1, X - 1) +
               Sum(m, n - 1, X - 2) +
```

```
..................... +
Sum(m, n - 1, X - m)
```

Why DP approach?

The above problem exhibits overlapping subproblems. See the below diagram. Also, see this recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:




```
Sum(6, 3, 8) = Sum(6, 2, 7) + Sum(6, 2, 6) + Sum(6, 2, 5) +
               Sum(6, 2, 4) + Sum(6, 2, 3) + Sum(6, 2, 2)
```

To evaluate Sum(6, 3, 8), we need to evaluate Sum(6, 2, 7) which can recursively written as following:

```
Sum(6, 2, 7) = Sum(6, 1, 6) + Sum(6, 1, 5) + Sum(6, 1, 4) +
               Sum(6, 1, 3) + Sum(6, 1, 2) + Sum(6, 1, 1)

We also need to evaluate Sum(6, 2, 6) which can recursively written
as following:
Sum(6, 2, 6) = Sum(6, 1, 5) + Sum(6, 1, 4) + Sum(6, 1, 3) +
               Sum(6, 1, 2) + Sum(6, 1, 1)
.........................................
.........................................
Sum(6, 2, 2) = Sum(6, 1, 1)
```

Please take a closer look at the above recursion. The sub-problems in RED are solved first time and sub-problems in BLUE are solved again (exhibit overlapping sub-problems). Hence, storing the results of the solved sub-problems saves time.

Following is implementation of Dynamic Programming approach.

C++

```cpp
// C++ program to find number of ways to get sum 'x' with 'n'
// dice where every dice has 'm' faces
#include <iostream>
#include <string.h>
using namespace std;

//  The main function that returns number of ways to get sum 'x'
//  with 'n' dice and 'm' with m faces.
int findWays(int m, int n, int x)
{
    // Create a table to store results of subproblems.  One extra
    // row and column are used for simplicity (Number of dice
    // is directly used as row index and sum is directly used
    // as column index).  The entries in 0th row and 0th column
    // are never used.
    int table[n + 1][x + 1];
    memset(table, 0, sizeof(table)); // Initialize all entries as 0

    // Table entries for only one dice
    for (int j = 1; j <= m && j <= x; j++)
        table[1][j] = 1;

    // Fill rest of the entries in table using recursive relation
    // i: number of dice, j: sum
    for (int i = 2; i <= n; i++)
```

```
            for (int j = 1; j <= x; j++)
                for (int k = 1; k <= m && k < j; k++)
                    table[i][j] += table[i-1][j-k];

    /* Uncomment these lines to see content of table
    for (int i = 0; i <= n; i++)
    {
      for (int j = 0; j <= x; j++)
        cout << table[i][j] << " ";
      cout << endl;
    } */
    return table[n][x];
}

// Driver program to test above functions
int main()
{
    cout << findWays(4, 2, 1) << endl;
    cout << findWays(2, 2, 3) << endl;
    cout << findWays(6, 3, 8) << endl;
    cout << findWays(4, 2, 5) << endl;
    cout << findWays(4, 3, 5) << endl;

    return 0;
}
```

Java

```
// Java program to find number of ways to get sum 'x' with 'n'
// dice where every dice has 'm' faces
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG {
    /* The main function that returns the number of ways to get sum 'x' with 'n' c
    public static long findWays(int m, int n, int x){

    /* Create a table to store the results of subproblems.
    One extra row and column are used for simplicity
    (Number of dice is directly used as row index and sum is directly used as colu
    The entries in 0th row and 0th column are never used. */
    long[][] table = new long[n+1][x+1];

    /* Table entries for only one dice */
    for(int j = 1; j <= m && j <= x; j++)
                table[1][j] = 1;
```

```
        /* Fill rest of the entries in table using recursive relation
        i: number of dice, j: sum */
        for(int i = 2; i <= n;i ++){
                    for(int j = 1; j <= x; j++){
                        for(int k = 1; k < j && k <= m; k++)
                            table[i][j] += table[i-1][j-k];
                    }
            }

            /* Uncomment these lines to see content of table
            for(int i = 0; i< n+1; i++){
                for(int j = 0; j< x+1; j++)
                    System.out.print(dt[i][j] + " ");
                System.out.println();
            } */

            return table[n][x];
        }

        // Driver Code
        public static void main (String[] args) {
            System.out.println(findWays(4, 2, 1));
            System.out.println(findWays(2, 2, 3));
            System.out.println(findWays(6, 3, 8));
            System.out.println(findWays(4, 2, 5));
            System.out.println(findWays(4, 3, 5));
        }
    }

    // This code is contributed by MaheshwariPiyush
```

Python3

```
# Python3 program to find the number of ways to get sum 'x' with 'n' dice
# where every dice has 'm' faces

# The main function that returns number of ways to get sum 'x'
# with 'n' dice and 'm' with m faces.
def findWays(m,n,x):
    # Create a table to store results of subproblems. One extra
    # row and column are used for simplicity (Number of dice
    # is directly used as row index and sum is directly used
    # as column index). The entries in 0th row and 0th column
    # are never used.
    table=[[0]*(x+1) for i in range(n+1)] #Initialize all entries as 0

    for j in range(1,min(m+1,x+1)): #Table entries for only one dice
```

```
            table[1][j]=1

        # Fill rest of the entries in table using recursive relation
        # i: number of dice, j: sum
        for i in range(2,n+1):
            for j in range(1,x+1):
                for k in range(1,min(m+1,j)):
                    table[i][j]+=table[i-1][j-k]

        #print(dt)
        # Uncomment above line to see content of table

        return table[-1][-1]

# Driver code
print(findWays(4,2,1))
print(findWays(2,2,3))
print(findWays(6,3,8))
print(findWays(4,2,5))
print(findWays(4,3,5))

# This code is contributed by MaheshwariPiyush
```

C#

```
// C# program to find number
// of ways to get sum 'x'
// with 'n' dice where every
// dice has 'm' faces
using System;

class GFG
{
// The main function that returns
// number of ways to get sum 'x'
// with 'n' dice and 'm' with m faces.
static int findWays(int m,
                    int n, int x)
{
    // Create a table to store
    // results of subproblems.
    // row and column are used
    // for simplicity (Number
    // of dice is directly used
    // as row index and sum is
    // directly used as column
    // index). The entries in 0th
    // row and 0th column are
```

```
        /* Uncomment these lines to see content of table
        for(int i = 0; i< n+1; i++){
            for(int j = 0; j< x+1; j++)
                System.out.print(dt[i][j] + " ");
            System.out.println();
        } */

        return table[n][x];
}

// Driver Code
document.write(findWays(4, 2, 1)+"<br>");
document.write(findWays(2, 2, 3)+"<br>");
document.write(findWays(6, 3, 8)+"<br>");
document.write(findWays(4, 2, 5)+"<br>");
document.write(findWays(4, 3, 5)+"<br>");


// This code is contributed by rag2127
```

Output :

```
0
2
21
4
6
```

Time Complexity: O(m * n * x) where m is number of faces, n is number of dice and x is given sum.

Auxiliary Space: O(n * x)
We can add the following two conditions at the beginning of findWays() to improve performance of the program for extreme cases (x is too high or x is too low)

C++

```
// When x is so high that sum can not go beyond x even when we
// get maximum value in every dice throw.
```

Subset Sum Problem

Last Updated : 23 Apr, 2024

Given a set of non-negative integers and a value **sum**, the task is to check if there is a subset of the given set whose sum is equal to the given **sum**.

Examples:

Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 9*
Output: *True*
Explanation: *There is a subset (4, 5) with sum 9.*

Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 30*
Output: *False*
Explanation: *There is no subset that add up to 30.*

Recommended Problem

Subset Sum Problem

Dynamic Programming Algorithms Amazon Microsoft

Solve Problem

Submission count: 1.9L

Subset Sum Problem using Recursion:

For the recursive approach, there will be two cases.

- *Consider the 'last' element to be a part of the subset. Now the **new required sum = required sum – value of 'last' element**.*
- *Don't include the 'last' element in the subset. Then the **new required sum = old required sum**.*

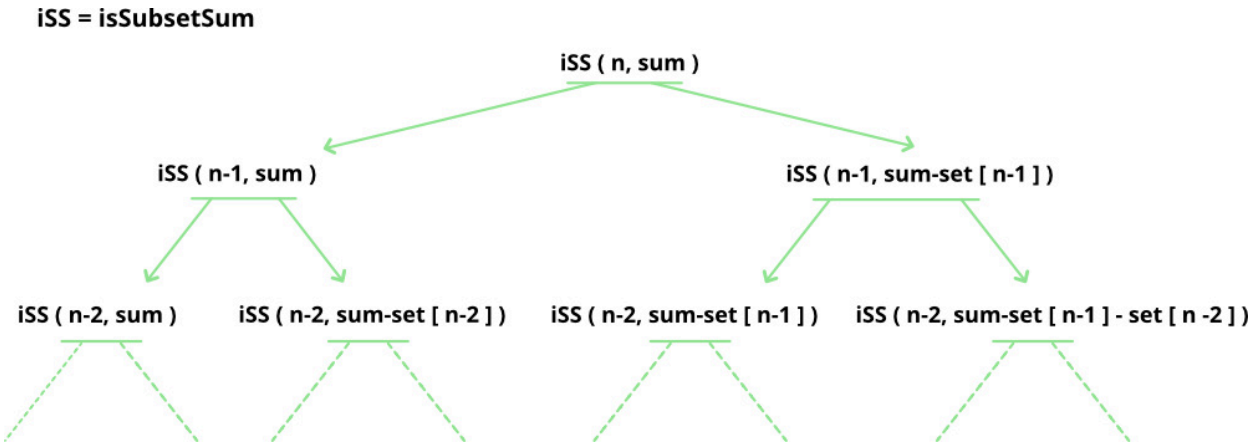
In both cases, the number of available elements decreases by 1.

Mathematically the recurrence relation will look like the following:

isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) |
isSubsetSum(set, n-1, sum-set[n-1])

Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n = 0
isSubsetSum(set, n, sum) = true, if sum = 0

The structure of the recursion tree will be like the following:



Structure of the recursion tree of the above recursion formula

Follow the below steps to implement the recursion:

- Build a recursive function and pass the index to be considered (here gradually moving from the last end) and the remaining sum amount.
- For each index check the base cases and utilize the above recursive call.
- If the answer is true for any recursion call, then there exists such a subset. Otherwise, no such subset exists.

Below is the implementation of the above approach.

C++ C Java Python3 C# Javascript PHP

```cpp
// A recursive solution for subset sum problem

#include <bits/stdc++.h>
using namespace std;

// Returns true if there is a subset
// of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0)
        return false;

    // If last element is greater than sum,
    // then ignore it
    if (set[n - 1] > sum)
        return isSubsetSum(set, n - 1, sum);

    // Else, check if sum can be obtained by any
    // of the following:
    // (a) including the last element
    // (b) excluding the last element
    return isSubsetSum(set, n - 1, sum)
```

```
                || isSubsetSum(set, n - 1, sum - set[n - 1]);
}

// Driver code
int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout << "Found a subset with given sum";
    else
        cout << "No subset with given sum";
    return 0;
}

// This code is contributed by shivanisinghss2110
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

- **Time Complexity:** O(2^n) The above solution may try all subsets of the given set in worst case. Therefore time complexity of the above solution is **exponential**. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).
- **Auxiliary Space:** O(n) where n is recursion stack space.

Subset Sum Problem using Memoization:

As seen in the previous recursion method, each state of the solution can be uniquely identified using two variables – the index and the remaining sum. So create a 2D array to store the value of each state to avoid recalculation of the same state.

Below is the implementation of the above approach:

C++ Java Python3 C# Javascript

```cpp
// CPP program for the above approach
#include <bits/stdc++.h>
using namespace std;

// Taking the matrix as globally
int tab[2000][2000];

// Check if possible subset with
// given sum is possible or not
int subsetSum(int a[], int n, int sum)
{
    // If the sum is zero it means
    // we got our expected sum
    if (sum == 0)
        return 1;

    if (n <= 0)
        return 0;

    // If the value is not -1 it means it
    // already call the function
    // with the same value.
    // it will save our from the repetition.
    if (tab[n - 1][sum] != -1)
        return tab[n - 1][sum];

    // If the value of a[n-1] is
    // greater than the sum.
    // we call for the next value
    if (a[n - 1] > sum)
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum);
    else
    {
        // Here we do two calls because we
        // don't know which value is
        // full-fill our criteria
        // that's why we doing two calls
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum) ||
                        subsetSum(a, n - 1, sum - a[n - 1]);
    }
}

// Driver Code
int main()
{
    // Storing the value -1 to the matrix
    memset(tab, -1, sizeof(tab));
    int n = 5;
    int a[] = {1, 5, 3, 7, 4};
    int sum = 12;

    if (subsetSum(a, n, sum))
    {
        cout << "YES" << endl;
    }
    else
        cout << "NO" << endl;
```

```
        /* This Code is Contributed by Ankit Kumar*/
    }
```

Output

```
YES
```

Complexity Analysis:

- **Time Complexity:** O(sum*n), where sum is the 'target sum' and 'n' is the size of array.
- **Auxiliary Space:** O(sum*n) + O(n) -> O(sum*n) = the size of 2-D array is sum*n and O(n)=auxiliary stack space.

Subset Sum Problem using Dynamic Programming:

To solve the problem in Pseudo-polynomial time we can use the Dynamic programming approach.

*So we will create a 2D array of size (n + 1) * (sum + 1) of type **boolean**. The state **dp[i][j]** will be **true** if there exists a subset of elements from **set[0 . . . i]** with **sum value = 'j'.***

The dynamic programming relation is as follows:

```
if (A[i-1] > j)
   dp[i][j] = dp[i-1][j]
else
   dp[i][j] = dp[i-1][j] OR dp[i-1][j-set[i-1]]
```

This means that if the current element has a value greater than the 'current sum value' we will copy the answer for previous cases and if the current sum value is greater than the 'ith' element we will see if any of the previous states have already experienced the **sum= j OR any previous states experienced a value 'j – set[i]'** which will solve our purpose.

Illustration:

See the below illustration for a better understanding:

*Consider **set[] = {3, 4, 5, 2}** and **sum = 6**.*

The table will look like the following where the column number represents the sum and the row number represents the index of set[]:

	0	1	2	3	4	5	6
no element (0)	T	F	F	F	F	F	F
0 (3)	T	F	F	T	F	F	F
1 (4)	T	F	F	T	T	F	F
2 (5)	T	F	F	T	T	T	F
3 (2)	T	F	T	T	T	T	T

Below is the implementation of the above approach:

C++ C Java Python3 C# Javascript PHP

```
// A Dynamic Programming solution
// for subset sum problem
#include <bits/stdc++.h>
using namespace std;

// Returns true if there is a subset of set[]
// with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if
    // there is a subset of set[0..j-1] with sum
    // equal to i
```

```
    bool subset[n + 1][sum + 1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[i][0] = true;

    // If sum is not 0 and set is empty,
    // then answer is false
    for (int i = 1; i <= sum; i++)
        subset[0][i] = false;

    // Fill the subset table in bottom up manner
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1])
                subset[i][j] = subset[i - 1][j];
            if (j >= set[i - 1])
                subset[i][j]
                    = subset[i - 1][j]
                      || subset[i - 1][j - set[i - 1]];
        }
    }

    return subset[n][sum];
}

// Driver code
int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout << "Found a subset with given sum";
    else
        cout << "No subset with given sum";
    return 0;
}
// This code is contributed by shivanisinghss2110
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

- **Time Complexity:** O(sum * n), where **n** is the size of the array.
- **Auxiliary Space:** O(sum*n), as the size of the 2-D array is **sum*n**.

Edit Distance

Last Updated : 04 Apr, 2024

Given two strings **str1** and **str2** of length **M** and **N** respectively and below operations that can be performed on **str1**. Find the minimum number of edits (operations) to convert '**str1**' into '**str2**'.

- **Operation 1 (INSERT)**: Insert any character before or after any index of **str1**
- **Operation 2 (REMOVE):** Remove a character of **str1**
- **Operation 3 (Replace):** Replace a character at any index of **str1** with some other character.

Note: All of the above operations are of equal cost.

Examples:

Input: *str1 = "geek", str2 = "gesek"*
Output: *1*
Explanation: *We can convert str1 into str2 by inserting a 's' between two consecutive 'e' in str2.*

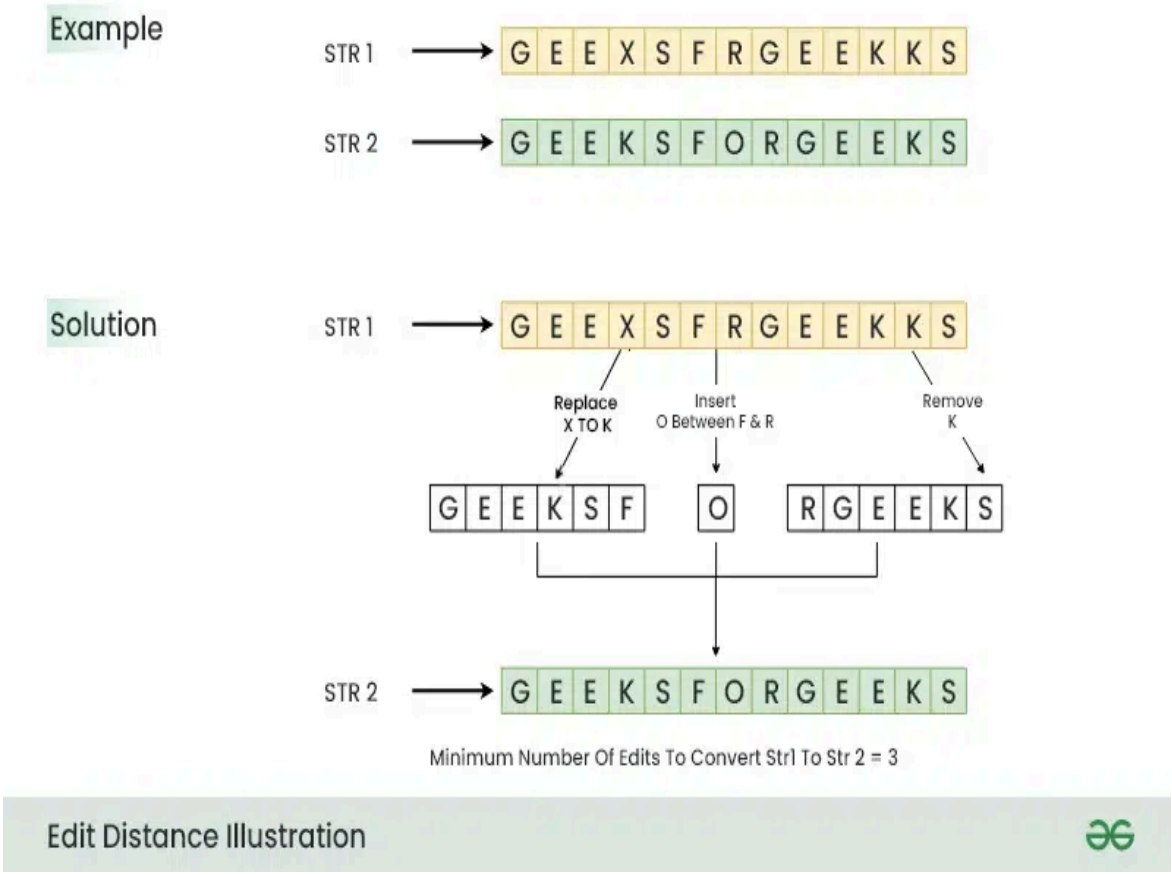
Input: *str1 = “cat”, str2 = “cut”*
Output: *1*
Explanation: *We can convert str1 into str2 by replacing ‘a’ with ‘u’.*

Input: *str1 = “sunday”, str2 = “saturday”*
Output: *3*
Explanation: *Last three and first characters are same. We basically need to convert “un” to “atur”. This can be done using below three operations. Replace ‘n’ with ‘r’, insert t, insert a*

Illustration of Edit Distance:

Let’s suppose we have ***str1=”GEEXSFRGEEKKS”*** *and* ***str2=”GEEKSFORGEEKS”***
Now to convert ***str1*** *into* ***str2*** *we would require* ***3*** *minimum operations:*
*Operation 1: Replace ‘****X****‘ to ‘****K****‘*
*Operation 2: Insert ‘****O****‘ between ‘****F****‘ and ‘****R****‘*
*Operation 3: Remove second last character i.e. ‘****K****‘*

Refer the below image for better understanding.



Edit Distance Illustration

Recommended Problem

Edit Distance

Strings Dynamic Programming +2 more

Amazon Microsoft +2 more

Solve Problem

Submission count: 1.9L

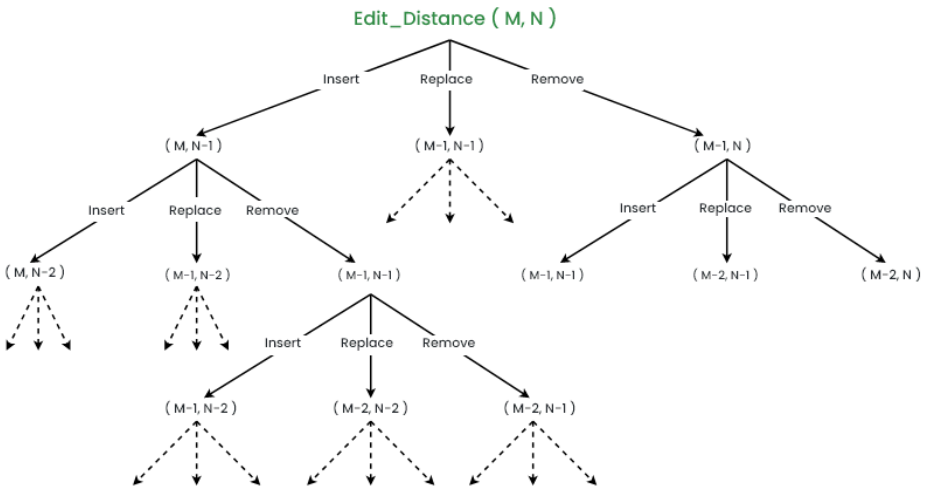
Edit Distance using Recursion:

Subproblems in Edit Distance:

*The idea is to process all characters one by one starting from either from **left** or **right** sides of both strings.*
*Let us process from the **right end** of the strings, there are two possibilities for every pair of characters being traversed, either they **match** or they **don't match**. If last characters of both string matches then there is no need to perform any operation So, recursively calculate the*

answer *for rest of part of the strings. When last characters do not match, we can perform all three operations to match the last characters in the given strings, i.e.* ***insert, replace, and remove.*** *We then recursively calculate the result for the remaining part of the string. Upon completion of these operations, we will select the minimum* ***answer****.*

Below is the ***recursive tree*** *for this problem:*



Recursion Tree For Edit Distance

When the last characters of strings matches. Make a recursive call ***EditDistance(M-1,N-1)*** *to calculate the answer for remaining part of the strings.*

When the last characters of strings don't matches. Make three recursive calls as show below:

- *Insert str1[N-1] at last of str2 : EditDistance(M, N-1)*
- *Replace str2[M-1] with str1[N-1] : EditDistance(M-1, N-1)*
- *Remove str2[M-1] : EditDistance(M-1, N)*

Recurrence Relations for Edit Distance:

- ***EditDistance(str1, str2, M, N) = EditDistance(str1, str2, M-1, N-1)***
- ***Case 1****: When the last character of both the strings are same*
- ***Case 2****: When the last characters are different*

- ***EditDistance(str1, str2, M, N) = 1 + Minimum{ EditDistance(str1, str2 ,M-1,N-1), EditDistance(str1, str2 ,M,N-1), EditDistance(str1, str2 ,M-1,N) }***

Base Case for Edit Distance:

- ***Case 1**: When **str1** becomes empty i.e. **M=0***
  - ***return N**, as it require **N** characters to convert an empty string to **str1** of size **N***
- ***Case 2**: When str2 becomes empty i.e. **N=0***
  - ***return M**, as it require **M** characters to convert an empty string to **str2** of size **M***

Below is the implementation of the above recursive solution.

C++ C Java Python3 C# JavaScript

```cpp
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include <bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z) { return min(min(x, y), z); }

int editDist(string str1, string str2, int m, int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0)
        return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0)
        return m;

    // If last characters of two strings are same, nothing
    // much to do. Get the count for
    // remaining strings.
    if (str1[m - 1] == str2[n - 1])
        return editDist(str1, str2, m - 1, n - 1);
```

```
        // If last characters are not same, consider all three
        // operations on last character of first string,
        // recursively compute minimum cost for all three
        // operations and take minimum of three values.
        return 1
               + min(editDist(str1, str2, m, n - 1), // Insert
                     editDist(str1, str2, m - 1, n), // Remove
                     editDist(str1, str2, m - 1,
                              n - 1) // Replace
               );
    }

    // Driver code
    int main()
    {
        // your code goes here
        string str1 = "GEEXSFRGEEKKS";
        string str2 = "GEEKSFORGEEKS";

        cout << editDist(str1, str2, str1.length(),
                         str2.length());

        return 0;
    }
```

Output

```
3
```

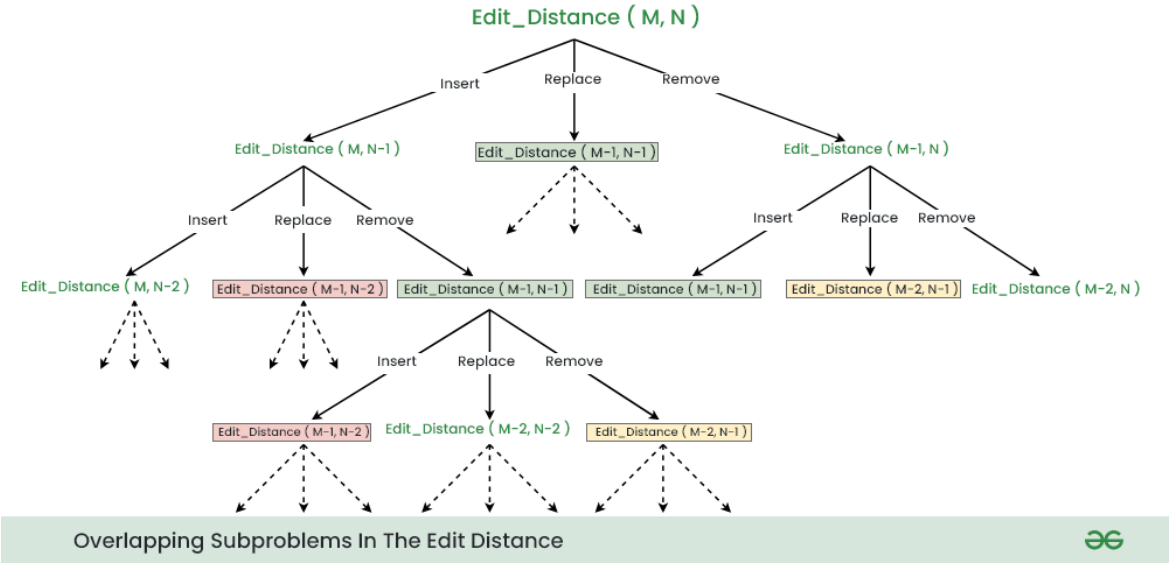
Time Complexity: O(3^m), when length of “str1” >= length of “str2” and O(3^n), when length of “str2” > length of “str1”. Here m=length of “str1 and n=length of “str2”
Auxiliary Space: O(m), when length of “str1” >= length of “str2” and O(n), when length of “str2” > length of “str1”. Here m=length of “str1 and n=length of “str2”

Edit Distance Using Dynamic Programming (Memoization):

In the above recursive approach, there are several overlapping subproblems:
Edit_Distance(M-1, N-1) *is called Three times*
Edit_Distance(M-1, N-2) *is called Two times*
Edit_Distance(M-2, N-1) *is called Two times. And so on…*

So, we can use Memoization technique to store the result of each subproblems to avoid recalculating the result again and again.

Below are the illustration of overlapping subproblems during the recursion.



Overlapping Subproblems In The Edit Distance

Below is the implementation of Edit Distance Using Dynamic Programming (Memoization):

C++ C Java Python3 C# JavaScript

```cpp
#include <bits/stdc++.h>
using namespace std;
int minDis(string s1, string s2, int n, int m,
           vector<vector<int> >& dp)
{

    // If any string is empty,
    // return the remaining characters of other string

    if (n == 0)
        return m;

    if (m == 0)
        return n;

    // To check if the recursive tree
    // for given n & m has already been executed

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

    // If characters are equal, execute
```

```
        // recursive function for n-1, m-1

        if (s1[n - 1] == s2[m - 1]) {
            return dp[n][m] = minDis(s1, s2, n - 1, m - 1, dp);
        }
        // If characters are nt equal, we need to
        // find the minimum cost out of all 3 operations.
        // 1. insert 2.delete 3.replace
        else {
            int insert, del, replace; // temp variables

            insert = minDis(s1, s2, n, m - 1, dp);
            del = minDis(s1, s2, n - 1, m, dp);
            replace = minDis(s1, s2, n - 1, m - 1, dp);
            return dp[n][m]
                   = 1 + min(insert, min(del, replace));
        }
    }

    // Driver program
    int main()
    {

        string str1 = "GEEXSFRGEEKKS";
        string str2 = "GEEKSFORGEEKS";

        int n = str1.length(), m = str2.length();
        vector<vector<int> > dp(n + 1, vector<int>(m + 1, -1));

        cout << minDis(str1, str2, n, m, dp);
        return 0;
    }
```

Output

```
3
```

Time Complexity: O(m x n)
Auxiliary Space: O(m *n)+O(m+n) , (m*n) extra array space and (m+n) recursive stack space.

Edit Distance Using Dynamic Programming (Bottom-Up Approach):

Use a table to store solutions of subproblems to avoiding recalculate the same subproblems multiple times. By doing this, if same subproblems

repeated during, we retrieve the solutions from the table itself.

Below are the steps to convert the recursive approach to Bottom up approach:

1. Choosing Dimensions of Table: The state of smaller sub-problems depends on the input parameters **m** and **n** because at least one of them will decrease after each recursive call. So we need to construct a 2D table **dp[][]** to store the solution of the sub-problems.

2. Choosing Correct size of Table: The size of the 2D table will be equal to the total number of different subproblems, which is equal to **(m + 1)*(n + 1)**. As both **m** and **n** are decreasing by **1** during the recursive calls and reaching the value **0**. So **m + 1** possibilities for the first parameter and **n + 1** possibilities for the second parameter. Total number of possible subproblems = **(m + 1)*(n + 1)**.

3. Filling the table: It consist of two stages, table initialization and building the solution from the smaller subproblems:

- **Table initialization:** Before building the solution, we need to initialize the table with the smaller version of the solution i.e. base case. Here **m = 0** and **n = 0** is the situation of the base case, so we initialize first-column **dp[i][0]** with **i** and first-row **dp[0][j]** with **j**.
- **Building the solution of larger problems from the smaller subproblems:** We can easily define the iterative structure by using the recursive structure of the above recursive solution.

  - ***if (str1[i – 1] == str2[j – 1])*** *dp[i][j] = dp[i – 1][j – 1];*
  - ***if (str1[i – 1] != str2[j – 1])*** *dp[i][j] = 1 + min(dp[i][j – 1], dp[i – 1][j], dp[i – 1][j – 1]);*

4. Returning final solution: After filling the table iteratively, our final solution gets stored at the bottom right corner of the 2-D table i.e. we return Edit[m][n] as an output.

Below is the implementation of the above algorithm:

C++ C Java Python3 C# JavaScript

```cpp
// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include <bits/stdc++.h>
using namespace std;

// Utility function to find the minimum of three numbers
int min(int x, int y, int z) { return min(min(x, y), z); }

int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m + 1][n + 1];

    // Fill d[][] in bottom up manner
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i == 0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j == 0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];

            // If the last character is different, consider
            // all possibilities and find the minimum
            else
                dp[i][j]
                    = 1
                      + min(dp[i][j - 1], // Insert
                            dp[i - 1][j], // Remove
                            dp[i - 1][j - 1]); // Replace
        }
    }

    return dp[m][n];
}

// Driver code
int main()
{
    // your code goes here
    string str1 = "GEEXSFRGEEKKS";
    string str2 = "GEEKSFORGEEKS";

    cout << editDistDP(str1, str2, str1.length(),
                       str2.length());
```

```
    return 0;
}
```

Output

```
3
```

Time Complexity: O(m x n)
Auxiliary Space: O(m x n)

Edit Distance Using Dynamic Programming (Optimization in Space Complexity):

> ***Optimized Space Complexity Solution****: In the above bottom up approach we require* ***O(m x n)*** *space. Let's take an observation and try to optimize our space complexity:*
>
> *To fill a row in* ***DP*** *array we require only one row i.e. the upper row. For example, if we are filling the row where* ***i=10*** *in* ***DP*** *array then we require only values of* ***9th*** *row. So we simply create a* ***DP*** *array of* ***2 x str1 length****. This approach reduces the space complexity from* ***O(N*M) to O(2*N).***

Below is the implementation of the above approach:

C++ C Java Python3 C# JavaScript

```
// A Space efficient Dynamic Programming
// based C++ program to find minimum
// number operations to convert str1 to str2
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // space optimization
    int editDistance(string s, string t)
    {
        int m = s.size();
```

```
        int n = t.size();

        vector<int> prev(n + 1, 0), curr(n + 1, 0);

        for (int j = 0; j <= n; j++) {
            prev[j] = j;
        }

        for (int i = 1; i <= m; i++) {
            curr[0] = i;
            for (int j = 1; j <= n; j++) {
                if (s[i - 1] == t[j - 1]) {
                    curr[j] = prev[j - 1];
                }
                else {
                    int mn
                        = min(1 + prev[j], 1 + curr[j - 1]);
                    curr[j] = min(mn, 1 + prev[j - 1]);
                }
            }
            prev = curr;
        }

        return prev[n];
    }
};

int main()
{
    string s = "GEEXSFRGEEKKS", t = "GEEKSFORGEEKS";

    Solution ob;
    int ans = ob.editDistance(s, t);
    cout << ans << "\n";

    return 0;
}
```

Output

```
3
```

Time Complexity: O(M x N) where M and N are lengths of the string
Auxiliary Space: O(N), Length of the str2

Edit Distance Using Dynamic Programming (Further Optimization in Space Complexity):

As discussed the above approach uses two 1-D arrays, now the question is can we achieve our task by using only a single 1-D array?

The answer is Yes and it requires a simple observation as mentioned below:
In the previous approach The curr[] array is updated using 3 values only :

***Value 1: curr[j] = prev[j-1]** when str1[i-1] is equal to str2[j-1]*
***Value 2: curr[j] = prev[j]** when str1[i-1] is not equal to str2[j-1]*
***Value 3: curr[j] = curr[j-1]** when str1[i-1] is not equal to str2[j-1]*

By keeping the track of these three values we can achiever our task using only a single 1-D array

Below is the code implementation of the approach:

C++ Java Python3 C# JavaScript

```cpp
// A Space efficient Dynamic Programming
// based C++ program to find minimum
// number operations to convert str1 to str2
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // space optimization
    int editDistance(string str1, string str2)
    {
        int m = str1.size();
        int n = str2.size();
        int previous;
        vector<int> curr(n + 1, 0);

        for (int j = 0; j <= n; j++) {
            curr[j] = j;
        }
        for (int i = 1; i <= m; i++) {
            previous = curr[0];
            curr[0] = i;
            for (int j = 1; j <= n; j++) {
                int temp = curr[j];
                if (str1[i - 1] == str2[j - 1]) {
                    curr[j] = previous;
                }
                else {
```

```
                        curr[j] = 1
                                  + min({ previous, curr[j - 1],
                                          curr[j] });
                    }
                    previous = temp;
                }
            }
            return curr[n];
        }
    };

    int main()
    {
        string str1 = "GEEXSFRGEEKKS", str2 = "GEEKSFORGEEKS";

        Solution ob;
        int ans = ob.editDistance(str1, str2);
        cout << ans << "\n";

        return 0;
    }
```

Output

```
3
```

Time Complexity: O(M*N)
Auxiliary Space: O(N)

Real-World Applications of Edit Distance:

- Spell Checking and Auto-Correction
- DNA Sequence Alignment
- Plagiarism Detection
- Natural Language Processing
- Version Control Systems
- String Matching

https://youtu.be/Thv3TfsZVpw

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great investment, the passion Sandeep sir has towards DSA/teaching is what

Egg Dropping Puzzle | DP-11

Last Updated : 23 Mar, 2023

The following is a description of the instance of this famous puzzle involving N = 2 eggs and a building with K = 36 floors.
Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks when dropped, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.
- It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor does not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg droppings that are guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that the total number of trials is minimized.

Note: In this post, we will discuss a solution to a general problem with ‘N’ eggs and ‘K’ floors

Recommended Problem

Egg Dropping Puzzle

Solve Problem

Dynamic Programming Algorithms VMWare Amazon +15 more

Submission count: 1.3L

Egg Dropping Puzzle using Recursion:

To solve the problem follow the below idea:

The solution is to try dropping an egg from every floor(from 1 to K) and recursively calculate the minimum number of droppings needed in the worst case. The floor which gives the minimum value in the worst case is going to be part of the solution.
In the following solutions, we return the minimum number of trials in the worst case; these solutions can be easily modified to print the floor numbers of every trial also.

What is worst case scenario?

Worst case scenario gives the user the surety of the threshold floor. For example- If we have '1' egg and 'K' floors, we will start dropping the egg from the first floor till the egg breaks suppose on the 'Kth' floor so the number of tries to give us surety is 'K'.

1. Optimal Substructure:

When we drop an egg from floor x, there can be two cases (1) The egg breaks (2) The egg doesn't break.

1. If the egg breaks after dropping from ‘xth’ floor, then we only need to check for floors lower than ‘x’ with remaining eggs as some floors should exist lower than ‘x’ in which the egg would not break, so the problem reduces to x-1 floors and n-1 eggs.
2. If the egg doesn’t break after dropping from the ‘xth’ floor, then we only need to check for floors higher than ‘x’; so the problem reduces to ‘k-x’ floors and n eggs.

Since we need to minimize the number of trials in the *worst* case, we take a maximum of two cases. We consider the max of the above two cases for every floor and choose the floor which yields the minimum number of trials.

Below is the illustration of the above approach:

K ==> Number of floors
N ==> Number of Eggs

eggDrop(N, K) ==> Minimum number of trials needed to find the critical floor in worst case.
eggDrop(N, K) = 1 + min{max(eggDrop(N – 1, x – 1), eggDrop(N, K – x)), where x is in {1, 2, …, K}}

Concept of worst case:

Let there be ‘2’ eggs and ‘2’ floors then-:
If we try throwing from ‘1st’ floor:
Number of tries in worst case= 1+max(0, 1)

0=>If the egg breaks from first floor then it is threshold floor (best case possibility).
1=>If the egg does not break from first floor we will now have ‘2’ eggs and 1 floor to test which will give answer as ‘1’.(worst case possibility)

We take the worst case possibility in account, so 1+max(0, 1)=2
If we try throwing from ‘2nd’ floor:
Number of tries in worst case= 1+max(1, 0)

1=>If the egg breaks from second floor then we will have 1 egg and 1 floor to find threshold floor.(Worst Case)
0=>If egg does not break from second floor then it is threshold floor.(Best Case)
We take worst case possibility for surety, so 1+max(1, 0)=2.
The final answer is min(1st, 2nd, 3rd....., kth floor)
So answer here is '2'.

Below is the implementation of the above approach:

C++

```
// C++ program for the above approach

#include <bits/stdc++.h>
using namespace std;

// A utility function to get
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Function to get minimum
// number of trials needed in worst
// case with n eggs and k floors
int eggDrop(int n, int k)
{
    // If there are no floors,
    // then no trials needed.
    // OR if there is one floor,
    // one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one
    // egg and k floors
    if (n == 1)
        return k;

    int min = INT_MAX, x, res;

    // Consider all droppings from
    // 1st floor to kth floor and
    // return the minimum of these
    // values plus 1.
```

```
    for (x = 1; x <= k; x++) {
        res = max(eggDrop(n - 1, x - 1), eggDrop(n, k - x));
        if (res < min)
            min = res;
    }

    return min + 1;
}

// Driver code
int main()
{
    int n = 2, k = 10;
    cout << "Minimum number of trials "
            "in worst case with "
         << n << " eggs and " << k << " floors is "
         << eggDrop(n, k) << endl;
    return 0;
}

// This code is contributed
// by Akanksha Rai
```

C

```
// C program for the above approach

#include <limits.h>
#include <stdio.h>

// A utility function to get
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

/* Function to get minimum number of
 trials needed in worst case with n eggs
 and k floors */
int eggDrop(int n, int k)
{
    // If there are no floors, then no
    // trials needed. OR if there is
    // one floor, one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one egg and
    // k floors
    if (n == 1)
```

```
            return k;

        int min = Integer.MAX_VALUE;
        int x, res;

        // Consider all droppings from
        // 1st floor to kth floor and
        // return the minimum of these
        // values plus 1.
        for (x = 1; x <= k; x++) {
            res = Math.max(eggDrop(n - 1, x - 1),
                           eggDrop(n, k - x));
            if (res < min)
                min = res;
        }

        return min + 1;
    }

    // Driver code
    public static void main(String args[])
    {
        int n = 2, k = 10;
        System.out.print("Minimum number of "
                         + "trials in worst case with " + n
                         + " eggs and " + k + " floors is "
                         + eggDrop(n, k));
    }
    // This code is contributed by Ryuga.
}
```

Python 3

```
import sys

# Function to get minimum number of trials
# needed in worst case with n eggs and k floors


def eggDrop(n, k):

    # If there are no floors, then no trials
    # needed. OR if there is one floor, one
    # trial needed.
    if (k == 1 or k == 0):
        return k

    # We need k trials for one egg
```

```
    # and k floors
    if (n == 1):
        return k

    min = sys.maxsize

    # Consider all droppings from 1st
    # floor to kth floor and return
    # the minimum of these values plus 1.
    for x in range(1, k + 1):

        res = max(eggDrop(n - 1, x - 1),
                  eggDrop(n, k - x))
        if (res < min):
            min = res

    return min + 1


# Driver Code
if __name__ == "__main__":

    n = 2
    k = 10
    print("Minimum number of trials in worst case with",
          n, "eggs and", k, "floors is", eggDrop(n, k))

# This code is contributed by ita_c
```

C#

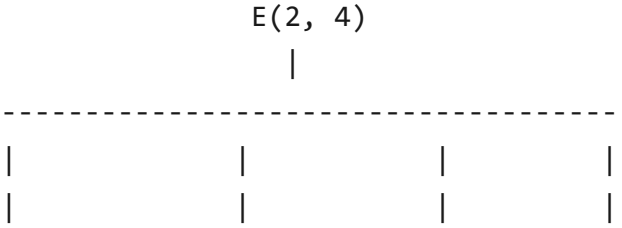
```
using System;

class GFG {

    /* Function to get minimum number of
    trials needed in worst case with n
    eggs and k floors */
    static int eggDrop(int n, int k)
    {
        // If there are no floors, then
        // no trials needed. OR if there
        // is one floor, one trial needed.
        if (k == 1 || k == 0)
            return k;

        // We need k trials for one egg
        // and k floors
```

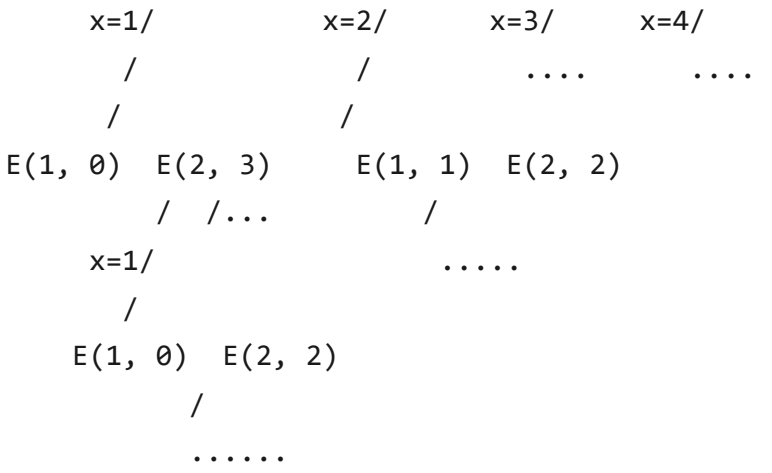
```
        // We need k trials for one egg
        // and k floors
        if (n == 1)
            return k;

        let min = Number.MAX_VALUE;
        let x, res;

        // Consider all droppings from
        // 1st floor to kth floor and
        // return the minimum of these
        // values plus 1.
        for (x = 1; x <= k; x++)
        {
            res = Math.max(eggDrop(n - 1, x - 1),
                           eggDrop(n, k - x));
            if (res < min)
                min = res;
        }
        return min + 1;
    }

    // Driver code
    let n = 2, k = 10;
    document.write("Minimum number of "
                         + "trials in worst case with "
                         + n + " eggs and " + k
                         + " floors is " + eggDrop(n, k));

    // This code is contributed by avanitrachhadiya2155
```

Output

```
Minimum number of trials in worst case with 2 eggs and 10 floors is 4
```

Note: The above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will be many repeated subproblems when you draw the complete recursion tree even for small values of N and K.





Partial recursion tree for 2 eggs and 4 floors.

Time Complexity: As there is a case of overlapping sub-problems the time complexity is exponential.
Auxiliary Space: O(1). As there was no use of any data structure for storing values.

Egg Dropping Puzzle using Dynamic Programming:

To solve the problem follow the below idea:

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array eggFloor[][] in a bottom-up manner.

> *In this approach, we work on the same idea as described above neglecting the case of calculating the answers to sub-problems again and again. The approach will be to make a table that will store the results of sub-problems so that to solve a sub-problem, would only require a look-up from the table which will take constant time, which earlier took exponential time.*

Find the longest path in a matrix with given constraints

Last Updated : 05 May, 2023

Given a n*n matrix where all numbers are distinct, find the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1.
We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1.

Example:

```
Input:  mat[][] = {{1, 2, 9}
                   {5, 3, 8}
                   {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.
```


The idea is simple, we calculate longest path beginning with every cell. Once we have computed longest for all cells, we return maximum of all longest paths. One important observation in this approach is many overlapping sub-problems. Therefore this problem can be optimally solved using Dynamic Programming.

Algorithm:

Step 1: Initialize a matrix and set its size to n x n.
Step 2: Define a function “findLongestFromACell” that takes in a cell’s row and column index, the matrix, and a lookup table. If the cell is out of bounds or the subproblem has already been solved, return 0 or the previously calculated value in the lookup table, respectively.
Step 3: Define four integer variables to store the length of the path in each of the four possible directions. Check if the adjacent cell in each direction satisfies the constraints and if so, recursively call the function for that cell and update the corresponding direction’s length variable.
Step 4: Return the maximum length of the four directions plus one, and store it in the lookup table.
Step 5: Define a function “finLongestOverAll” that takes in the matrix. Initialize a result variable to 1 and a lookup table as a two-dimensional array of size n x n, filled with -1.
Step 6: For each cell in the matrix, call “findLongestFromACell” and update the result as needed.
Step 7: Return the result.

Below is Dynamic Programming based implementation that uses a lookup table dp[][] to check if a problem is already solved or not.

C++

```
// C++ program to find the longest path in a matrix
// with given constraints
#include <bits/stdc++.h>
#define n 3
using namespace std;

// Returns length of the longest path beginning with
```

```
// mat[i][j]. This function mainly uses lookup table
// dp[n][n]
int findLongestFromACell(int i, int j, int mat[n][n],
                         int dp[n][n])
{
    if (i < 0 || i >= n || j < 0 || j >= n)
        return 0;

    // If this subproblem is already solved
    if (dp[i][j] != -1)
        return dp[i][j];

    // To store the path lengths in all the four directions
    int x = INT_MIN, y = INT_MIN, z = INT_MIN, w = INT_MIN;

    // Since all numbers are unique and in range from 1 to
    // n*n, there is atmost one possible direction from any
    // cell
    if (j < n - 1 && ((mat[i][j] + 1) == mat[i][j + 1]))
        x = 1 + findLongestFromACell(i, j + 1, mat, dp);

    if (j > 0 && (mat[i][j] + 1 == mat[i][j - 1]))
        y = 1 + findLongestFromACell(i, j - 1, mat, dp);

    if (i > 0 && (mat[i][j] + 1 == mat[i - 1][j]))
        z = 1 + findLongestFromACell(i - 1, j, mat, dp);

    if (i < n - 1 && (mat[i][j] + 1 == mat[i + 1][j]))
        w = 1 + findLongestFromACell(i + 1, j, mat, dp);

    // If none of the adjacent fours is one greater we will
    // take 1 otherwise we will pick maximum from all the
    // four directions
    return dp[i][j] = max({x, y, z, w, 1});
}

// Returns length of the longest path beginning with any
// cell
int finLongestOverAll(int mat[n][n])
{
    int result = 1; // Initialize result

    // Create a lookup table and fill all entries in it as
    // -1
    int dp[n][n];
    memset(dp, -1, sizeof dp);

    // Compute longest path beginning from all cells
    for (int i = 0; i < n; i++) {
```

```
        for (int j = 0; j < n; j++) {
            if (dp[i][j] == -1)
                findLongestFromACell(i, j, mat, dp);

            // Update result if needed
            result = max(result, dp[i][j]);
        }
    }

    return result;
}

// Driver program
int main()
{
    int mat[n][n]
        = { { 1, 2, 9 }, { 5, 3, 8 }, { 4, 6, 7 } };
    cout << "Length of the longest path is "
         << finLongestOverAll(mat);
    return 0;
}
```

Java

```
// Java program to find the longest path in a matrix
// with given constraints

class GFG {
    public static int n = 3;

    // Function that returns length of the longest path
    // beginning with mat[i][j]
    // This function mainly uses lookup table dp[n][n]
    static int findLongestFromACell(int i, int j,
                                    int mat[][], int dp[][])
    {
        // Base case
        if (i < 0 || i >= n || j < 0 || j >= n)
            return 0;

        // If this subproblem is already solved
        if (dp[i][j] != -1)
            return dp[i][j];

        // To store the path lengths in all the four
        // directions
        int x = Integer.MIN_VALUE, y = Integer.MIN_VALUE,
            z = Integer.MIN_VALUE, w = Integer.MIN_VALUE;
```

```
    {
        int[][] mat = new int[][] { new int[] { 1, 2, 9 },
                                    new int[] { 5, 3, 8 },
                                    new int[] { 4, 6, 7 } };
        Console.WriteLine("Length of the longest path is "
                          + finLongestOverAll(mat));
    }
}

// This code is contributed by Shrikant13
```

Output

```
Length of the longest path is 4
```

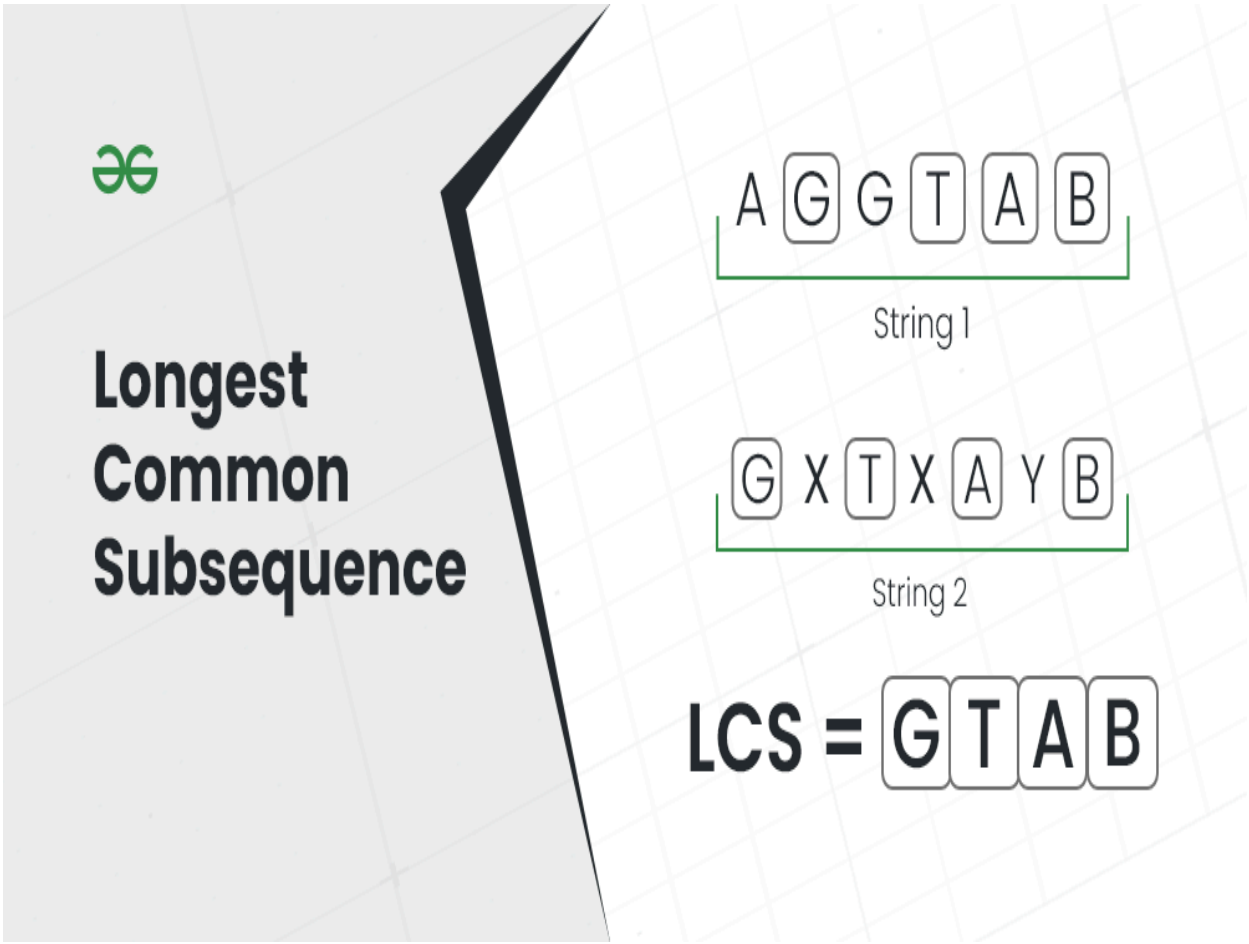
Time complexity of the above solution is $O(n^2)$. It may seem more at first look. If we take a closer look, we can notice that all values of dp[i][j] are computed only once.
Auxiliary Space: O(N x N), since N x N extra space has been taken.

Longest Common Subsequence (LCS)

Last Updated : 03 May, 2024

Given two strings, **S1** and **S2**, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.

> *A **longest common subsequence (LCS)** is defined as the longest subsequence which is common in all given input sequences.*



Longest Common Subsequence

Examples:

Input: *S1 = "AGGTAB", S2 = "GXTXAYB"*
Output: *4*
Explanation: *The longest subsequence which is present in both strings is "GTAB".*

Input: *S1 = "BD", S2 = "ABCD"*
Output: *2*
Explanation: *The longest subsequence which is present in both strings is "BD".*

Longest Common Subsequence (LCS) using Recursion:

Generate all the possible subsequences and find the longest among them that is present in both strings using recursion.

Follow the below steps to implement the idea:

- Create a recursive function [say **lcs()**].

- Check the relation between the First characters of the strings that are not yet processed.
  - Depending on the relation call the next recursive function as mentioned above.
- Return the length of the LCS received as the answer.

Below is the implementation of the recursive approach:

C++ C Java Python C# Javascript PHP

```cpp
// A Naive recursive implementation of LCS problem

#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1], Y[0..n-1]
int lcs(string X, string Y, int m, int n)
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return 1 + lcs(X, Y, m - 1, n - 1);
    else
        return max(lcs(X, Y, m, n - 1),
                   lcs(X, Y, m - 1, n));
}

// Driver code
int main()
{
    string S1 = "AGGTAB";
    string S2 = "GXTXAYB";
    int m = S1.size();
    int n = S2.size();

    cout << "Length of LCS is " << lcs(S1, S2, m, n);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output

```
Length of LCS is 4
```

Time Complexity: $O(2^{m+n})$
Auxiliary Space: O(1)

Longest Common Subsequence (LCS) using Memoization:

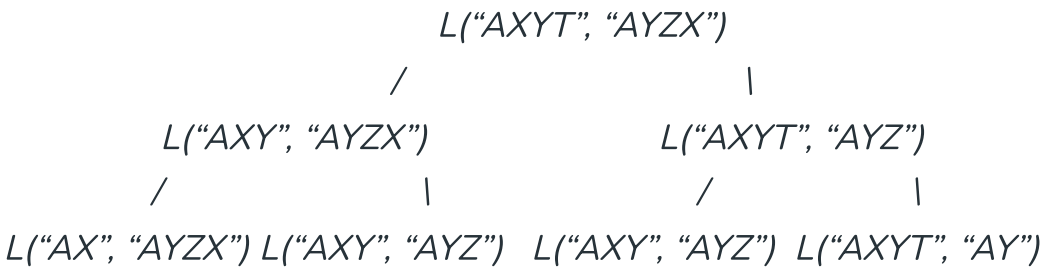
If we notice carefully, we can observe that the above recursive solution holds the following two properties:

1. Optimal Substructure:

See for solving the structure of L(X[0, 1, . . ., m-1], Y[0, 1, . . . , n-1]) we are taking the help of the substructures of X[0, 1, ..., m-2], Y[0, 1,..., n-2], depending on the situation (i.e., using them optimally) to find the solution of the whole.

2. Overlapping Subproblems:

If we use the above recursive approach for strings "**BD**" and "**ABCD**", we will get a partial recursion tree as shown below. Here we can see that the subproblem L("BD", "ABCD") is being calculated more than once. If the total tree is considered there will be several such overlapping subproblems.



Approach: Because of the presence of these two properties we can use Dynamic programming or Memoization to solve the problem. Below is the approach for the solution using recursion.

- *Create a recursive function. Also create a 2D array to store the result of a unique state.*

- *During the recursion call, if the same state is called more than once, then we can directly return the answer stored for that state instead of calculating again.*

Below is the implementation of the above approach:

C++ Java Python C# Javascript

```cpp
// A Top-Down DP implementation
// of LCS problem
#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1],
// Y[0..n-1]
int lcs(char* X, char* Y, int m, int n,
        vector<vector<int> >& dp)
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {
        return dp[m][n];
    }
    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                          lcs(X, Y, m - 1, n, dp));
}

// Driver code
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);
    vector<vector<int> > dp(m + 1, vector<int>(n + 1, -1));
    cout << "Length of LCS is " << lcs(X, Y, m, n, dp);

    return 0;
}
```

Output

```
Length of LCS is 4
```

Time Complexity: O(m * n) where m and n are the string lengths.
Auxiliary Space: O(m * n) Here the recursive stack space is ignored.

Longest Common Subsequence (LCS) using Bottom-Up (Tabulation):

We can use the following steps to implement the dynamic programming approach for LCS.

- Create a 2D array **dp[][]** with rows and columns equal to the length of each input string plus 1 [the number of rows indicates the indices of **S1** and the columns indicate the indices of **S2**].
- Initialize the first row and column of the dp array to 0.
- Iterate through the rows of the dp array, starting from 1 (say using iterator **i**).
  - For each **i**, iterate all the columns from **j = 1 to n**:
    - If **S1[i-1]** is equal to **S2[j-1]**, set the current element of the dp array to the value of the element to (**dp[i-1][j-1] + 1**).
    - Else, set the current element of the dp array to the maximum value of **dp[i-1][j]** and **dp[i][j-1]**.
- After the nested loops, the last element of the dp array will contain the length of the LCS.

See the below illustration for a better understanding:

Illustration:

*Say the strings are **S1 = "AGGTAB"** and **S2 = "GXTXAYB"**.*

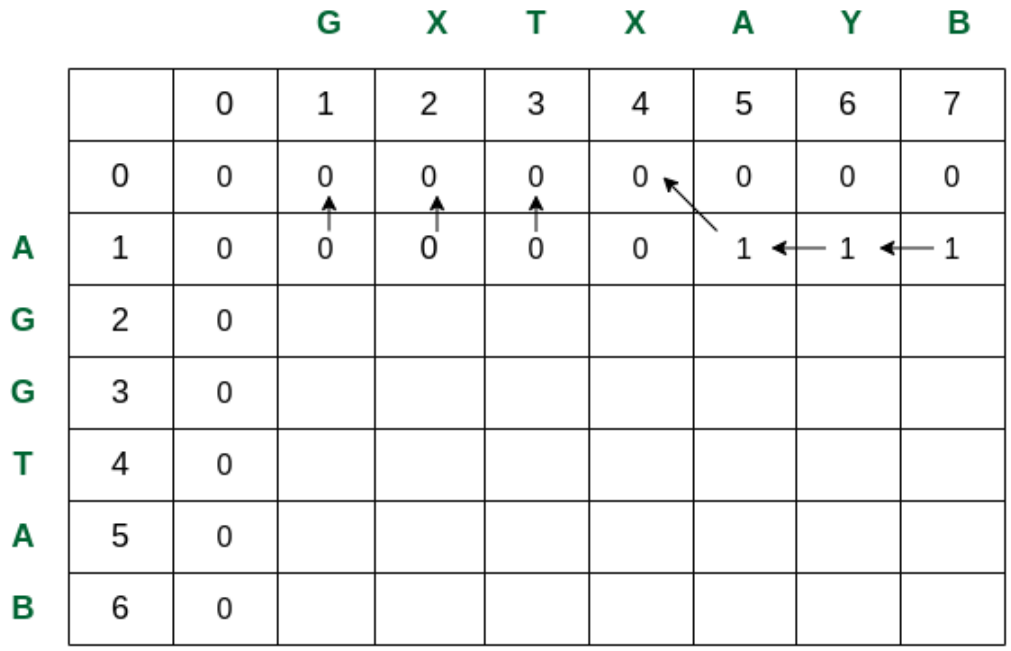
***First step:** Initially create a 2D matrix (say dp[][]) of size 8 x 7 whose first row and first column are filled with 0.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0							
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Creating dp table and filling 0 in 0th row and column

Creating the dp table

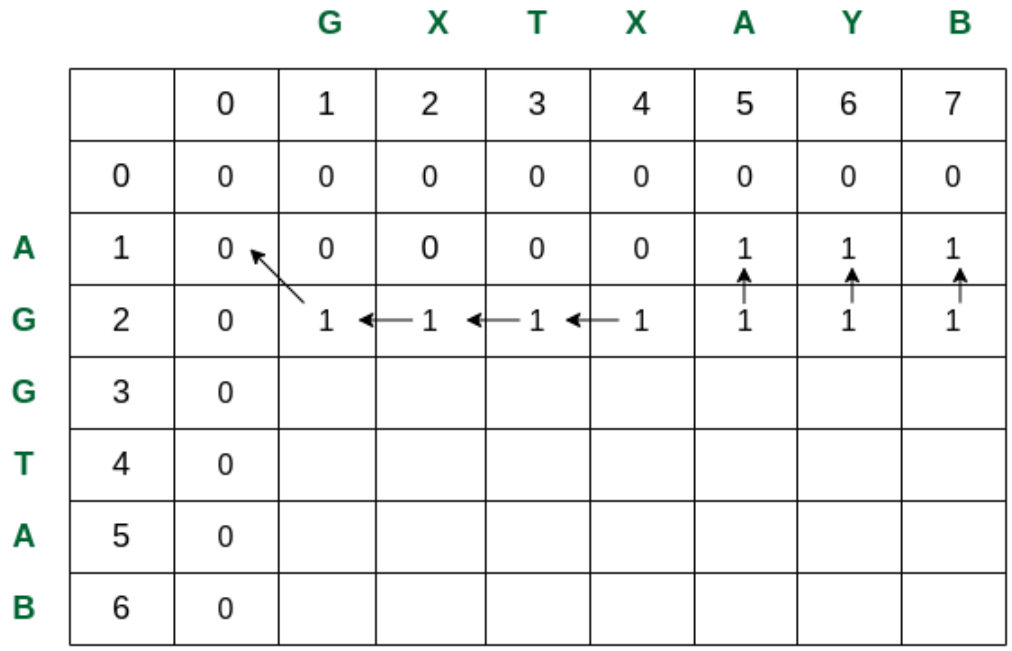
Second step: *Traverse for i = 1. When j becomes 5, S1[0] and S2[4] are equal. So the dp[][] is updated. For the other elements take the maximum of dp[i-1][j] and dp[i][j-1]. (In this case, if both values are equal, we have used arrows to the previous rows).*



Updating dp table for row 1

Filling the row no 1

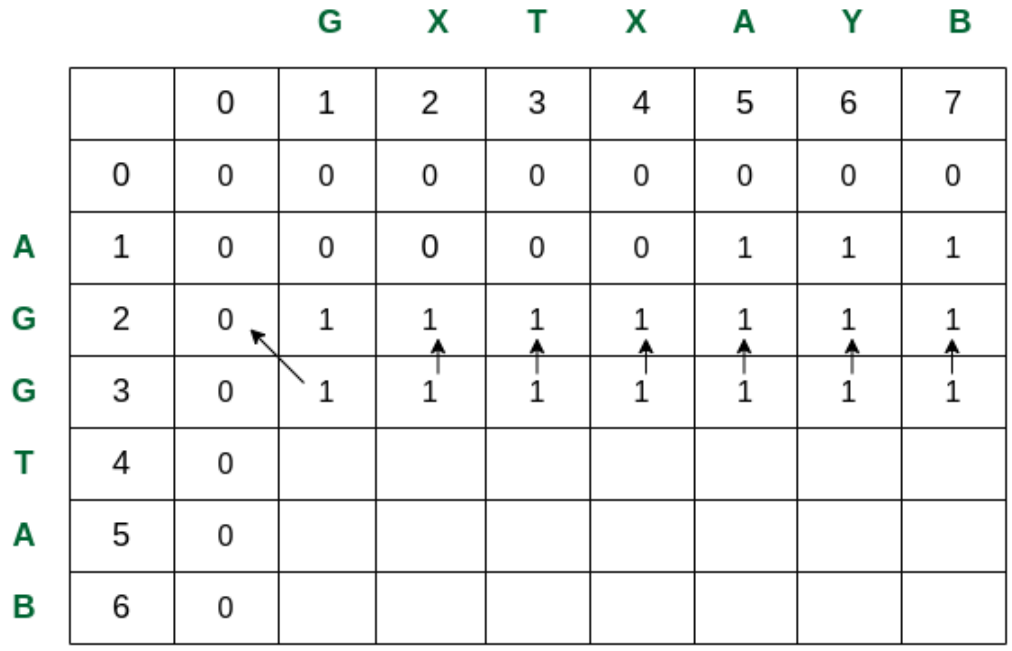
Third step: *While traversed for i = 2, S1[1] and S2[0] are the same (both are 'G'). So the dp value in that cell is updated. Rest of the elements are updated as per the conditions.*



Updating dp table for row 2

Filling the row no. 2

Fourth step: *For i = 3, S1[2] and S2[0] are again same. The updations are as follows.*



Updating dp table for row 3

Filling row no. 3

Fifth step: *For i = 4, we can see that S1[3] and S2[2] are same. So dp[4][3] updated as dp[3][2] + 1 = 2.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0							
B	6	0							

Updating dp table for row 4

Filling row 4

Sixth step: *Here we can see that for i = 5 and j = 5 the values of S1[4] and S2[4] are same (i.e., both are 'A'). So dp[5][5] is updated accordingly and becomes 3.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0							

Updating dp table for row 5

Filling row 5

Final step: *For i = 6, see the last characters of both strings are same (they are 'B'). Therefore the value of dp[6][7] becomes 4.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0	1	1	2	2	3	3	4

Updating dp table for row 6

Filling the final row

*So we get the maximum length of common subsequence as **4**.*

Following is a tabulated implementation for the LCS problem.

C++ Java Python C# Javascript PHP

```cpp
// Dynamic Programming C++ implementation
// of LCS problem
#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1],
// Y[0..n-1]
int lcs(string X, string Y, int m, int n)
{

    // Initializing a matrix of size
    // (m+1)*(n+1)
    int L[m + 1][n + 1];

    // Following steps build L[m+1][n+1]
    // in bottom up fashion. Note that
    // L[i][j] contains length of LCS of
    // X[0..i-1] and Y[0..j-1]
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;

            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    // L[m][n] contains length of LCS
    // for X[0..n-1] and Y[0..m-1]
    return L[m][n];
}

// Driver code
int main()
{
    string S1 = "AGGTAB";
    string S2 = "GXTXAYB";
    int m = S1.size();
    int n = S2.size();

    // Function call
```

```
    cout << "Length of LCS is " << lcs(S1, S2, m, n);

    return 0;
}
```

Output

```
Length of LCS is 4
```

Time Complexity: O(m * n) which is much better than the worst-case time complexity of Naive Recursive implementation.
Auxiliary Space: O(m * n) because the algorithm uses an array of size (m+1)* (n+1) to store the length of the common substrings.

Longest Common Subsequence (LCS) using Bottom-Up (Space-Optimization):

- *In the above tabulation approach we are using L[i-1][j] and L[i][j] etc, here the L[i-1] will refers to the matrix L's previous row and L[i] refers to the current row.*
- *We can do space optimization by using two vectors one is previous and another one is current.*
- *When the inner for loop exits we are initializing previous equal to current.*

Below is the implementation:

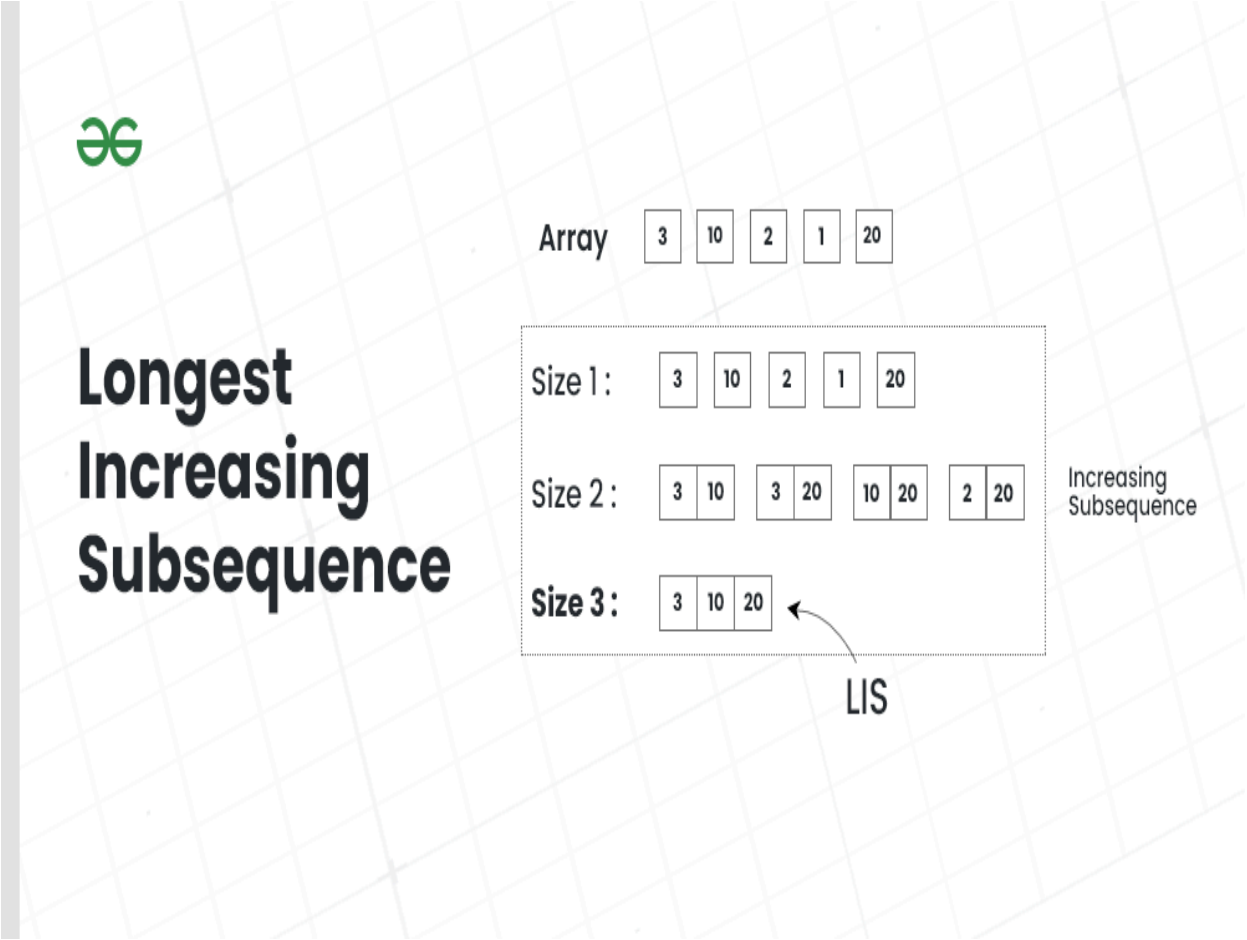
C++ Java Python C# Javascript

```
// Dynamic Programming C++ implementation
// of LCS problem
#include <bits/stdc++.h>
using namespace std;

int longestCommonSubsequence(string& text1, string& text2)
{
    int n = text1.size();
    int m = text2.size();
```

Longest Increasing Subsequence (LIS)

Last Updated : 27 Sep, 2023

Given an array **arr[]** of size **N**, the task is to find the length of the Longest Increasing Subsequence (LIS) i.e., the longest possible subsequence in which the elements of the subsequence are sorted in increasing order.



Longest Increasing Subsequence

Examples:

Input: *arr[] = {3, 10, 2, 1, 20}*
Output: *3*
Explanation: *The longest increasing subsequence is 3, 10, 20*

Input: *arr[] = {3, 2}*
Output:*1*
Explanation: *The longest increasing subsequences are {3} and {2}*

Input: *arr[] = {50, 3, 10, 7, 40, 80}*
Output: *4*
Explanation: *The longest increasing subsequence is {3, 7, 40, 80}*

Longest Increasing Sequence using Recursion:

The problem can be solved based on the following idea:

*Let **L(i)** be the length of the LIS ending at index **i** such that arr[i] is the last element of the LIS. Then, L(i) can be recursively written as:*

- *L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or*
- *L(i) = 1, if no such j exists.*

*Formally, the length of LIS ending at index **i**, is 1 greater than the maximum of lengths of all LIS ending at some index **j** such that **arr[j] < arr[i]** where **j < i**.*

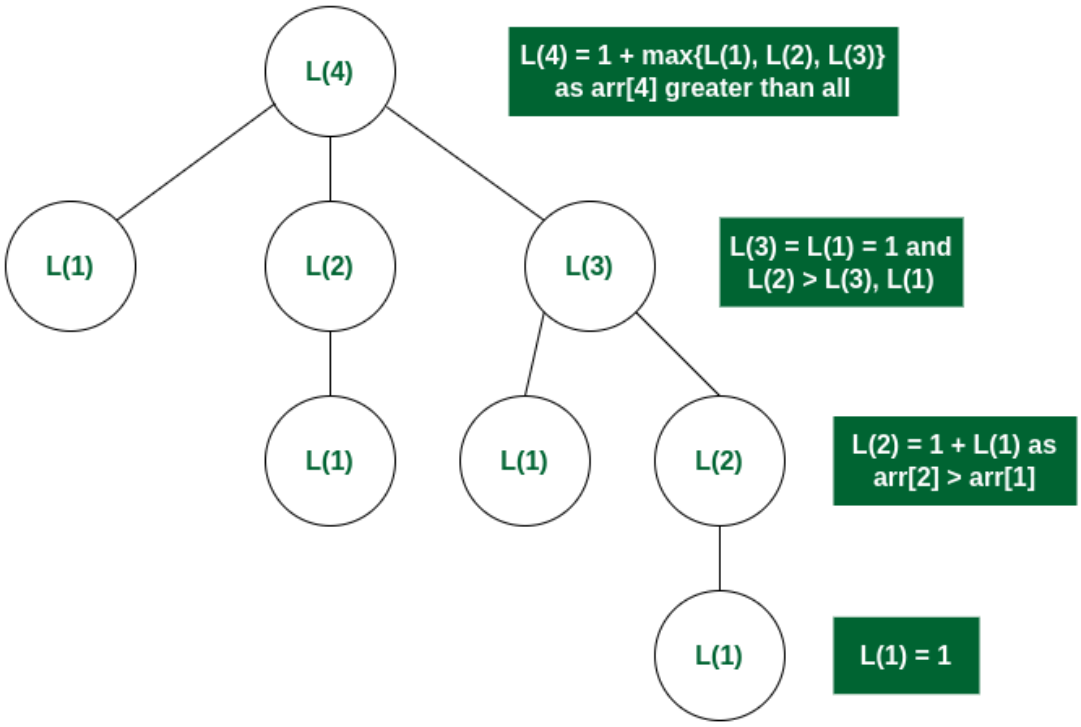
We can see that the above recurrence relation follows the **optimal substructure** property.

Illustration:

Follow the below illustration for a better understanding:

Consider arr[] = {3, 10, 2, 11}

L(i): Denotes LIS of subarray ending at position 'i'



Recursion Tree

Follow the steps mentioned below to implement the above idea:

- Create a recursive function.

- For each recursive call, Iterate from the **i = 1** to the current position and do the following:
  - Find the possible length of the longest increasing subsequence ending at the current position if the previous sequence ended at **i**.
  - Update the maximum possible length accordingly.
- Repeat this for all indices and find the answer

Below is the implementation of the recursive approach:

C++

```
// A Naive C++ recursive implementation
// of LIS problem
#include <bits/stdc++.h>
using namespace std;

// To make use of recursive calls, this
// function must return two things:
// 1) Length of LIS ending with element
// arr[n-1].
// We use max_ending_here for this purpose
// 2) Overall maximum as the LIS may end
// with an element before arr[n-1] max_ref
// is used this purpose.
// The value of LIS of full array of size
// n is stored in *max_ref which is
// our final result
int _lis(int arr[], int n, int* max_ref)
{

    // Base case
    if (n == 1)
        return 1;

    // 'max_ending_here' is length of
    // LIS ending with arr[n-1]
    int res, max_ending_here = 1;

    // Recursively get all LIS ending with
    // arr[0], arr[1] ... arr[n-2]. If
    // arr[i-1] is smaller than arr[n-1],
    // and max ending with arr[n-1] needs
    // to be updated, then update it
    for (int i = 1; i < n; i++) {
        res = _lis(arr, i, max_ref);
```

```
            if (arr[i - 1] < arr[n - 1]
                && res + 1 > max_ending_here)
                max_ending_here = res + 1;
        }

        // Compare max_ending_here with the
        // overall max. And update the
        // overall max if needed
        if (*max_ref < max_ending_here)
            *max_ref = max_ending_here;

        // Return length of LIS ending
        // with arr[n-1]
        return max_ending_here;
    }

    // The wrapper function for _lis()
    int lis(int arr[], int n)
    {

        // The max variable holds the result
        int max = 1;

        // The function _lis() stores its
        // result in max
        _lis(arr, n, &max);

        // Returns max
        return max;
    }

    // Driver program to test above function
    int main()
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = sizeof(arr) / sizeof(arr[0]);

        // Function call
        cout << "Length of lis is " << lis(arr, n);
        return 0;
    }
```

C

```
// A Naive C recursive implementation
// of LIS problem
#include <stdio.h>
#include <stdlib.h>
```

```
// To make use of recursive calls, this
// function must return two things:
// 1) Length of LIS ending with element arr[n-1].
// We use max_ending_here for this purpose
// 2) Overall maximum as the LIS may end with
// an element before arr[n-1] max_ref is
//  used this purpose.
// The value of LIS of full array of size n
// is stored in *max_ref which is our final result
int _lis(int arr[], int n, int* max_ref)
{
    // Base case
    if (n == 1)
        return 1;

    // 'max_ending_here' is length of LIS
    // ending with arr[n-1]
    int res, max_ending_here = 1;

    // Recursively get all LIS ending with arr[0],
    // arr[1] ... arr[n-2]. If arr[i-1] is smaller
    // than arr[n-1], and max ending with arr[n-1]
    // needs to be updated, then update it
    for (int i = 1; i < n; i++) {
        res = _lis(arr, i, max_ref);
        if (arr[i - 1] < arr[n - 1]
            && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall
    // max. And update the overall max if needed
    if (*max_ref < max_ending_here)
        *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis(arr, n, &max);
```

```
    // returns max
    return max;
}

// Driver program to test above function
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    printf("Length of lis is %d", lis(arr, n));
    return 0;
}
```

Java

```
// A Naive Java Program for LIS Implementation
import java.io.*;
import java.util.*;

class LIS {

    // Stores the LIS
    static int max_ref;

    // To make use of recursive calls, this function must
    // return two things: 1) Length of LIS ending with
    // element arr[n-1]. We use max_ending_here for this
    // purpose 2) Overall maximum as the LIS may end with an
    // element before arr[n-1] max_ref is used this purpose.
    // The value of LIS of full array of size n is stored in
    // *max_ref which is our final result
    static int _lis(int arr[], int n)
    {
        // Base case
        if (n == 1)
            return 1;

        // 'max_ending_here' is length of LIS ending with
        // arr[n-1]
        int res, max_ending_here = 1;

        // Recursively get all LIS ending with arr[0],
        // arr[1] ... arr[n-2]. If   arr[i-1] is smaller
        // than arr[n-1], and max ending with arr[n-1] needs
        // to be updated, then update it
        for (int i = 1; i < n; i++) {
```

```
            res = _lis(arr, i);
            if (arr[i - 1] < arr[n - 1]
                && res + 1 > max_ending_here)
                max_ending_here = res + 1;
        }

        // Compare max_ending_here with the overall max. And
        // update the overall max if needed
        if (max_ref < max_ending_here)
            max_ref = max_ending_here;

        // Return length of LIS ending with arr[n-1]
        return max_ending_here;
    }

    // The wrapper function for _lis()
    static int lis(int arr[], int n)
    {
        // The max variable holds the result
        max_ref = 1;

        // The function _lis() stores its result in max
        _lis(arr, n);

        // Returns max
        return max_ref;
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.length;

        // Function call
        System.out.println("Length of lis is "
                           + lis(arr, n));
    }
}
// This code is contributed by Rajat Mishra
```

DSA Interview Problems on DP Practice DP MCQs on DP Tutorial on Dynamic Programming Optimal Substru

Python3

```
# A naive Python implementation of LIS problem


# Global variable to store the maximum
global maximum
```

```
# To make use of recursive calls, this function must return
# two things:
# 1) Length of LIS ending with element arr[n-1]. We use
# max_ending_here for this purpose
# 2) Overall maximum as the LIS may end with an element
# before arr[n-1] max_ref is used this purpose.
# The value of LIS of full array of size n is stored in
# *max_ref which is our final result
def _lis(arr, n):

    # To allow the access of global variable
    global maximum

    # Base Case
    if n == 1:
        return 1

    # maxEndingHere is the length of LIS ending with arr[n-1]
    maxEndingHere = 1

    # Recursively get all LIS ending with
    # arr[0], arr[1]..arr[n-2]
    # If arr[i-1] is smaller than arr[n-1], and
    # max ending with arr[n-1] needs to be updated,
    # then update it
    for i in range(1, n):
        res = _lis(arr, i)
        if arr[i-1] < arr[n-1] and res+1 > maxEndingHere:
            maxEndingHere = res + 1

    # Compare maxEndingHere with overall maximum. And
    # update the overall maximum if needed
    maximum = max(maximum, maxEndingHere)

    return maxEndingHere


def lis(arr):

    # To allow the access of global variable
    global maximum

    # Length of arr
    n = len(arr)

    # Maximum variable holds the result
    maximum = 1
```

```
    # The function _lis() stores its result in maximum
    _lis(arr, n)
    return maximum


# Driver program to test the above function
if __name__ == '__main__':
    arr = [10, 22, 9, 33, 21, 50, 41, 60]
    n = len(arr)

    # Function call
    print("Length of lis is", lis(arr))

# This code is contributed by NIKHIL KUMAR SINGH
```

C#

```
using System;

// A Naive C# Program for LIS Implementation
class LIS {

    // Stores the LIS
    static int max_ref;

    // To make use of recursive calls, this function must
    // return two things: 1) Length of LIS ending with
    // element arr[n-1]. We use max_ending_here for this
    // purpose 2) Overall maximum as the LIS may end with an
    // element before arr[n-1] max_ref is used this purpose.
    // The value of LIS of full array of size n is stored in
    // *max_ref which is our final result
    static int _lis(int[] arr, int n)
    {
        // Base case
        if (n == 1)
            return 1;

        // 'max_ending_here' is length of LIS ending with
        // arr[n-1]
        int res, max_ending_here = 1;

        // Recursively get all LIS ending with arr[0],
        // arr[1] ... arr[n-2]. If   arr[i-1] is smaller
        // than arr[n-1], and max ending with arr[n-1] needs
        // to be updated, then update it
        for (int i = 1; i < n; i++) {
```

```
            res = _lis(arr, i);
            if (arr[i - 1] < arr[n - 1]
                && res + 1 > max_ending_here)
                max_ending_here = res + 1;
        }

        // Compare max_ending_here with the overall max
        // and update the overall max if needed
        if (max_ref < max_ending_here)
            max_ref = max_ending_here;

        // Return length of LIS ending with arr[n-1]
        return max_ending_here;
    }

    // The wrapper function for _lis()
    static int lis(int[] arr, int n)
    {
        // The max variable holds the result
        max_ref = 1;

        // The function _lis() stores its result in max
        _lis(arr, n);

        // Returns max
        return max_ref;
    }

    // Driver program to test above functions
    public static void Main()
    {
        int[] arr = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.Length;

        // Function call
        Console.Write("Length of lis is " + lis(arr, n)
                      + "\n");
    }
}
```

Javascript

```
/* A Naive javascript Program for LIS Implementation */


    let  max_ref; // stores the LIS
    /* To make use of recursive calls, this function must return
```

```
two things:
1) Length of LIS ending with element arr[n-1]. We use
   max_ending_here for this purpose
2) Overall maximum as the LIS may end with an element
   before arr[n-1] max_ref is used this purpose.
The value of LIS of full array of size n is stored in
*max_ref which is our final result */
 function  _lis(arr,n)
 {
     // base case
     if (n == 1)
         return 1;

     // 'max_ending_here' is length of LIS ending with arr[n-1]
     let res, max_ending_here = 1;
     /* Recursively get all LIS ending with arr[0], arr[1] ...
        arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and
        max ending with arr[n-1] needs to be updated, then
        update it */
     for (let i = 1; i < n; i++)
     {
         res = _lis(arr, i);
         if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
             max_ending_here = res + 1;
     }
     // Compare max_ending_here with the overall max. And
     // update the overall max if needed
     if (max_ref < max_ending_here)
         max_ref = max_ending_here;

     // Return length of LIS ending with arr[n-1]
     return max_ending_here;
 }

 // The wrapper function for _lis()
 function  lis(arr,n)
 {
     // The max variable holds the result
     max_ref = 1;

     // The function _lis() stores its result in max
     _lis( arr, n);

     // returns max
     return max_ref;
 }

 // driver program to test above functions
 let arr=[10, 22, 9, 33, 21, 50, 41, 60 ]
```

```
    let n = arr.length;
    document.write("Length of lis is "
                           + lis(arr, n) + "<br>");

    // This code is contributed by avanitrachhadiya2155
```

Output

```
Length of lis is 5
```

Time Complexity: $O(2^n)$ The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems as explained in the recursive tree diagram above.
Auxiliary Space: O(1). No external space is used for storing values apart from the internal stack space.

Longest Increasing Subsequence using Memoization:

*If noticed carefully, we can see that the above recursive solution also follows the **overlapping subproblems** property i.e., same substructure solved again and again in different recursion call paths. We can avoid this using the memoization approach.*

We can see that each state can be uniquely identified using two parameters:

- ***Current index** (denotes the last index of the LIS) and*
- ***Previous index** (denotes the ending index of the previous LIS behind which the **arr[i]** is being concatenated).*

Below is the implementation of the above approach.

C++

```
// C++ code of memoization approach for LIS
#include <bits/stdc++.h>
```

```
using namespace std;

// To make use of recursive calls, this
// function must return two things:
// 1) Length of LIS ending with element
// arr[n-1].
// We use max_ending_here for this purpose
// Overall maximum as the LIS may end with
// an element before arr[n-1] max_ref is
// used this purpose.
// The value of LIS of full array of size
// n is stored in *max_ref which is
// our final result
int f(int idx, int prev_idx, int n, int a[],
      vector<vector<int> >& dp)
{
    if (idx == n) {
        return 0;
    }

    if (dp[idx][prev_idx + 1] != -1) {
        return dp[idx][prev_idx + 1];
    }

    int notTake = 0 + f(idx + 1, prev_idx, n, a, dp);
    int take = INT_MIN;
    if (prev_idx == -1 || a[idx] > a[prev_idx]) {
        take = 1 + f(idx + 1, idx, n, a, dp);
    }

    return dp[idx][prev_idx + 1] = max(take, notTake);
}

// Function to find length of
// longest increasing subsequence
int longestSubsequence(int n, int a[])
{
    vector<vector<int> > dp(n + 1, vector<int>(n + 1, -1));
    return f(0, -1, n, a, dp);
}

// Driver program to test above function
int main()
{
    int a[] = { 3, 10, 2, 1, 20 };
    int n = sizeof(a) / sizeof(a[0]);

    // Function call
    cout << "Length of lis is " << longestSubsequence(n, a);
```

```
    return 0;
}
```

Java

```
// A Memoization Java Program for LIS Implementation

import java.lang.*;
import java.util.Arrays;

class LIS {

    // To make use of recursive calls, this function must
    // return two things: 1) Length of LIS ending with
    // element arr[n-1]. We use max_ending_here for this
    // purpose 2) Overall maximum as the LIS may end with an
    // element before arr[n-1] max_ref is used this purpose.
    // The value of LIS of full array of size n is stored in
    // *max_ref which is our final result
    static int f(int idx, int prev_idx, int n, int a[],
                 int[][] dp)
    {
        if (idx == n) {
            return 0;
        }

        if (dp[idx][prev_idx + 1] != -1) {
            return dp[idx][prev_idx + 1];
        }

        int notTake = 0 + f(idx + 1, prev_idx, n, a, dp);
        int take = Integer.MIN_VALUE;
        if (prev_idx == -1 || a[idx] > a[prev_idx]) {
            take = 1 + f(idx + 1, idx, n, a, dp);
        }

        return dp[idx][prev_idx + 1]
            = Math.max(take, notTake);
    }

    // The wrapper function for _lis()
    static int lis(int arr[], int n)
    {
        // The function _lis() stores its result in max
        int dp[][] = new int[n + 1][n + 1];
        for (int row[] : dp)
            Arrays.fill(row, -1);
```

```
        return f(0, -1, n, arr, dp);
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        int a[] = { 3, 10, 2, 1, 20 };
        int n = a.length;

        // Function call
        System.out.println("Length of lis is " + lis(a, n));
    }
}

// This code is contributed by Sanskar.
```

Python3

```
# A Naive Python recursive implementation
# of LIS problem


import sys

# To make use of recursive calls, this
# function must return two things:
# 1) Length of LIS ending with element arr[n-1].
#     We use max_ending_here for this purpose
# 2) Overall maximum as the LIS may end with
#     an element before arr[n-1] max_ref is
#     used this purpose.
# The value of LIS of full array of size n
# is stored in *max_ref which is our final result


def f(idx, prev_idx, n, a, dp):

    if (idx == n):
        return 0

    if (dp[idx][prev_idx + 1] != -1):
        return dp[idx][prev_idx + 1]

    notTake = 0 + f(idx + 1, prev_idx, n, a, dp)
    take = -sys.maxsize - 1
    if (prev_idx == -1 or a[idx] > a[prev_idx]):
        take = 1 + f(idx + 1, idx, n, a, dp)
```

```
    dp[idx][prev_idx + 1] = max(take, notTake)
    return dp[idx][prev_idx + 1]

# Function to find length of longest increasing
# subsequence.


def longestSubsequence(n, a):

    dp = [[-1 for i in range(n + 1)]for j in range(n + 1)]
    return f(0, -1, n, a, dp)


# Driver program to test above function
if __name__ == '__main__':
    a = [3, 10, 2, 1, 20]
    n = len(a)

    # Function call
    print("Length of lis is", longestSubsequence(n, a))

# This code is contributed by shinjanpatra
```

C#

```
// C# approach to implementation the memoization approach

using System;

class GFG {

    // To make use of recursive calls, this
    // function must return two things:
    // 1) Length of LIS ending with element arr[n-1].
    // We use max_ending_here for this purpose
    // 2) Overall maximum as the LIS may end with
    // an element before arr[n-1] max_ref is
    //  used this purpose.
    // The value of LIS of full array of size n
    // is stored in *max_ref which is our final result
    public static int INT_MIN = -2147483648;

    public static int f(int idx, int prev_idx, int n,
                        int[] a, int[, ] dp)
    {
        if (idx == n) {
            return 0;
        }
```

```
        if (dp[idx, prev_idx + 1] != -1) {
            return dp[idx, prev_idx + 1];
        }
        int notTake = 0 + f(idx + 1, prev_idx, n, a, dp);
        int take = INT_MIN;
        if (prev_idx == -1 || a[idx] > a[prev_idx]) {
            take = 1 + f(idx + 1, idx, n, a, dp);
        }

        return dp[idx, prev_idx + 1]
            = Math.Max(take, notTake);
    }

    // Function to find length of longest increasing
    // subsequence.
    public static int longestSubsequence(int n, int[] a)
    {
        int[, ] dp = new int[n + 1, n + 1];

        for (int i = 0; i < n + 1; i++) {
            for (int j = 0; j < n + 1; j++) {
                dp[i, j] = -1;
            }
        }
        return f(0, -1, n, a, dp);
    }

    // Driver code
    static void Main()
    {
        int[] a = { 3, 10, 2, 1, 20 };
        int n = a.Length;
        Console.WriteLine("Length of lis is "
                          + longestSubsequence(n, a));
    }
}

// The code is contributed by Nidhi goel.
```

Javascript

```
/* A Naive Javascript recursive implementation
      of LIS problem */

      /* To make use of recursive calls, this
      function must return two things:
      1) Length of LIS ending with element arr[n-1].
          We use max_ending_here for this purpose
```

```
2) Overall maximum as the LIS may end with
    an element before arr[n-1] max_ref is
    used this purpose.
The value of LIS of full array of size n
is stored in *max_ref which is our final result
*/

function f(idx, prev_idx, n, a, dp) {
  if (idx == n) {
    return 0;
  }

  if (dp[idx][prev_idx + 1] != -1) {
    return dp[idx][prev_idx + 1];
  }

  var notTake = 0 + f(idx + 1, prev_idx, n, a, dp);
  var take = Number.MIN_VALUE;
  if (prev_idx == -1 || a[idx] > a[prev_idx]) {
    take = 1 + f(idx + 1, idx, n, a, dp);
  }

  return (dp[idx][prev_idx + 1] = Math.max(take, notTake));
}
// Function to find length of longest increasing
// subsequence.
function longestSubsequence(n, a) {
  var dp = Array(n + 1)
    .fill()
    .map(() => Array(n + 1).fill(-1));
  return f(0, -1, n, a, dp);
}

/* Driver program to test above function */

var a = [3, 10, 2, 1, 20];
var n = 5;
console.log("Length of lis is " + longestSubsequence(n, a));

// This code is contributed by satwiksuman.
```

Output

```
Length of lis is 3
```

Time Complexity: $O(N^2)$
Auxiliary Space: $O(N^2)$

Longest Increasing Subsequence using Dynamic Programming:

Due to optimal substructure and overlapping subproblem property, we can also utilise Dynamic programming to solve the problem. Instead of memoization, we can use the nested loop to implement the recursive relation.

*The outer loop will run from **i = 1 to N** and the inner loop will run from **j = 0 to i** and use the recurrence relation to solve the problem.*

Below is the implementation of the above approach:

C++

```
// Dynamic Programming C++ implementation
// of LIS problem
#include <bits/stdc++.h>
using namespace std;

// lis() returns the length of the longest
// increasing subsequence in arr[] of size n
int lis(int arr[], int n)
{
    int lis[n];

    lis[0] = 1;

    // Compute optimized LIS values in
    // bottom up manner
    for (int i = 1; i < n; i++) {
        lis[i] = 1;
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
```

```
    return *max_element(lis, lis + n);
}

// Driver program to test above function
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    printf("Length of lis is %d\n", lis(arr, n));
    return 0;
}
```

Java

```
// Dynamic Programming Java implementation
// of LIS problem

import java.lang.*;

class LIS {

    // lis() returns the length of the longest
    // increasing subsequence in arr[] of size n
    static int lis(int arr[], int n)
    {
        int lis[] = new int[n];
        int i, j, max = 0;

        // Initialize LIS values for all indexes
        for (i = 0; i < n; i++)
            lis[i] = 1;

        // Compute optimized LIS values in
        // bottom up manner
        for (i = 1; i < n; i++)
            for (j = 0; j < i; j++)
                if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;

        // Pick maximum of all LIS values
        for (i = 0; i < n; i++)
            if (max < lis[i])
                max = lis[i];

        return max;
    }
```

Matrix Chain Multiplication | DP-8

Last Updated : 20 Dec, 2022

Given the dimension of a sequence of matrices in an array **arr[]**, where the dimension of the **i^{th}** matrix is **(arr[i-1] * arr[i])**, the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

Examples:

Input: *arr[] = {40, 20, 30, 10, 30}*
Output: *26000*
Explanation:*There are 4 matrices of dimensions 40×20, 20×30, 30×10, 10×30.*
Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by
putting parenthesis in following way (A(BC))D.
*The minimum is 20*30*10 + 40*20*10 + 40*10*30*

Input: *arr[] = {1, 2, 3, 4, 3}*
Output: *30*
Explanation: *There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3.*
Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by
putting parenthesis in following way ((AB)C)D.
*The minimum number is 1*2*3 + 1*3*4 + 1*4*3 = 30*

Input: *arr[] = {10, 20, 30}*
Output: *6000*
Explanation: *There are only two matrices of dimensions 10×20 and 20×30.*
*So there is only one way to multiply the matrices, cost of which is 10*20*30*

Recommended Problem

Matrix Chain Multiplication

Dynamic Programming Matrix +2 more Flipkart Microsoft +1 more

Solve Problem

Submission count: 1.1L

Matrix Chain Multiplication using Recursion:

We can solve the problem using recursion based on the following facts and observations:

*Two matrices of size **m*n** and **n*p** when multiplied, they generate a matrix of size **m*p** and the number of multiplications performed are **m*n*p**.*

*Now, for a given chain of **N** matrices, the first partition can be done in **N-1** ways. For example, sequence of matrices A, B, C and D can be grouped*

as (A)(BCD), (AB)(CD) or (ABC)(D) in these 3 ways.

*So a range **[i, j]** can be broken into two groups like **{[i, i+1], [i+1, j]}, {[i, i+2], [i+2, j]}, . . . , {[i, j-1], [j-1, j]}**.*

- *Each of the groups can be further partitioned into smaller groups and we can find the total required multiplications by solving for each of the groups.*
- *The minimum number of multiplications among all the first partitions is the required answer.*

Follow the steps mentioned below to implement the approach:

- Create a recursive function that takes **i** and **j** as parameters that determines the range of a group.
  - Iterate from **k = i to j** to partition the given range into two groups.
  - Call the recursive function for these groups.
  - Return the minimum value among all the partitions as the required minimum number of multiplications to multiply all the matrices of this group.
- The minimum value returned for the range **0** to **N-1** is the required answer.

Below is the implementation of the above approach.

C++

```
// C++ code to implement the
// matrix chain multiplication using recursion

#include <bits/stdc++.h>
using namespace std;

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1 . . . n
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int mini = INT_MAX;
```

```
    int count;

    // Place parenthesis at different places
    // between first and last matrix,
    // recursively calculate count of multiplications
    // for each parenthesis placement
    // and return the minimum count
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k)
                + MatrixChainOrder(p, k + 1, j)
                + p[i - 1] * p[k] * p[j];

        mini = min(count, mini);
    }

    // Return minimum count
    return mini;
}

// Driver Code
int main()
{
    int arr[] = { 1, 2, 3, 4, 3 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    cout << "Minimum number of multiplications is "
         << MatrixChainOrder(arr, 1, N - 1);
    return 0;
}

// This code is contributed by Shivi_Aggarwal
```

C

```
// C code to implement the
// matrix chain multiplication using recursion

#include <limits.h>
#include <stdio.h>

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1 . . . n
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
```

```
    // Place parenthesis at different places
    // between first and last matrix,
    // recursively calculate count of multiplications
    // for each parenthesis placement
    // and return the minimum count
    var k=0;
    for (k = i; k < j; k++)
    {
        var count = MatrixChainOrder(p, i, k)
                    + MatrixChainOrder(p, k + 1, j)
                    + p[i - 1] * p[k] * p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver code
var arr = [ 1, 2, 3, 4, 3 ];
var N = arr.length;

document.write(
    "Minimum number of multiplications is "
    + MatrixChainOrder(arr, 1, N - 1));

// This code contributed by shikhasingrajput
```

Output

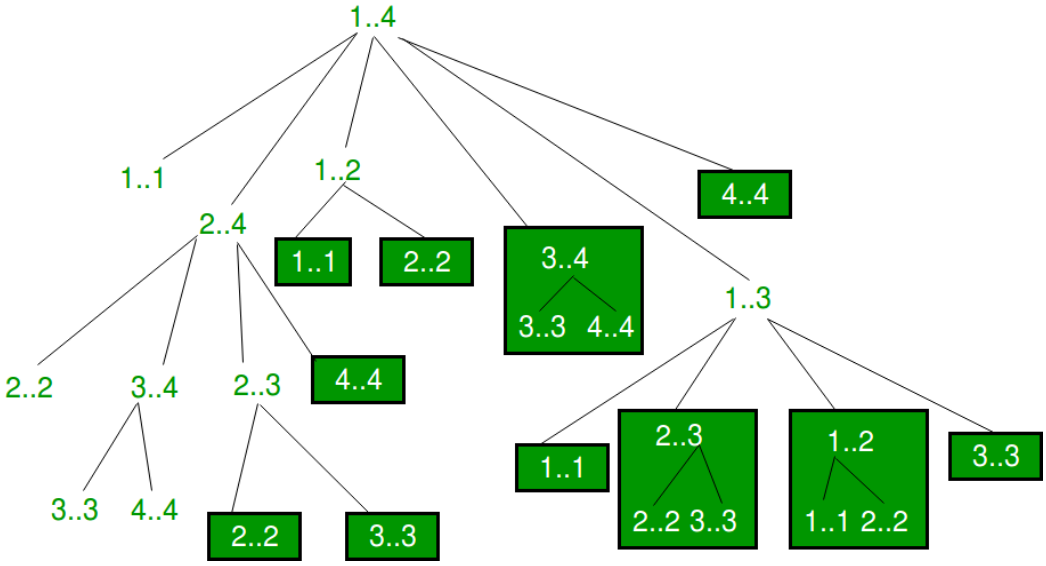
```
Minimum number of multiplications is 30
```

The time complexity of the solution is exponential

Auxiliary Space: O(1)

Dynamic Programming Solution for Matrix Chain Multiplication using Memoization:

Below is the recursion tree for the 2nd example of the above recursive approach:



If observed carefully you can find the following two properties:

> ***1) Optimal Substructure:*** *In the above case, we are breaking the bigger groups into smaller subgroups and solving them to finally find the minimum number of multiplications. Therefore, it can be said that the problem has optimal substructure property.*
>
> ***2) Overlapping Subproblems:*** *We can see in the recursion tree that the same subproblems are called again and again and this problem has the Overlapping Subproblems property.*
>
> *So Matrix Chain Multiplication problem has both properties of a dynamic programming problem. So recomputations of same subproblems can be avoided by constructing a temporary array **dp[][]** in a bottom up manner.*

Follow the below steps to solve the problem:

- Build a matrix **dp[][]** of size **N*N** for memoization purposes.
- Use the same recursive call as done in the above approach:
  - When we find a range **(i, j)** for which the value is already calculated, return the minimum value for that range (i.e., **dp[i][j]**).
  - Otherwise, perform the recursive calls as mentioned earlier.
- The value stored at **dp[0][N-1]** is the required answer.

Below is the implementation of the above approach

C++

```
// C++ program using memoization
#include <bits/stdc++.h>
using namespace std;
int dp[100][100];

// Function for matrix chain multiplication
int matrixChainMemoised(int* p, int i, int j)
{
    if (i == j)
    {
        return 0;
    }
    if (dp[i][j] != -1)
    {
        return dp[i][j];
    }
    dp[i][j] = INT_MAX;
    for (int k = i; k < j; k++)
    {
        dp[i][j] = min(
            dp[i][j], matrixChainMemoised(p, i, k)
                    + matrixChainMemoised(p, k + 1, j)
                      + p[i - 1] * p[k] * p[j]);
    }
    return dp[i][j];
}
int MatrixChainOrder(int* p, int n)
{
    int i = 1, j = n - 1;
    return matrixChainMemoised(p, i, j);
}

// Driver Code
int main()
{
    int arr[] = { 1, 2, 3, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    memset(dp, -1, sizeof dp);

    cout << "Minimum number of multiplications is "
         << MatrixChainOrder(arr, n);
}

// This code is contributed by Sumit_Yadav
```

```
{
    for(var j = 0; j < dp.length; j++)
    {
        dp[i][j] = -1;
    }
}

document.write("Minimum number of multiplications is " +
               MatrixChainOrder(arr, n));

// This code is contributed by target_2
```

Output

```
Minimum number of multiplications is 18
```

Time Complexity: $O(N^3)$
Auxiliary Space: $O(N^2)$ ignoring recursion stack space

Dynamic Programming Solution for Matrix Chain Multiplication using Tabulation (Iterative Approach):

> *In iterative approach, we initially need to find the number of multiplications required to multiply two adjacent matrices. We can use these values to find the minimum multiplication required for matrices in a range of length 3 and further use those values for ranges with higher lengths.*
>
> *Build on the answer in this manner till the range becomes [0, N-1].*

Follow the steps mentioned below to implement the idea:

- Iterate from **l = 2 to N-1** which denotes the length of the range:
  - Iterate from **i = 0 to N-1**:
    - Find the right end of the range (**j**) having **l** matrices.
    - Iterate from **k = i+1 to j** which denotes the point of partition.

- Multiply the matrices in range (**i, k**) and (**k, j**).
- This will create two matrices with dimensions **arr[i-1]*arr[k]** and **arr[k]*arr[j]**.
- The number of multiplications to be performed to multiply these two matrices (say **X**) are **arr[i-1]*arr[k]*arr[j]**.
- The total number of multiplications is **dp[i][k]+ dp[k+1][j] + X**.

- The value stored at **dp[1][N-1]** is the required answer.

Below is the implementation of the above approach.

C++

```
// See the Cormen book for details of the
// following algorithm
#include <bits/stdc++.h>
using namespace std;

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one
    extra row and one extra column are
    allocated in m[][]. 0th row and 0th
    column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i, j] = Minimum number of scalar
    multiplications needed to compute the
    matrix A[i]A[i+1]...A[j] = A[i..j] where
    dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying
    // one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
```

```
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j]
                    + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

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

// Driver Code
int main()
{
    int arr[] = { 1, 2, 3, 4 };
    int size = sizeof(arr) / sizeof(arr[0]);

    cout << "Minimum number of multiplications is "
         << MatrixChainOrder(arr, size);

    getchar();
    return 0;
}

// This code is contributed
// by Akanksha Rai
```

C

```
// See the Cormen book for details of the following
// algorithm
#include <limits.h>
#include <stdio.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program,
       one extra row and one
       extra column are allocated in m[][].
       0th row and 0th
       column of m[][] are not used */
```

```
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j]
                    + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

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

// Driver code
var arr = [ 1, 2, 3, 4 ];
var size = arr.length;

document.write(
    "Minimum number of multiplications is "
    + MatrixChainOrder(arr, size));

// This code contributed by Princi Singh
```

Output

```
 Minimum number of multiplications is 18
```

Time Complexity: $O(N^3)$

Auxiliary Space: $O(N^2)$

Matrix Chain Multiplication (A O(N^2) Solution)

Printing brackets in Matrix Chain Multiplication Problem

Applications:

Minimum and Maximum values of an expression with * and +

Maximum Product Cutting | DP-36

Last Updated : 19 Apr, 2024

Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Examples:

```
Input: n = 2
Output: 1 (Maximum obtainable product is 1*1)

Input: n = 3
Output: 2 (Maximum obtainable product is 1*2)

Input: n = 4
Output: 4 (Maximum obtainable product is 2*2)

Input: n = 5
Output: 6 (Maximum obtainable product is 2*3)

Input: n = 10
Output: 36 (Maximum obtainable product is 3*3*4)
```

1) Optimal Substructure:

This problem is similar to Rod Cutting Problem. We can get the maximum product by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.
Let maxProd(n) be the maximum product for a rope of length n. maxProd(n) can be written as following.
maxProd(n) = max(i*(n-i), maxProdRec(n-i)*i) for all i in {1, 2, 3 .. n}

2) Overlapping Subproblems:

Following is simple recursive implementation of the problem. The implementation simply follows the recursive structure mentioned above.

C++ Java Python3 C# Javascript PHP

```cpp
// A Naive Recursive method to find maximum product
#include <iostream>
using namespace std;

// Utility function to get the maximum of two and three integers
int max(int a, int b) { return (a > b)? a : b;}
int max(int a, int b, int c) { return max(a, max(b, c));}

// The main function that returns maximum product obtainable
// from a rope of length n
int maxProd(int n)
{
    // Base cases
    if (n == 0 || n == 1) return 0;

    // Make a cut at different places and take the maximum of all
    int max_val = 0;
    for (int i = 1; i < n; i++)
      max_val = max(max_val, i*(n-i), maxProd(n-i)*i);

    // Return the maximum of all values
    return max_val;
}

/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output

```
Maximum Product is 36
```

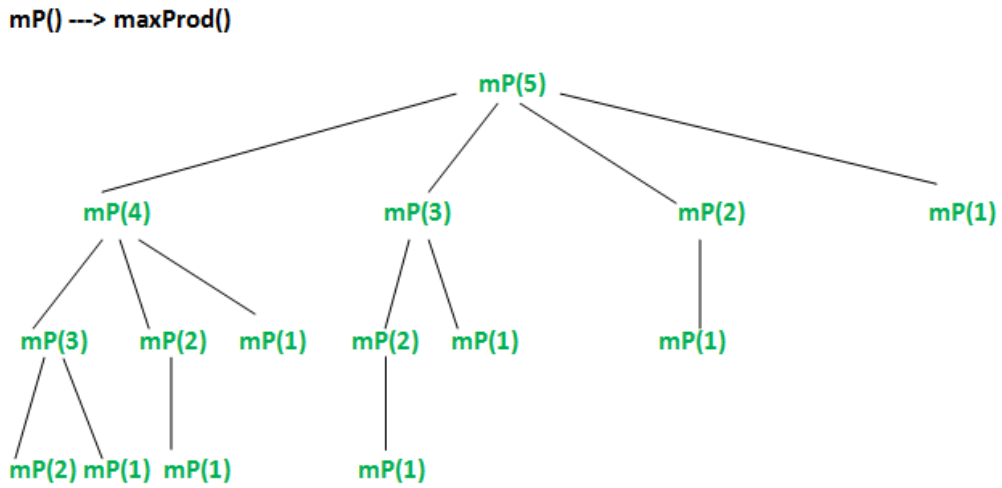
Time complexity: O(n^2)

The time complexity of the maxProd function is O(n^2), because it contains a loop that iterates n-1 times, and inside the loop it calls itself recursively with a smaller input size (n-i). The maximum recursion depth is n, so the total time complexity is n * (n-1) = O(n^2).

Space complexity: O(n)

The space complexity of the maxProd function is O(n), because the maximum recursion depth is n, and each recursive call adds a new activation record to the call stack, which contains local variables and return addresses. Therefore, the space used by the call stack is proportional to the input size n.

Considering the above implementation, following is recursion tree for a Rope of length 5.



In the above partial recursion tree, mP(3) is being solved twice. We can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the problem has both properties (see this and this)

of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array val[] in bottom up manner.

C++ C Java Python3 C# Javascript

```cpp
// C++ code to implement the approach\

// A Dynamic Programming solution for Max Product Problem
int maxProd(int n)
{
   int val[n+1];
   val[0] = val[1] = 0;

   // Build the table val[] in bottom up manner and return
   // the last entry from the table
   for (int i = 1; i <= n; i++)
   {
      int max_val = 0;
      for (int j = 1; j <= i; j++)
         max_val = max(max_val, (i-j)*j, j*val[i-j]);
      val[i] = max_val;
   }
   return val[n];
}

// This code is contributed by sanjoy_62.
```

Time Complexity of the Dynamic Programming solution is O(n^2) and it requires O(n) extra space.

A Tricky Solution:

If we see some examples of this problems, we can easily observe following pattern.
The maximum product can be obtained be repeatedly cutting parts of size 3 while size is greater than 4, keeping the last part as size of 2 or 3 or 4. For example, n = 10, the maximum product is obtained by 3, 3, 4. For n = 11, the maximum product is obtained by 3, 3, 3, 2. Following is the implementation of this approach.

C++ Java Python3 C# Javascript PHP

Min Cost Path | DP-6

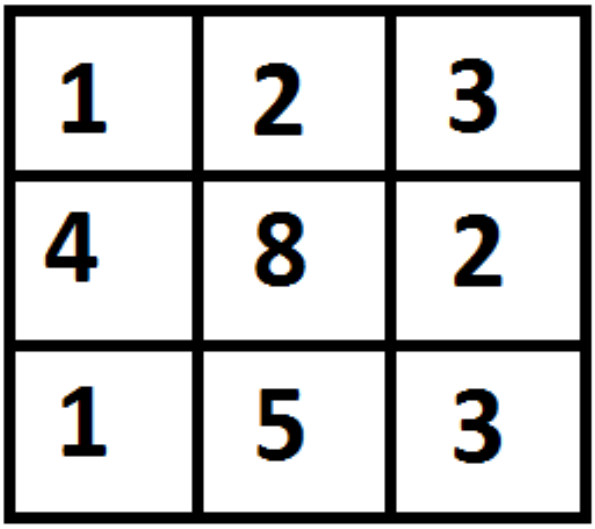
Last Updated : 09 Oct, 2023

Given a cost matrix cost[][] and a position (M, N) in cost[][], write a function that returns cost of minimum cost path to reach (M, N) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (M, N) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed.

Note: You may assume that all costs are positive integers.

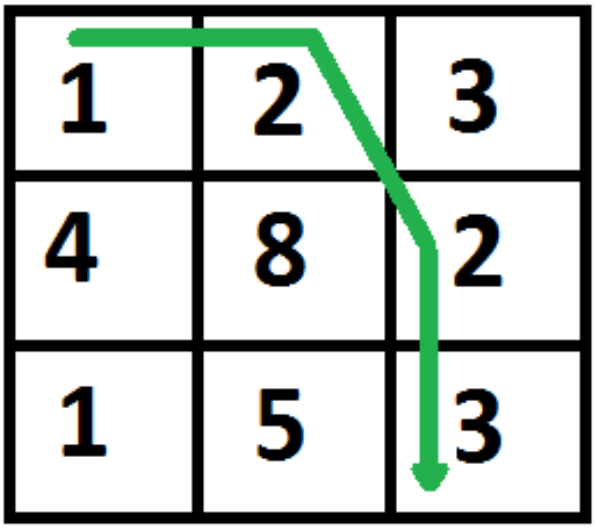
Example:

Input:



The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).

Output:



Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

Min cost path using recursion:

To solve the problem follow the below idea:

This problem has the optimal substructure property. The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as "minimum of the 3 cells plus cost[m][n]".
minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

Follow the below steps to solve the problem:

- If N is less than zero or M is less than zero then return Integer Maximum(Base Case)
- If M is equal to zero and N is equal to zero then return cost[M][N](Base Case)
- Return cost[M][N] + minimum of (minCost(M-1, N-1), minCost(M-1, N), minCost(M, N-1))

Below is the implementation of the above approach:

C++

```
// A Naive recursive implementation
// of MCP(Minimum Cost Path) problem
#include <bits/stdc++.h>
using namespace std;

#define R 3
#define C 3

int min(int x, int y, int z);

// Returns cost of minimum cost path
// from (0,0) to (m, n) in mat[R][C]
int minCost(int cost[R][C], int m, int n)
{
    if (n < 0 || m < 0)
        return INT_MAX;
    else if (m == 0 && n == 0)
        return cost[m][n];
    else
        return cost[m][n]
               + min(minCost(cost, m - 1, n - 1),
                     minCost(cost, m - 1, n),
                     minCost(cost, m, n - 1));
}

// A utility function that returns
// minimum of 3 integers
int min(int x, int y, int z)
{
    if (x < y)
        return (x < z) ? x : z;
    else
        return (y < z) ? y : z;
}

// Driver code
int main()
{
    int cost[R][C]
        = { { 1, 2, 3 }, { 4, 8, 2 }, { 1, 5, 3 } };

    cout << minCost(cost, 2, 2) << endl;

    return 0;
```

```python
# A Naive recursive implementation of MCP(Minimum Cost Path) problem
import sys
R = 3
C = 3

# Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]


def minCost(cost, m, n):
    if (n < 0 or m < 0):
        return sys.maxsize
    elif (m == 0 and n == 0):
        return cost[m][n]
    else:
        return cost[m][n] + min(minCost(cost, m-1, n-1),
                                minCost(cost, m-1, n),
                                minCost(cost, m, n-1))

# A utility function that returns minimum of 3 integers */


def min(x, y, z):
    if (x < y):
        return x if (x < z) else z
    else:
        return y if (y < z) else z


# Driver code
cost = [[1, 2, 3],
        [4, 8, 2],
        [1, 5, 3]]
print(minCost(cost, 2, 2))

# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```csharp
/* A Naive recursive implementation of
MCP(Minimum Cost Path) problem */
using System;

class GFG {

    /* A utility function that
```

```
if ($n < 0 || $m < 0)
    return PHP_INT_MAX;
else if ($m == 0 && $n == 0)
    return $cost[$m][$n];
else
    return $cost[$m][$n] +
            min1(minCost($cost, $m - 1, $n - 1),
            minCost($cost, $m - 1, $n),
            minCost($cost, $m, $n - 1) );
}

/* A utility function that
returns minimum of 3 integers */
function min1($x, $y, $z)
{
if ($x < $y)
    return ($x < $z)? $x : $z;
else
    return ($y < $z)? $y : $z;
}

// Driver Code
$cost = array(array(1, 2, 3),
              array (4, 8, 2),
              array (1, 5, 3));
echo minCost($cost, 2, 2);

// This code is contributed by mits.
?>
```

Output

```
8
```

Time Complexity: $O((M * N)^3)$

Auxiliary Space: O(M + N), for recursive stack space

Min cost path using Memoization DP:

Optimal Strategy for a Game | DP-31

Last Updated : 11 Mar, 2024

Consider a row of N coins of values V1 . . . Vn, where N is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Examples:

Input: *{5, 3, 7, 10}*
Output: *15 -> (10 + 5)*

Input: *{8, 15, 3, 7}*
Output: *22 -> (7 + 15)*

Why greedy algorithm fails here?

Does choosing the best at each move give an optimal solution? No.
In the second example, this is how the game can be finished in two ways:

1. .....The user chooses 8.
   .....The opponent chooses 15.
   .....The user chooses 7.
   .....The opponent chooses 3.
   The total value collected by the user is 15(8 + 7)
2. .....The user chooses 7.
   .....The opponent chooses 8.
   .....The user chooses 15.
   .....The opponent chooses 3.
   The total value collected by the user is 22(7 + 15)

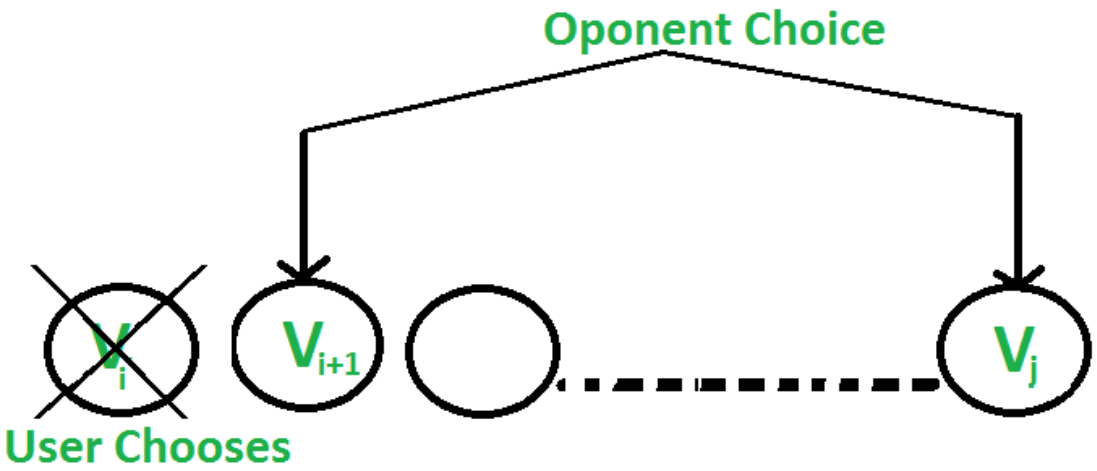
Note: If the user follows the second game state, the maximum value can be collected although the first move is not the best.

Optimal Strategy for a Game using memoization:

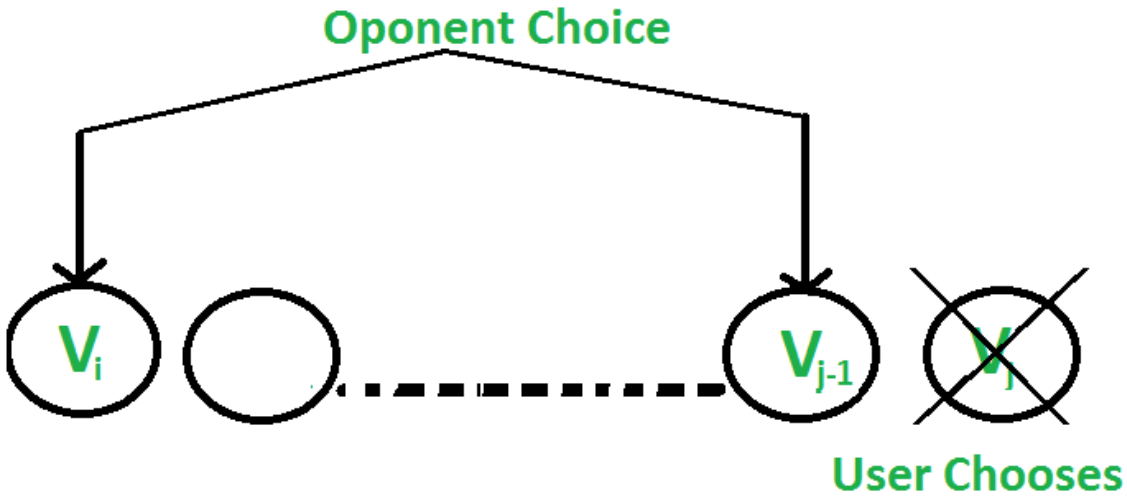
To solve the problem follow the below idea:

There are two choices:

- *The user chooses the 'ith' coin with value 'Vi': The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with* ***minimum value****.*
  i.e. The user can collect the value ***Vi + min(F(i+2, j), F(i+1, j-1))*** *where [i+2,j] is the range of array indices available to the user if the opponent chooses Vi+1 and [i+1,j-1] is the range of array indexes available if opponent chooses the jth coin.*



- *The user chooses the 'jth' coin with value 'Vj': The opponent either chooses 'ith' coin or '(j-1)th' coin. The opponent intends to choose the coin which leaves the user with the minimum value, i.e. the user can collect the value* ***Vj + min(F(i+1, j-1), F(i, j-2))*** *where [i,j-2] is the range of array indices available for the user if the opponent picks jth coin and [i+1,j-1] is the range of indices available to the user if the opponent picks up the ith coin.*



Below is the recursive approach that is based on the above two choices. We take a maximum of two choices.

F(i, j) represents the maximum value the user can collect from i'th coin to j'th coin.

F(i, j) = Max(Vi + min(F(i+2, j), F(i+1, j-1)),
Vj + min(F(i+1, j-1), F(i, j-2)))

As user wants to maximise the number of coins.

Base Cases

F(i, j) = Vi If j == i
F(i, j) = max(Vi, Vj) If j == i + 1

Below is the implementation of the above approach:

C++

```
// C++ code to implement the approach

#include <bits/stdc++.h>
using namespace std;

vector<int> arr;
```

```
    map<vector<int>, int> memo;
    int n = arr.size();

    // recursive top down memoized solution
    int solve(int i, int j)
    {
        if ((i > j) || (i >= n) || (j < 0))
            return 0;

        vector<int> k{ i, j };
        if (memo[k] != 0)
            return memo[k];

        // if the user chooses ith coin, the opponent can choose
        // from i+1th or jth coin. if he chooses i+1th coin,
        // user is left with [i+2,j] range. if opp chooses jth
        // coin, then user is left with [i+1,j-1] range to
        // choose from. Also opponent tries to choose in such a
        // way that the user has minimum value left.
        int option1
            = arr[i]
              + min(solve(i + 2, j), solve(i + 1, j - 1));

        // if user chooses jth coin, opponent can choose ith
        // coin or j-1th coin. if opp chooses ith coin,user can
        // choose in range [i+1,j-1]. if opp chooses j-1th coin,
        // user can choose in range [i,j-2].
        int option2
            = arr[j]
              + min(solve(i + 1, j - 1), solve(i, j - 2));

        // since the user wants to get maximum money
        memo[k] = max(option1, option2);
        return memo[k];
    }

    int optimalStrategyOfGame()
    {

        memo.clear();
        return solve(0, n - 1);
    }

    // Driver code
    int main()
    {
        arr.push_back(8);
        arr.push_back(15);
        arr.push_back(3);
```

```
    arr.push_back(7);
    n = arr.size();
    cout << optimalStrategyOfGame() << endl;

    arr.clear();
    arr.push_back(2);
    arr.push_back(2);
    arr.push_back(2);
    arr.push_back(2);
    n = arr.size();
    cout << optimalStrategyOfGame() << endl;

    arr.clear();
    arr.push_back(20);
    arr.push_back(30);
    arr.push_back(2);
    arr.push_back(2);
    arr.push_back(2);
    arr.push_back(10);
    n = arr.size();
    cout << optimalStrategyOfGame() << endl;
}

// This code is contributed by phasing17
```

Java

```
// Java code to implement the approach
import java.util.*;

class GFG {
    static ArrayList<Integer> arr = new ArrayList<>();
    static HashMap<ArrayList<Integer>, Integer> memo
        = new HashMap<>();
    static int n = 0;

    // recursive top down memoized solution
    static int solve(int i, int j)
    {
        if ((i > j) || (i >= n) || (j < 0))
            return 0;

        ArrayList<Integer> k = new ArrayList<Integer>();
        k.add(i);
        k.add(j);
        if (memo.containsKey(k))
            return memo.get(k);
```

```
print(optimalStrategyOfGame(arr1, n))

arr2 = [2, 2, 2, 2]
n = len(arr2)
print(optimalStrategyOfGame(arr2, n))

arr3 = [20, 30, 2, 2, 2, 10]
n = len(arr3)
print(optimalStrategyOfGame(arr3, n))

# This code is contributed
# by sahilshelangia
```

Output

```
22
4
42
```

Time complexity: O(n^2), The time complexity of this approach is O(n^2) as we are using memoization to store the subproblem solutions which are calculated again and again.
Auxiliary Space: O(n^2), The space complexity of this approach is O(n^2) as we are using a map of size n^2 to store the solutions of the subproblems.

Optimal Strategy for a Game using dp:

To solve the problem follow the below idea:

> *Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the re-computations of the same subproblems can be avoided by constructing a temporary array in a bottom-up manner using the above recursive formula.*

Follow the below steps to solve the problem:

- Create a 2-D array table of size N * N

Partition a set into two subsets such that the difference of subset sums is minimum

Last Updated : 16 Oct, 2023

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.
If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of abs(sum(Subset1) – sum(Subset2)) should be minimum.

Example:

```
Input:  arr[] = {1, 6, 11, 5}
Output: 1
Explanation:
Subset1 = {1, 5, 6}, sum of Subset1 = 12
Subset2 = {11}, sum of Subset2 = 11
```

Recommended Problem

Minimum sum partition

Dynamic Programming Algorithms Amazon Samsung

Solve Problem

Submission count: 1.4L

This problem is mainly an extension to the Dynamic Programming| Set 18 (Partition Problem).

Recursive Solution:

The recursive approach is to generate all possible sums from all the values of the array and to check which solution is the most optimal one. To generate sums we either include the i’th item in set 1 or don’t include, i.e., include in set 2.

C++

```
// A Recursive C++ program to solve minimum sum partition
// problem.
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum sum
int findMinRec(int arr[], int i, int sumCalculated,
               int sumTotal)
{
    // If we have reached last element.  Sum of one
    // subset is sumCalculated, sum of other subset is
    // sumTotal-sumCalculated.  Return absolute difference
    // of two sums.
    if (i == 0)
        return abs((sumTotal - sumCalculated)
                   - sumCalculated);

    // For every item arr[i], we have two choices
    // (1) We do not include it first set
    // (2) We include it in first set
    // We return minimum of two choices
    return min(
        findMinRec(arr, i - 1, sumCalculated + arr[i - 1],
                   sumTotal),
        findMinRec(arr, i - 1, sumCalculated, sumTotal));
}

// Returns minimum possible difference between sums
// of two subsets
int findMin(int arr[], int n)
{
    // Compute total sum of elements
    int sumTotal = 0;
    for (int i = 0; i < n; i++)
        sumTotal += arr[i];
```

```
    // Compute result using recursive function
    return findMinRec(arr, n, 0, sumTotal);
}

// Driver program to test above function
int main()
{
    int arr[] = { 3, 1, 4, 2, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The minimum difference between two sets is "
         << findMin(arr, n);
    return 0;
}
```

Java

```
// JAVA code to partition a set into two subsets
// such that the difference of subset sums
// is minimum
import java.util.*;

class GFG {

    // Function to find the minimum sum
    public static int findMinRec(int arr[], int i,
                                 int sumCalculated,
                                 int sumTotal)
    {
        // If we have reached last element.
        // Sum of one subset is sumCalculated,
        // sum of other subset is sumTotal-
        // sumCalculated.  Return absolute
        // difference of two sums.
        if (i == 0)
            return Math.abs((sumTotal - sumCalculated)
                            - sumCalculated);

        // For every item arr[i], we have two choices
        // (1) We do not include it first set
        // (2) We include it in first set
        // We return minimum of two choices
        return Math.min(
            findMinRec(arr, i - 1,
                       sumCalculated + arr[i - 1],
                       sumTotal),
            findMinRec(arr, i - 1, sumCalculated,
                       sumTotal));
```

```
        return findMinRec(arr, n, 0, sumTotal);
    }

     /* Driver program to test above function */
    let arr=[3, 1, 4, 2, 2, 1];
    let n = arr.length;
    document.write("The minimum difference"+
                        " between two sets is " +
                         findMin(arr, n));

    // This code is contributed by rag2127
```

Output

```
The minimum difference between two sets is 1
```

Time Complexity:

```
All the sums can be generated by either
(1) including that element in set 1.
(2) without including that element in set 1.
So possible combinations are :-
arr[0]      (1 or 2)  -> 2 values
arr[1]    (1 or 2)  -> 2 values
.
.
.
arr[n]     (2 or 2)  -> 2 values
So time complexity will be 2*2*..... *2 (For n times),
that is O(2^n).
```

Auxiliary Space: O(n), extra space for the recursive function call stack.

An approach using Memoization:

Simplify the process by considering the concepts of taking and not taking

Time Complexity: O(n*sum) where n is the number of elements and sum is the sum of all elements.
Auxiliary Space: O(n*sum)

An approach using dynamic Programming:
The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array dp[n+1][sum+1] where n is the number of elements in a given set and sum is the sum of all elements. We can construct the solution in a bottom-up manner.

```
The task is to divide the set into two parts.
We will consider the following factors for dividing it.
Let
  dp[i][j] = {1 if some subset from 1st to i'th has a sum
                      equal to j
                   0 otherwise}

    i ranges from {1..n}
    j ranges from {0..(sum of all elements)}
So
    dp[i][j]  will be 1 if
    1) The sum j is achieved including i'th item
    2) The sum j is achieved excluding i'th item.
Let sum of all the elements be S.
To find Minimum sum difference, we have to find j such
that Min{sum - 2*j  : dp[n][j]  == 1 }
    where j varies from 0 to sum/2
The idea is, sum of S1 is j and it should be closest
to sum/2, i.e., 2*j should be closest to sum (as this will ideally
minimize sum-2*j.
```

Below is the implementation of the above code.

C++

```cpp
// A Recursive C++ program to solve minimum sum partition
// problem.
#include <bits/stdc++.h>
using namespace std;

// Returns the minimum value of the difference of the two
// sets.
int findMin(int arr[], int n)
{
    // Calculate sum of all elements
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // Create an array to store results of subproblems
    bool dp[n + 1][sum + 1];

    // Initialize first column as true. 0 sum is possible
    // with all elements.
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    // Initialize top row, except dp[0][0], as false. With
    // 0 elements, no other sum except 0 is possible
    for (int i = 1; i <= sum; i++)
        dp[0][i] = false;

    // Fill the partition table in bottom up manner
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            // If i'th element is excluded
            dp[i][j] = dp[i - 1][j];

            // If i'th element is included
            if (arr[i - 1] <= j)
                dp[i][j] |= dp[i - 1][j - arr[i - 1]];
        }
    }

    // Initialize difference of two sums.
    int diff = INT_MAX;

    // Find the largest j such that dp[n][j]
    // is true where j loops from sum/2 t0 0
    for (int j = sum / 2; j >= 0; j--) {
        // Find the
        if (dp[n][j] == true) {
            diff = sum - 2 * j;
```

```
            break;
        }
    }
    return diff;
}

// Driver program to test above function
int main()
{
    int arr[] = { 3, 1, 4, 2, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The minimum difference between 2 sets is "
         << findMin(arr, n);
    return 0;
}
```

Java

```
// A Recursive java program to solve
// minimum sum partition problem.
import java.io.*;

class GFG {
    // Returns the minimum value of
    // the difference of the two sets.
    static int findMin(int arr[], int n)
    {
        // Calculate sum of all elements
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += arr[i];

        // Create an array to store
        // results of subproblems
        boolean dp[][] = new boolean[n + 1][sum + 1];

        // Initialize first column as true.
        // 0 sum is possible  with all elements.
        for (int i = 0; i <= n; i++)
            dp[i][0] = true;

        // Initialize top row, except dp[0][0],
        // as false. With 0 elements, no other
        // sum except 0 is possible
        for (int i = 1; i <= sum; i++)
            dp[0][i] = false;

        // Fill the partition table
```

Partition problem | DP-18

Last Updated : 22 Feb, 2023

The partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same.

Examples:

Input: *arr[] = {1, 5, 11, 5}*
Output: *true*
The array can be partitioned as {1, 5, 5} and {11}

Input: *arr[] = {1, 5, 3}*
Output: *false*
The array cannot be partitioned into equal sum sets.

We strongly recommend that you click here and practice it, before moving on to the solution.

The following are the two main steps to solve this problem:

- Calculate the sum of the array. If the sum is odd, there can not be two subsets with an equal sum, so return false.
- If the sum of the array elements is even, calculate sum/2 and find a subset of the array with a sum equal to sum/2.
  The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Partition problem using recursion:

To solve the problem follow the below idea:

Let isSubsetSum(arr, n, sum/2) be the function that returns true if there is a subset of arr[0..n-1] with sum equal to sum/2
The isSubsetSum problem can be divided into two subproblems

- *isSubsetSum() without considering last element (reducing n to n-1)*
- *isSubsetSum considering the last element (reducing sum/2 by arr[n-1] and n to n-1)*

If any of the above subproblems return true, then return true.
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||
isSubsetSum (arr, n-1, sum/2 – arr[n-1])

Follow the below steps to solve the problem:

- First, check if the sum of the elements is even or not
- After checking, call the recursive function isSubsetSum with parameters as input array, array size, and sum/2
  - If the sum is equal to zero then return true (Base case)
  - If n is equal to 0 and sum is not equal to zero then return false (Base case)
  - Check if the value of the last element is greater than the remaining sum then call this function again by removing the last element
  - else call this function again for both the cases stated above and return true, if anyone of them returns true
- Print the answer

Below is the implementation of the above approach:

C++

```
// A recursive C++ program for partition problem
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns true if there is
// a subset of arr[] with sum equal to given sum
bool isSubsetSum(int arr[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then
    // ignore it
    if (arr[n - 1] > sum)
        return isSubsetSum(arr, n - 1, sum);
```

```
    /* else, check if sum can be obtained by any of
        the following
        (a) including the last element
        (b) excluding the last element
    */
    return isSubsetSum(arr, n - 1, sum)
           || isSubsetSum(arr, n - 1, sum - arr[n - 1]);
}

// Returns true if arr[] can be partitioned in two
// subsets of equal sum, otherwise false
bool findPartiion(int arr[], int n)
{
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // If sum is odd, there cannot be two subsets
    // with equal sum
    if (sum % 2 != 0)
        return false;

    // Find if there is subset with sum equal to
    // half of total sum
    return isSubsetSum(arr, n, sum / 2);
}

// Driver code
int main()
{
    int arr[] = { 3, 1, 5, 9, 12 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    if (findPartiion(arr, n) == true)
        cout << "Can be divided into two subsets "
                "of equal sum";
    else
        cout << "Can not be divided into two subsets"
                " of equal sum";
    return 0;
}

// This code is contributed by rathbhupendra
```

C

Output

```
Can be divided into two subsets of equal sum
```

Time Complexity: O(2^N) In the worst case, this solution tries two possibilities (whether to include or exclude) for every element.
Auxiliary Space: O(N). Recursion stack space

Partition problem using memoization:

To solve the problem follow the below idea:

> *As the above recursive solution has overlapping subproblems so we can declare a 2-D array to save the values for different states of the recursive function instead of solving them more than once*

Follow the below steps to solve the problem:

- Declare a 2-D array of size N+1 X sum+1
- Call the recursive function with parameters as input array, size, sum, and dp array
- In this recursive function
  - If the sum is equal to zero then return true (Base case)
  - If n is equal to 0 and sum is not equal to zero then return false (Base case)
  - If the value of this subproblem is already calculated then return the answer from dp array
  - Else calculate the answer for this subproblem using the recursive formula in the above approach and save the answer in the dp array
  - Return the answer as true or false
- Print the answer

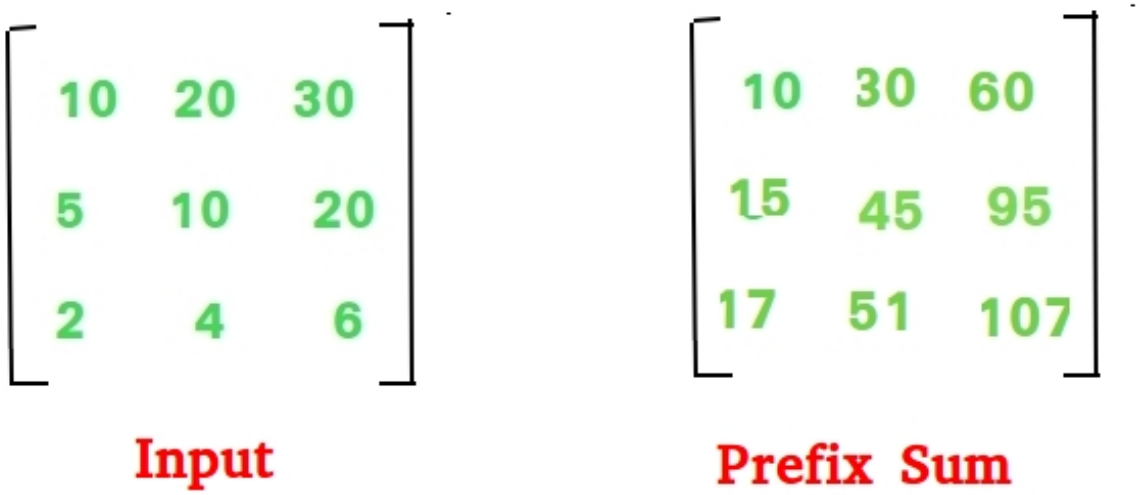
Below is the implementation of the above approach:

C++

Prefix Sum of Matrix (Or 2D Array)

Last Updated : 29 Mar, 2024

Given a matrix (or 2D array) **a[][]** of integers, find the prefix sum matrix for it. Let prefix sum matrix be psa[][]. The value of psa[i][j] contains the sum of all values which are above it or on the left of it.



Recommended Practice

Prefix Sum of Matrix (Or 2D Array) Try It!

Prerequisite: Prefix Sum – 1D

A **simple solution** is to find psa[i][j] by traversing and adding values from a[0][0] to a[i][j]. Time complexity of this solution is O(R * C * R * C).

An **efficient solution** is to use previously computed values to compute psa[i][j]. Unlike 1D array prefix sum, this is tricky, here if we simply add psa[i][j-1] and psa[i-1][j], we get sum of elements from a[0][0] to a[i-1][j-1] twice, so we subtract psa[i-1][j-1].

Example :

```
psa[3][3] = psa[2][3] + psa[3][2] -
            psa[2][2] + a[3][3]
          = 6 + 6 - 4 + 1
          = 9
The general formula:
psa[i][j] = psa[i-1][j] + psa[i][j-1] -
            psa[i-1][j-1] + a[i][j]

Corner Cases (First row and first column)
If i = 0 and j = 0
   psa[i][j] = a[i][j]
If i = 0 and j > 0
   psa[i][j] = psa[i][j-1] + a[i][j]
If i > 0 and j = 0
```

DSA Data Structures Array String Linked List Stack Queue Tree Binary Tree Binary Search Tree

Below is the implementation of the above approach

C++

```
// C++ Program to find prefix sum of 2d array
#include <bits/stdc++.h>
using namespace std;

#define R 4
#define C 5

// calculating new array
void prefixSum2D(int a[][C])
{
```

```
    int psa[R][C];
    psa[0][0] = a[0][0];

    // Filling first row and first column
    for (int i = 1; i < C; i++)
        psa[0][i] = psa[0][i - 1] + a[0][i];
    for (int i = 1; i < R; i++)
        psa[i][0] = psa[i - 1][0] + a[i][0];

    // updating the values in the cells
    // as per the general formula
    for (int i = 1; i < R; i++) {
        for (int j = 1; j < C; j++)

            // values in the cells of new
            // array are updated
            psa[i][j] = psa[i - 1][j] + psa[i][j - 1]
                        - psa[i - 1][j - 1] + a[i][j];
    }

    // displaying the values of the new array
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++)
            cout << psa[i][j] << " ";
        cout << "\n";
    }
}

// driver code
int main()
{
    int a[R][C] = { { 1, 1, 1, 1, 1 },
                    { 1, 1, 1, 1, 1 },
                    { 1, 1, 1, 1, 1 },
                    { 1, 1, 1, 1, 1 } };

    prefixSum2D(a);

    return 0;
}
```

Java

```
// Java program to find prefix sum of 2D array
import java.util.*;

class GFG {
```

```
    // calculating new array
    public static void prefixSum2D(int a[][])
    {
        int R = a.length;
        int C = a[0].length;

        int psa[][] = new int[R][C];

        psa[0][0] = a[0][0];

        // Filling first row and first column
        for (int i = 1; i < C; i++)
            psa[0][i] = psa[0][i - 1] + a[0][i];
        for (int i = 1; i < R; i++)
            psa[i][0] = psa[i - 1][0] + a[i][0];

        // updating the values in the
        // cells as per the general formula.
        for (int i = 1; i < R; i++)
            for (int j = 1; j < C; j++)

                // values in the cells of new array
                // are updated
                psa[i][j] = psa[i - 1][j] + psa[i][j - 1]
                            - psa[i - 1][j - 1] + a[i][j];

        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++)
                System.out.print(psa[i][j] + " ");
            System.out.println();
        }
    }

    // driver code
    public static void main(String[] args)
    {
        int a[][] = { { 1, 1, 1, 1, 1 },
                      { 1, 1, 1, 1, 1 },
                      { 1, 1, 1, 1, 1 },
                      { 1, 1, 1, 1, 1 } };
        prefixSum2D(a);
    }
}
```

Python3

```
# Python Program to find
# prefix sum of 2d array
```

```
R = 4
C = 5

# calculating new array
def prefixSum2D(a) :
    global C, R
    psa = [[0 for x in range(C)]
              for y in range(R)]
    psa[0][0] = a[0][0]

    # Filling first row
    # and first column
    for i in range(1, C) :
        psa[0][i] = (psa[0][i - 1] +
                        a[0][i])
    for i in range(0, R) :
        psa[i][0] = (psa[i - 1][0] +
                        a[i][0])

    # updating the values in
    # the cells as per the
    # general formula
    for i in range(1, R) :
        for j in range(1, C) :

            # values in the cells of
            # new array are updated
            psa[i][j] = (psa[i - 1][j] +
                         psa[i][j - 1] -
                         psa[i - 1][j - 1] +
                           a[i][j])

    # displaying the values
    # of the new array
    for i in range(0, R) :
        for j in range(0, C) :
            print (psa[i][j],
                   end = " ")
        print ()

# Driver Code
a = [[ 1, 1, 1, 1, 1 ],
     [ 1, 1, 1, 1, 1 ],
     [ 1, 1, 1, 1, 1 ],
     [ 1, 1, 1, 1, 1 ]]

prefixSum2D(a)

# This code is contributed by
```

```
# Manish Shaw(manishshaw1)
```

C#

```
// C# program to find prefix
// sum of 2D array
using System;

class GFG
{

    // calculating new array
    static void prefixSum2D(int [,]a)
    {
        int R = a.GetLength(0);
        int C = a.GetLength(1);

        int [,]psa = new int[R, C];

        psa[0, 0] = a[0, 0];

        // Filling first row
        // and first column
        for (int i = 1; i < C; i++)
            psa[0, i] = psa[0, i - 1] +
                                a[0, i];
        for (int i = 1; i < R; i++)
            psa[i, 0] = psa[i - 1, 0] +
                                a[i, 0];

        // updating the values in the
        // cells as per the general formula.
        for (int i = 1; i < R; i++)
            for (int j = 1; j < C; j++)

                // values in the cells of
                // new array are updated
                psa[i, j] = psa[i - 1, j] +
                            psa[i, j - 1] -
                            psa[i - 1, j - 1] +
                            a[i, j];

        for (int i = 0; i < R; i++)
        {
            for (int j = 0; j < C; j++)
                Console.Write(psa[i, j] + " ");
            Console.WriteLine();
        }
```

```
    }

    // Driver Code
    static void Main()
    {
        int [,]a = new int[,]{{1, 1, 1, 1, 1},
                              {1, 1, 1, 1, 1},
                              {1, 1, 1, 1, 1},
                              {1, 1, 1, 1, 1}};
        prefixSum2D(a);
    }
}

// This code is contributed by manishshaw1
```

PHP

```
<?php
// PHP Program to find
// prefix sum of 2d array
$R = 4;
$C = 5;

// calculating new array
function prefixSum2D($a)
{
    global $C, $R;
    $psa = array();
    $psa[0][0] = $a[0][0];

    // Filling first row
    // and first column
    for ($i = 1; $i < $C; $i++)
        $psa[0][$i] = $psa[0][$i - 1] +
                            $a[0][$i];
    for ($i = 0; $i < $R; $i++)
        $psa[$i][0] = $psa[$i - 1][0] +
                            $a[$i][0];

    // updating the values in
    // the cells as per the
    // general formula
    for ($i = 1; $i < $R; $i++)
    {
        for ($j = 1; $j < $C; $j++)

            // values in the cells of
            // new array are updated
```

```
            $psa[$i][$j] = $psa[$i - 1][$j] +
                           $psa[$i][$j - 1] -
                           $psa[$i - 1][$j - 1] +
                           $a[$i][$j];
    }

    // displaying the values
    // of the new array
    for ($i = 0; $i < $R; $i++)
    {
        for ($j = 0; $j < $C; $j++)
            echo ($psa[$i][$j]. " ");
        echo ("\n");
    }
}

// Driver Code
$a = array(array( 1, 1, 1, 1, 1 ),
           array( 1, 1, 1, 1, 1 ),
           array( 1, 1, 1, 1, 1 ),
           array( 1, 1, 1, 1, 1 ));

prefixSum2D($a);

// This code is contributed by
// Manish Shaw(manishshaw1)
?>
```

Javascript

```
// Javascript program to find prefix sum of 2D array

// calculating new array
function prefixSum2D(a)
{
    let R = a.length;
        let C = a[0].length;

        let psa = new Array(R);
        for(let i = 0; i < R; i++)
        {
            psa[i] = new Array(C);
            for(let j = 0; j < C; j++)
                psa[i][j] = 0;
        }

        psa[0][0] = a[0][0];
```

```
        // Filling first row and first column
        for (let i = 1; i < C; i++)
            psa[0][i] = psa[0][i - 1] + a[0][i];
        for (let i = 1; i < R; i++)
            psa[i][0] = psa[i - 1][0] + a[i][0];

        // updating the values in the
        // cells as per the general formula.
        for (let i = 1; i < R; i++)
            for (let j = 1; j < C; j++)

                // values in the cells of new array
                // are updated
                psa[i][j] = psa[i - 1][j] + psa[i][j - 1]
                            - psa[i - 1][j - 1] + a[i][j];

        for (let i = 0; i < R; i++) {
            for (let j = 0; j < C; j++)
                document.write(psa[i][j] + " ");
            document.write("<br>");
        }
}

// driver code
let a=[[ 1, 1, 1, 1, 1 ],
                    [ 1, 1, 1, 1, 1 ],
                    [ 1, 1, 1, 1, 1 ],
                    [ 1, 1, 1, 1, 1 ]];
prefixSum2D(a);

// This code is contributed by avanitrachhadiya2155
```

Output

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

Time Complexity: O(R*C)
Auxiliary Space: O(R*C)

Another **Efficient** solution in which we also use the previously calculated sums in two main steps would be:

1. Calculate the vertical prefix sum for each column.
2. Calculate the horizontal prefix sum for each row.

Example

```
// c = the number of columns
// r = the number of rows
// a is the matrix

// calculating the vertical sum for each column in the Matrix
for(column = 0 to column = c-1)
    for(row = 1 to row = r-1)
        a[row][column] += a[row-1][column];

// calculating the horizontal sum for each row in the Matrix
for(row = 0 to row = r-1)
    for(column = 1 to column = c-1)
        a[row][column] += a[row][column -1];
```

Below is the Implementation of the above approach

C++

```
#include <iostream>
#include <iomanip>
using namespace std;
void prefixSum(int arr[3][3], int n);
void print(int arr[3][3], int n);
int main()
{
    int n = 3;
    int arr[3][3] = {{10,20,30},
                     {5, 10, 20},
                     {2, 4, 6}
                    };
    prefixSum(arr, n);
      print(arr, n);

}
void prefixSum(int arr[3][3], int n) {
    //vertical prefixsum
    for (int j = 0; j < n; j++) {
        for (int i = 1; i < n; i++) {
            arr[i][j] += arr[i-1][j];
```

```
        }
    }
    //horizontal prefixsum
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j++) {
            arr[i][j] += arr[i][j-1];
        }
    }
}


void print(int arr[3][3], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << setw(3) << left << arr[i][j] << " ";
        }
        cout << '\n';
    }
}
```

Java

```
import java.io.*;
import java.lang.*;
import java.util.*;
public class GFG {
  public static void main(String[] args)
  {
    int n = 3;
    int arr[][] = new int[][] { { 10, 20, 30 },
                               { 5, 10, 20 },
                               { 2, 4, 6 } };
    prefixSum(arr, n);
    print(arr, n);
  }
  static void prefixSum(int arr[][], int n)
  {

    // vertical prefixsum
    for (int j = 0; j < n; j++) {
      for (int i = 1; i < n; i++) {
        arr[i][j] += arr[i - 1][j];
      }
    }

    // horizontal prefixsum
    for (int i = 0; i < n; i++) {
      for (int j = 1; j < n; j++) {
```

```
        arr[i][j] += arr[i][j - 1];
      }
    }
  }
  static void print(int arr[][], int n)
  {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        System.out.print(arr[i][j] + " ");
      }
      System.out.println();
    }
  }
}

// This code is contributed by ishankhandelwals.
```

Python3

```
def prefixsum(arr, n):
    # vertical prefixsum
    for j in range(n):
        for i in range(1, n):
            arr[i][j] += arr[i - 1][j]

    # horizontal prefixsum
    for i in  range(n):
        for j in range(1, n):
            arr[i][j] += arr[i][j - 1]

def printarr(arr, n):
    for i in range(n):
        for j in range(n):
            print(arr[i][j], end = " ")
        print()

# Driver Code
n = 3
arr = [[10,20,30],[5,10,20],[2,4,6]]
prefixsum(arr,n)
printarr(arr,n)

# This code is contributed by
# Vibhu Karnwal
```

C#

```
// C# code for above approach
using System;
public class gfg
{
  public static void prefixSum(int[,] arr,int n){
    for (int j = 0; j < n; j++) {
      for (int i = 1; i < n; i++) {
        arr[i,j] += arr[i-1,j];
      }
    }

    //horizontal prefixsum
    for (int i = 0; i < n; i++) {
      for (int j = 1; j < n; j++) {
        arr[i,j] += arr[i,j-1];
      }
    }
  }
  public static void print(int[,] arr,int n){
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        Console.Write("{0} ",arr[i,j]);
      }
      Console.WriteLine();
    }

  }

  public static void Main(string[] args)
  {
    int n = 3;
    int[ , ] arr = new int[3,3] {{10,20,30},
                                 {5, 10, 20},
                                 {2, 4, 6}
                                } ;
    prefixSum(arr, n);
    print(arr, n);
  }
}

// This code is contributed by ishankhandelwals.
```

Javascript

```
// Js code for above approach
function prefixSum(arr, n) {
```

```
    //vertical prefixsum
    for (let j = 0; j < n; j++) {
        for (let i = 1; i < n; i++) {
            arr[i][j] += arr[i - 1][j];
        }
    }
    //horizontal prefixsum
    for (let i = 0; i < n; i++) {
        for (let j = 1; j < n; j++) {
            arr[i][j] += arr[i][j - 1];
        }
    }
}
function print(arr, n) {
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            console.log(arr[i][j]);
        }
    }
}
let n = 3;
let arr= [[ 10, 20, 30 ],
        [5, 10, 20 ],
        [ 2, 4, 6 ]];
prefixSum(arr, n);
print(arr, n);

// This code is contributed by ishankhandelwals.
```

Output

```
10  30  60
15  45  95
17  51  107
```

Time Complexity: O(R*C), where R and C are the Rows and Columns of the given matrix respectively.
Auxiliary Space: O(R*C)

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great investment, the passion Sandeep sir has towards DSA/teaching is what

Shortest Common Supersequence

Last Updated : 11 Sep, 2023

Given two strings str1 and str2, the task is to find the length of the shortest string that has both str1 and str2 as subsequences.

Examples :

```
Input:   str1 = "geek",  str2 = "eke"
Output: 5
Explanation:
String "geeke" has both string "geek"
and "eke" as subsequences.

Input:   str1 = "AGGTAB",  str2 = "GXTXAYB"
Output:  9
Explanation:
String "AGXGTXAYB" has both string
"AGGTAB" and "GXTXAYB" as subsequences.
```


Method 1: This problem is closely related to longest common subsequence problem.
Below are steps.

1. Find Longest Common Subsequence (lcs) of two given strings. For example, lcs of “geek” and “eke” is “ek”.
2. Insert non-lcs characters (in their original order in strings) to the lcs found above, and return the result. So “ek” becomes “geeke” which is shortest common supersequence.
   Let us consider another example, str1 = “AGGTAB” and str2 = “GXTXAYB”.

LCS of str1 and str2 is “GTAB”. Once we find LCS, we insert characters of both strings in order and we get “AGXGTXAYB”

How does this work?

We need to find a string that has both strings as subsequences and is the shortest such string. If both strings have all characters different, then result is sum of lengths of two given strings. If there are common characters, then we don’t want them multiple times as the task is to minimize length. Therefore, we first find the longest common subsequence, take one occurrence of this subsequence and add extra characters.

```
Length of the shortest supersequence
= (Sum of lengths of given two strings)
- (Length of LCS of two given strings)
```

Below is the implementation of above idea. The below implementation only finds length of the shortest super sequence.

C++

```cpp
// C++ program to find length of the
// shortest supersequence
#include <bits/stdc++.h>
using namespace std;

// Utility function to get max
// of 2 integers
int max(int a, int b) { return (a > b) ? a : b; }
```

```
// Returns length of LCS for
// X[0..m - 1], Y[0..n - 1]
int lcs(char* X, char* Y, int m, int n);

// Function to find length of the
// shortest supersequence of X and Y.
int shortestSuperSequence(char* X, char* Y)
{
    int m = strlen(X), n = strlen(Y);

    // find lcs
    int l = lcs(X, Y, m, n);

    // Result is sum of input string
    // lengths - length of lcs
    return (m + n - l);
}

// Returns length of LCS
// for X[0..m - 1], Y[0..n - 1]
int lcs(char* X, char* Y, int m, int n)
{
    int L[m + 1][n + 1];
    int i, j;

    // Following steps build L[m + 1][n + 1]
    // in bottom up fashion. Note that
    // L[i][j] contains length of LCS of
    // X[0..i - 1] and Y[0..j - 1]
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;

            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    // L[m][n] contains length of LCS
    // for X[0..n - 1] and Y[0..m - 1]
    return L[m][n];
}

// Driver code
```

```
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    cout << "Length of the shortest supersequence is "
         << shortestSuperSequence(X, Y) << endl;

    return 0;
}

// This code is contributed by Akanksha Rai
```

C

```
// C program to find length of
// the shortest supersequence
#include <stdio.h>
#include <string.h>

// Utility function to get
// max of 2 integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns length of LCS for
// X[0..m - 1], Y[0..n - 1]
int lcs(char* X, char* Y, int m, int n);

// Function to find length of the
// shortest supersequence of X and Y.
int shortestSuperSequence(char* X, char* Y)
{
    int m = strlen(X), n = strlen(Y);

    // find lcs
    int l = lcs(X, Y, m, n);

    // Result is sum of input string
    // lengths - length of lcs
    return (m + n - l);
}

// Returns length of LCS
// for X[0..m - 1], Y[0..n - 1]
int lcs(char* X, char* Y, int m, int n)
{
    int L[m + 1][n + 1];
    int i, j;
```

```
var Y = "GXTXAYB";

document.write("Length of the shortest "
                  + "supersequence is "
                  + shortestSuperSequence(X, Y));

// This code contributed by shikhasingrajput
```

Output

```
Length of the shortest supersequence is 9
```

Time Complexity: O(m*n).
Auxiliary Space: O(m*n)

Method 2: A simple analysis yields below simple recursive solution.

```
Let X[0..m - 1] and Y[0..n - 1] be two
strings and m and n be respective
lengths.

  if (m == 0) return n;
  if (n == 0) return m;

  // If last characters are same, then
  // add 1 to result and
  // recur for X[]
  if (X[m - 1] == Y[n - 1])
     return 1 + SCS(X, Y, m - 1, n - 1);

  // Else find shortest of following two
  //  a) Remove last character from X and recur
  //  b) Remove last character from Y and recur
  else
    return 1 + min( SCS(X, Y, m - 1, n), SCS(X, Y, m, n - 1) );
```

Below is simple naive recursive solution based on above recursive formula.

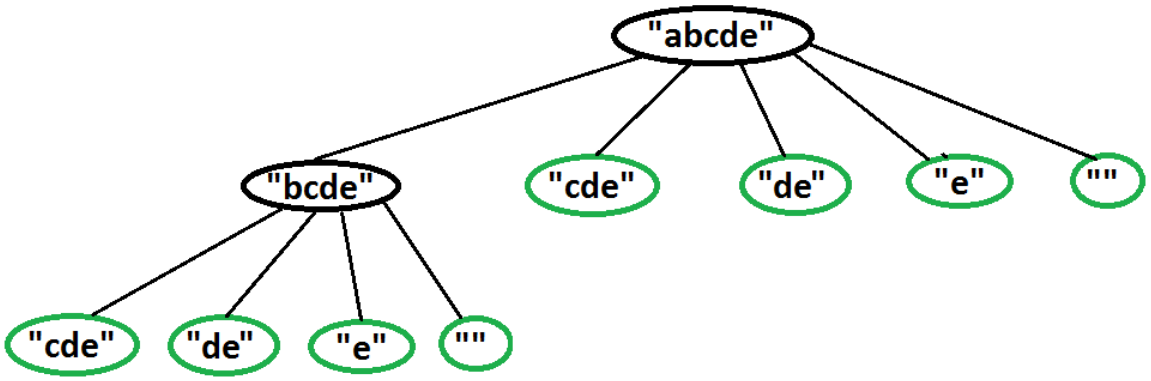
C++

```
// A Naive recursive C++ program to find
```

Time Complexity: The time complexity of the above code will be O(2^n).
Auxiliary Space: The space complexity will be O(n) as we are using recursion and the recursive call stack will take O(n) space.

Dynamic Programming

Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string “abcde” in the worst case.



Partial recursion tree for input string "abcde". The subproblems encircled with green color are overlapping subproblems

CPP

```
// A Dynamic Programming based program to test whether a given string can
// be segmented into space separated words in dictionary
#include <iostream>
#include <string.h>
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
  An array of strings is used for dictionary.  Using array of strings for
  dictionary is definitely not a good idea. We have used for simplicity of
  the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile","samsung","sam","sung","man","mango",
```

```
                                "icecream","and","go","i","like","ice","cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
           return true;
    return false;
}

// Returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();
    if (size == 0)   return true;

    // Create the DP table to store results of subproblems. The value wb[i]
    // will be true if str[0..i-1] can be segmented into dictionary words,
    // otherwise false.
    bool wb[size+1];
    memset(wb, 0, sizeof(wb)); // Initialize all values as false.

    for (int i=1; i<=size; i++)
    {
        // if wb[i] is false, then check if current prefix can make it true.
        // Current prefix is "str.substr(0, i)"
        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
            wb[i] = true;

        // wb[i] is true, then check for all substrings starting from
        // (i+1)th character and store their results.
        if (wb[i] == true)
        {
            // If we reached the last prefix
            if (i == size)
                return true;

            for (int j = i+1; j <= size; j++)
            {
                // Update wb[j] if it is false and can be updated
                // Note the parameter passed to dictionaryContains() is
                // substring starting from index 'i' and length 'j-i'
                if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
                    wb[j] = true;

                // If we reached the last character
                if (j == size && wb[j] == true)
                    return true;
            }
        }
```

```
    }

    /* Uncomment these lines to print DP table "wb[]"
     for (int i = 1; i <= size; i++)
        cout << " " << wb[i]; */

    // If we have tried all prefixes and none of them worked
    return false;
}

// Driver program to test above functions
int main()
{
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}
```

Java

```
// A Dynamic Programming based program to test whether a given String can
// be segmented into space separated words in dictionary
import java.util.*;

class GFG{

/* A utility function to check whether a word is present in dictionary or not.
  An array of Strings is used for dictionary.  Using array of Strings for
  dictionary is definitely not a good idea. We have used for simplicity of
  the program*/
static boolean dictionaryContains(String word)
{
    String dictionary[] = {"mobile","samsung","sam","sung","man","mango",
                           "icecream","and","go","i","like","ice","cream"};
    int size = dictionary.length;
    for (int i = 0; i < size; i++)
        if (dictionary[i].compareTo(word) == 0)
           return true;
    return false;
}

// Returns true if String can be segmented into space separated
// words, otherwise returns false
```

```
            document.write("No<br/>");
        if (wordBreak("ilikelikeimangoiii"))
            document.write("Yes<br/>");
        else
            document.write("No<br/>");
        if (wordBreak("samsungandmango"))
            document.write("Yes<br/>");
        else
            document.write("No<br/>");
        if (wordBreak("samsungandmangok"))
            document.write("Yes<br/>");
        else
            document.write("No<br/>");

// This code contributed by Rajput-Ji
```

Output

```
Yes
Yes
Yes
Yes
Yes
No
```

The time complexity of the given implementation of the wordBreak function is O(n^3), where n is the length of the input string.

The space complexity of the implementation is O(n), as an extra boolean array of size n+1 is created. Additionally, the dictionary array occupies a space of O(kL), where k is the number of words in the dictionary and L is the maximum length of a word in the dictionary. However, since the dictionary is a constant