Dynamic Programming

# 379. Coin Change Problem

Given a value N, find the number of ways to make change for N cents, if we have infinite supply of each of S = { S1, S2, .. , SM} valued coins.

**Example 1:**

**Input:**

n = 4 , m = 3

S[] = {1,2,3}

**Output:** 4

**Explanation**: Four Possible ways are:

{1,1,1,1},{1,1,2},{2,2},{1,3}.

**Example 2:**

**Input**:

n = 10 , m = 4

S[] ={2,5,3,6}

**Output:** 5

**Explanation**: Five Possible ways are:

{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5}

and {5,5}.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **count()**which accepts an array **S[]**its size **m**and **n** as input parameter and returns the number of ways to make change for N cents.

**Expected Time Complexity:**O(m\*n).  
**Expected Auxiliary Space:**O(n).

**Constraints:**  
1 <= n,m <= 103

## Solution:

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.  
For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

**1) Optimal Substructure**   
To count the total number of solutions, we can divide all set solutions into two sets.   
1) Solutions that do not contain mth coin (or Sm).   
2) Solutions that contain at least one Sm.   
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).  
Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems**   
Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

**3) Approach (Algorithm)**

See, here each coin of a given denomination can come an infinite number of times. (Repetition allowed), this is what we call UNBOUNDED KNAPSACK. We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.  But here, the inclusion process is not for just once; we can include any denomination any number of times until N<0.

Basically, If we are at s[m-1], we can take as many instances of that coin ( unbounded inclusion ) i.e **count(S, m, n - S[m-1] )** ; then we move to s[m-2]. After moving to s[m-2], we can't move back and can't make choices for s[m-1] i.e **count(S, m-1, n )**.

Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(S, m, n - S[m-1] ) + count(S, m-1, n ) ;**which will be our required answer.

// Recursive C++ program for

// coin change problem.

#include <bits/stdc++.h>

using namespace std;

// Returns the count of ways we can

// sum S[0...m-1] coins to get sum n

int count(int S[], int m, int n)

{

// If n is 0 then there is 1 solution

// (do not include any coin)

if (n == 0)

return 1;

// If n is less than 0 then no

// solution exists

if (n < 0)

return 0;

// If there are no coins and n

// is greater than 0, then no

// solution exist

if (m <= 0 && n >= 1)

return 0;

// count is sum of solutions (i)

// including S[m-1] (ii) excluding S[m-1]

return count(S, m - 1, n) +

count(S, m, n - S[m - 1]);

}

// Driver code

int main()

{

int i, j;

int arr[] = { 1, 2, 3 };

int m = sizeof(arr) / sizeof(arr[0]);

cout << " " << count(arr, m, 4);

return 0;

}

**Output**

4

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.

The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.

C() --> count()

C({1,2,3}, 5)

/ \

/ \

C({1,2,3}, 2) C({1,2}, 5)

/ \ / \

/ \ / \

C({1,2,3}, -1) C({1,2}, 2) C({1,2}, 3) C({1}, 5)

/ \ / \ / \

/ \ / \ / \

C({1,2},0) C({1},2) C({1,2},1) C({1},3) C({1}, 4) C({}, 5)

/ \ / \ /\ / \

/ \ / \ / \ / \

. . . . . . C({1}, 3) C({}, 4)

/ \

/ \

. .

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Coin Change problem has both properties (see [this](https://www.cdn.geeksforgeeks.org/archives/12635)and [this](https://www.cdn.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.cdn.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

**Dynamic Programming Solution**

// C++ program for coin change problem.

#include<bits/stdc++.h>

using namespace std;

int count( int S[], int m, int n )

{

int i, j, x, y;

// We need n+1 rows as the table

// is constructed in bottom up

// manner using the base case 0

// value case (n = 0)

int table[n + 1][m];

// Fill the entries for 0

// value case (n = 0)

for (i = 0; i < m; i++)

table[0][i] = 1;

// Fill rest of the table entries

// in bottom up manner

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

{

for (j = 0; j < m; j++)

{

// Count of solutions including S[j]

x = (i-S[j] >= 0) ? table[i - S[j]][j] : 0;

// Count of solutions excluding S[j]

y = (j >= 1) ? table[i][j - 1] : 0;

// total count

table[i][j] = x + y;

}

}

return table[n][m - 1];

}

// Driver Code

int main()

{

int arr[] = {1, 2, 3};

int m = sizeof(arr)/sizeof(arr[0]);

int n = 4;

cout << count(arr, m, n);

return 0;

}

**Output**

4

**Time Complexity:** O(mn)   
Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

int count( int S[], int m, int n )

{

// table[i] will be storing the number of solutions for

// value i. We need n+1 rows as the table is constructed

// in bottom up manner using the base case (n = 0)

int table[n+1];

// Initialize all table values as 0

memset(table, 0, sizeof(table));

// Base case (If given value is 0)

table[0] = 1;

// Pick all coins one by one and update the table[] values

// after the index greater than or equal to the value of the

// picked coin

for(int i=0; i<m; i++)

for(int j=S[i]; j<=n; j++)

table[j] += table[j-S[i]];

return table[n];

}

**Output:**

4

Following is another Top Down DP Approach using memoization:

#include <bits/stdc++.h>

using namespace std;

int coinchange(vector<int>& a, int v, int n,

vector<vector<int> >& dp)

{

if (v == 0)

return dp[n][v] = 1;

if (n == 0)

return 0;

if (dp[n][v] != -1)

return dp[n][v];

if (a[n - 1] <= v) {

// Either Pick this coin or not

return dp[n][v] = coinchange(a, v - a[n - 1], n, dp)

+ coinchange(a, v, n - 1, dp);

}

else // We have no option but to lea ve this coin

return dp[n][v] = coinchange(a, v, n - 1, dp);

}

int32\_t main()

{

int tc = 1;

// cin >> tc;

while (tc--) {

int n, v;

n = 3, v = 4;

vector<int> a = { 1, 2, 3 };

vector<vector<int> > dp(n + 1,

vector<int>(v + 1, -1));

int res = coinchange(a, v, n, dp);

cout << res << endl;

}

}

**Output**

4

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

**My Implementation:**

long long int fun(int S[], int m, int n, int ind, vector<vector<long long int>> &dp){

if(dp[n][ind]!=-1)

return dp[n][ind];

if(n==0)

return dp[n][ind] = 1;

if(ind==m)

return dp[n][ind] = 0;

long long ans = 0;

if(S[ind]<=n){

int quo = n/S[ind];

for(int j=0;j<=quo;j++){

ans += fun(S, m, n-(j\*S[ind]), ind+1, dp);

}

}

else{

ans = fun(S, m, n, ind+1, dp);

}

return dp[n][ind] = ans;

}

long long int count(int S[], int m, int n) {

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

return fun(S, m, n, 0, dp);

}

# 380. Knapsack Problem

You are given weights and values of **N** items, put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. Note that we have only **one quantity of each item**.  
In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent values and weights associated with **N** items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of **val[]** such that sum of the weights of this subset is smaller than or equal to **W.** You cannot break an item, **either pick the complete item or dont pick it (0-1 property)**.

**Example 1:**

**Input:**

N = 3

W = 4

values[] = {1,2,3}

weight[] = {4,5,1}

**Output:** 3

**Example 2:**

**Input:**

N = 3

W = 3

values[] = {1,2,3}

weight[] = {4,5,6}

**Output:** 0

**Your Task:**  
Complete the function **knapSack()** which takes maximum capacity W, weight array wt[], value array val[], and the number of items n as a parameter and returns the **maximum possible** value you can get.

**Expected Time Complexity:** O(N\*W).  
**Expected Auxiliary Space:** O(N\*W)

**Constraints:**  
1 ≤ N ≤ 1000  
1 ≤ W ≤ 1000  
1 ≤ wt[i] ≤ 1000  
1 ≤ v[i] ≤ 1000

## Solution:

**Method 1:** Recursion by Brute-Force algorithm OR Exhaustive Search.  
**Approach:** A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.  
***Optimal Sub-structure*:** To consider all subsets of items, there can be two cases for every item.

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

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and **Case 1** is the only possibility.

Below is the implementation of the above approach:

/\* 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 val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout << knapSack(W, wt, val, n);

return 0;

}

**Output**

220

It should be noted that the above function computes the same sub-problems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. The time complexity of this naive recursive solution is exponential (2^n).

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.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(n, W)

K(3, 2)

/ \

/ \

K(2, 2) K(2, 1)

/ \ / \

/ \ / \

K(1, 2) K(1, 1) K(1, 1) K(1, 0)

/ \ / \ / \

/ \ / \ / \

K(0, 2) K(0, 1) K(0, 1) K(0, 0) K(0, 1) K(0, 0)

Recursion tree for Knapsack capacity 2

units and 3 items of 1 unit weight.

**Complexity Analysis:**

* **Time Complexity:** O(2n).   
  As there are redundant subproblems.
* **Auxiliary Space :**O(1).   
  As no extra data structure has been used for storing values.

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0-1 Knapsack problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem.

**Method 2:** Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), re-computation of same subproblems can be avoided by constructing a temporary array K[][] in bottom-up manner. Following is Dynamic Programming based implementation.

**Approach:** In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.   
The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place:

* Fill ‘wi’ in the given column.
* Do not fill ‘wi’ in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualisation will make the concept clear:

Let weight elements = {1, 2, 3}

Let weight values = {10, 15, 40}

Capacity=6

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

**Explanation:**

For filling 'weight = 2' we come

across 'j = 3' in which

we take maximum of

(10, 15 + DP[1][3-2]) = 25

| |

'2' '2 filled'

not filled

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

**Explanation:**

For filling 'weight=3',

we come across 'j=4' in which

we take maximum of (25, 40 + DP[2][4-3])

= 50

For filling 'weight=3'

we come across 'j=5' in which

we take maximum of (25, 40 + DP[2][5-3])

= 55

For filling 'weight=3'

we come across 'j=6' in which

we take maximum of (25, 40 + DP[2][6-3])

= 65

// 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 val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout << knapSack(W, wt, val, n);

return 0;

}

**Output**

220

**Complexity Analysis:**

* **Time Complexity:** O(N\*W).   
  where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
* **Auxiliary Space:** O(N\*W).   
  The use of 2-D array of size ‘N\*W’.

**Scope for Improvement :-**We used the same approach but with optimized space complexity

#include <bits/stdc++.h>

using namespace std;

// we can further improve the above Knapsack function's space

// complexity

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

{

int i, w;

int K[2][W + 1];

// We know we are always using the the current row or

// the previous row of the array/vector . Thereby we can

// improve it further by using a 2D array but with only

// 2 rows i%2 will be giving the index inside the bounds

// of 2d array K

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

for (w = 0; w <= W; w++) {

if (i == 0 || w == 0)

K[i % 2][w] = 0;

else if (wt[i - 1] <= w)

K[i % 2][w] = max(

val[i - 1]

+ K[(i - 1) % 2][w - wt[i - 1]],

K[(i - 1) % 2][w]);

else

K[i % 2][w] = K[(i - 1) % 2][w];

}

}

return K[n % 2][W];

}

// Driver Code

int main()

{

int val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout << knapSack(W, wt, val, n);

return 0;

}

**Complexity Analysis:**

* **Time Complexity:**O(N\*W).
* **Auxiliary Space:**O(2\*W)   
  As we are using a 2-D array but with only 2 rows.

**Method 3:** This method uses Memoization Technique (an extension of recursive approach).  
This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. We can solve this problem by simply creating a 2-D array that can store a particular state (n, w) if we get it the first time. 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. This method gives an edge over the recursive approach in this aspect.

// 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 i,

int\*\* dp)

{

// base condition

if (i < 0)

return 0;

if (dp[i][W] != -1)

return dp[i][W];

if (wt[i] > W) {

// Store the value of function call

// stack in table before return

dp[i][W] = knapSackRec(W, wt,

val, i - 1,

dp);

return dp[i][W];

}

else {

// Store value in a table before return

dp[i][W] = max(val[i]

+ knapSackRec(W - wt[i],

wt, val,

i - 1, dp),

knapSackRec(W, wt, val,

i - 1, dp));

// Return value of table after storing

return dp[i][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 val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout << knapSack(W, wt, val, n);

return 0;

}

**Output**

220

**Complexity Analysis:**

* **Time Complexity:** O(N\*W).   
  As redundant calculations of states are avoided.
* **Auxiliary Space:** O(N\*W).   
  The use of 2D array data structure for storing intermediate states

**Method 4 :-** Again we use the dynamic programming approach with even more optimized space complexity .

#include <bits/stdc++.h>

using namespace std;

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

{

// making and initializing dp array

int dp[W + 1];

memset(dp, 0, sizeof(dp));

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

for (int w = W; w >= 0; w--) {

if (wt[i - 1] <= w)

// finding the maximum value

dp[w] = max(dp[w],

dp[w - wt[i - 1]] + val[i - 1]);

}

}

return dp[W]; // returning the maximum value of knapsack

}

int main()

{

int val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout << knapSack(W, wt, val, n);

return 0;

}

**Output**

220

**Complexity Analysis**:

**Time Complexity**: O(N\*W). As redundant calculations of states are avoided.

**Auxiliary Space**: O(W) As we are using 1-D array instead of 2-D array.

# 381. Binomial Coefficient Problem

Given two integers n and r, find nCr.Since the answer may be very large, calculate the answer modulo 109+7.

**Example 1:**

**Input:** n = 3, r = 2

**Output:** 3

**Explaination:** 3C2 = 3.

**Example 2:**

**Input:** n = 2, r = 4

**Output:** 0

**Explaination:** r is greater than n.

**Your Task:**  
You do not need to take input or print anything. Your task is to complete the function **nCr()** which takes n and r as input parameters and returns nCrmodulo 109+7..

**Expected Time Complexity:** O(n\*r)  
**Expected Auxiliary Space:** O(r)

**Constraints:**  
1 ≤ n ≤ 1000  
1 ≤ r ≤ 800

## Solution:

**1) Optimal Substructure**   
The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1

Following is a simple recursive implementation that simply follows the recursive structure mentioned above.

// A naive recursive C++ implementation

#include <bits/stdc++.h>

using namespace std;

// Returns value of Binomial Coefficient C(n, k)

int binomialCoeff(int n, int k)

{

// Base Cases

if (k > n)

return 0;

if (k == 0 || k == n)

return 1;

// Recur

return binomialCoeff(n - 1, k - 1)

+ binomialCoeff(n - 1, k);

}

/\* Driver code\*/

int main()

{

int n = 5, k = 2;

cout << "Value of C(" << n << ", " << k << ") is "

<< binomialCoeff(n, k);

return 0;

}

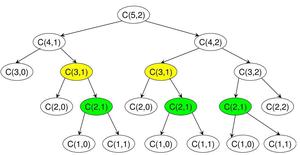
**Output**

Value of C(5, 2) is 10

***Time Complexity:***O(n\*max(k,n-k))

***Auxiliary Space:***O(n\*max(k,n-k))

**2) Overlapping Subproblems**   
It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems. 



*Binomial Coefficients Recursion tree for C(5,2)*

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), re-computations of the same subproblems can be avoided by constructing a temporary 2D-array C[][] in a bottom-up manner. Following is Dynamic Programming-based implementation.

// A Dynamic Programming based solution that uses

// table C[][] to calculate the Binomial Coefficient

#include <bits/stdc++.h>

using namespace std;

// Prototype of a utility function that

// returns minimum of two integers

int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)

int binomialCoeff(int n, int k)

{

int C[n + 1][k + 1];

int i, j;

// Calculate value of Binomial Coefficient

// in bottom up manner

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

for (j = 0; j <= min(i, k); j++) {

// Base Cases

if (j == 0 || j == i)

C[i][j] = 1;

// Calculate value using previously

// stored values

else

C[i][j] = C[i - 1][j - 1] + C[i - 1][j];

}

}

return C[n][k];

}

// A utility function to return

// minimum of two integers

int min(int a, int b) { return (a < b) ? a : b; }

// Driver Code

int main()

{

int n = 5, k = 2;

cout << "Value of C[" << n << "][" << k << "] is "

<< binomialCoeff(n, k);

}

**Output**

Value of C[5][2] is 10

**Time Complexity:** O(n\*k)   
**Auxiliary Space:** O(n\*k)

Following is a space-optimized version of the above code. The following code only uses O(k).

// C++ program for space optimized Dynamic Programming

// Solution of Binomial Coefficient

#include <bits/stdc++.h>

using namespace std;

int binomialCoeff(int n, int k)

{

int C[k + 1];

memset(C, 0, sizeof(C));

C[0] = 1; // nC0 is 1

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

{

// Compute next row of pascal triangle using

// the previous row

for (int j = min(i, k); j > 0; j--)

C[j] = C[j] + C[j - 1];

}

return C[k];

}

/\* Driver code\*/

int main()

{

int n = 5, k = 2;

cout << "Value of C(" << n << "," << k << ")"<< "is " <<binomialCoeff(n, k);

return 0;

}

**Output**

Value of C(5, 2) is 10

**Time Complexity:** O(n\*k)   
**Auxiliary Space:** O(k)

**Explanation:**   
1==========>> n = 0, C(0,0) = 1   
1–1========>> n = 1, C(1,0) = 1, C(1,1) = 1   
1–2–1======>> n = 2, C(2,0) = 1, C(2,1) = 2, C(2,2) = 1   
1–3–3–1====>> n = 3, C(3,0) = 1, C(3,1) = 3, C(3,2) = 3, C(3,3)=1   
1–4–6–4–1==>> n = 4, C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3)=4, C(4,4)=1   
So here every loop on i, builds i’th row of pascal triangle, using (i-1)th row  
At any time, every element of array C will have some value (ZERO or more) and in the next iteration, the value for those elements comes from the previous iteration.   
In statement,   
C[j] = C[j] + C[j-1]   
The right-hand side represents the value coming from the previous iteration (A row of Pascal’s triangle depends on the previous row). The left-Hand side represents the value of the current iteration which will be obtained by this statement.

Let's say we want to calculate C(4, 3),

i.e. n=4, k=3:

All elements of array C of size 4 (k+1) are

initialized to ZERO.

i.e. C[0] = C[1] = C[2] = C[3] = C[4] = 0;

Then C[0] is set to 1

For i = 1:

C[1] = C[1] + C[0] = 0 + 1 = 1 ==>> C(1,1) = 1

For i = 2:

C[2] = C[2] + C[1] = 0 + 1 = 1 ==>> C(2,2) = 1

C[1] = C[1] + C[0] = 1 + 1 = 2 ==>> C(2,1) = 2

For i=3:

C[3] = C[3] + C[2] = 0 + 1 = 1 ==>> C(3,3) = 1

C[2] = C[2] + C[1] = 1 + 2 = 3 ==>> C(3,2) = 3

C[1] = C[1] + C[0] = 2 + 1 = 3 ==>> C(3,1) = 3

For i=4:

C[4] = C[4] + C[3] = 0 + 1 = 1 ==>> C(4,4) = 1

C[3] = C[3] + C[2] = 1 + 3 = 4 ==>> C(4,3) = 4

C[2] = C[2] + C[1] = 3 + 3 = 6 ==>> C(4,2) = 6

C[1] = C[1] + C[0] = 3 + 1 = 4 ==>> C(4,1) = 4

C(4,3) = 4 is would be the answer in our example.

**Memoization Approach:**The idea is to create a lookup table and follow the recursive top-down approach. Before computing any value, we check if it is already in the lookup table. If yes, we return the value. Else we compute the value and store it in the lookup table. Following is the Top-down approach of dynamic programming to finding the value of the Binomial Coefficient.

// A Dynamic Programming based

// solution that uses

// table dp[][] to calculate

// the Binomial Coefficient

// A naive recursive approach

// with table C++ implementation

#include <bits/stdc++.h>

using namespace std;

// Returns value of Binomial Coefficient C(n, k)

int binomialCoeffUtil(int n, int k, int\*\* dp)

{

// If value in lookup table then return

if (dp[n][k] != -1) //

return dp[n][k];

// store value in a table before return

if (k == 0) {

dp[n][k] = 1;

return dp[n][k];

}

// store value in table before return

if (k == n) {

dp[n][k] = 1;

return dp[n][k];

}

// save value in lookup table before return

dp[n][k] = binomialCoeffUtil(n - 1, k - 1, dp) +

binomialCoeffUtil(n - 1, k, dp);

return dp[n][k];

}

int binomialCoeff(int n, int k)

{

int\*\* dp; // make a temporary lookup table

dp = new int\*[n + 1];

// loop to create table dynamically

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

dp[i] = new int[k + 1];

}

// nested loop to initialise the table with -1

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

for (int j = 0; j < (k + 1); j++) {

dp[i][j] = -1;

}

}

return binomialCoeffUtil(n, k, dp);

}

/\* Driver code\*/

int main()

{

int n = 5, k = 2;

cout << "Value of C(" << n << ", " << k << ") is "

<< binomialCoeff(n, k) << endl;

return 0;

}

***Time Complexity:***O(max(n,n-k))

***Auxiliary Space:***O(n\*k)

**Output**

Value of C(5, 2) is 10

**My Implementation:**

class Solution{

public:

int dp[1001][801];

void fun(){

memset(dp, 0, sizeof(dp));

for(int i=0;i<=1001;i++)

dp[i][0] = 1;

for(int i=1;i<1001;i++){

for(int j=1;j<801;j++){

dp[i][j] = (dp[i-1][j] + dp[i-1][j-1])%(int)(pow(10,9)+7);

}

}

}

int nCr(int n, int r){

// code here

static bool oneTime = false;

if(oneTime==false){

//cout<<"heyyy";

fun();

oneTime = true;

}

return dp[n][r];

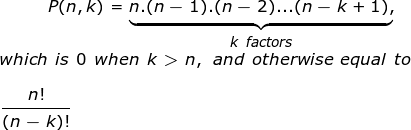
}

};

# 382. [Permutation Coefficient Problem](https://www.geeksforgeeks.org/permutation-coefficient/)

Permutation refers to the process of arranging all the members of a given set to form a sequence. The number of permutations on a set of n elements is given by n! , where “!” represents factorial.

The **Permutation Coefficient** represented by P(n, k) is used to represent the number of ways to obtain an ordered subset having k elements from a set of n elements.  
Mathematically it’s given as: 



**Examples :**

P(10, 2) = 90

P(10, 3) = 720

P(10, 0) = 1

P(10, 1) = 10

## Solution:

The coefficient can also be computed recursively using the below recursive formula:

P(n, k) = P(n-1, k) + k\* P(n-1, k-1)

If we observe closely, we can analyze that the problem has overlapping substructure, hence we can apply dynamic programming here. Below is a program implementing the same idea.

// A Dynamic Programming based

// solution that uses table P[][]

// to calculate the Permutation

// Coefficient

#include<bits/stdc++.h>

// Returns value of Permutation

// Coefficient P(n, k)

int permutationCoeff(int n, int k)

{

int P[n + 1][k + 1];

// Calculate value of Permutation

// Coefficient in bottom up manner

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

{

for (int j = 0; j <= std::min(i, k); j++)

{

// Base Cases

if (j == 0)

P[i][j] = 1;

// Calculate value using

// previously stored values

else

P[i][j] = P[i - 1][j] +

(j \* P[i - 1][j - 1]);

// This step is important

// as P(i,j)=0 for j>i

P[i][j + 1] = 0;

}

}

return P[n][k];

}

// Driver Code

int main()

{

int n = 10, k = 2;

printf("Value of P(%d, %d) is %d ",

n, k, permutationCoeff(n, k));

return 0;

}

**Output :**

Value of P(10, 2) is 90

Here as we can see the time complexity is O(n\*k) and space complexity is O(n\*k) as the program uses an auxiliary matrix to store the result.

**Can we do it in O(n) time ?**  
Let us suppose we maintain a single 1D array to compute the factorials up to n. We can use computed factorial value and apply the formula P(n, k) = n! / (n-k)!. Below is a program illustrating the same concept.

// A O(n) solution that uses

// table fact[] to calculate

// the Permutation Coefficient

#include<bits/stdc++.h>

using namespace std;

// Returns value of Permutation

// Coefficient P(n, k)

int permutationCoeff(int n, int k)

{

int fact[n + 1];

// Base case

fact[0] = 1;

// Calculate value

// factorials up to n

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

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

// P(n,k) = n! / (n - k)!

return fact[n] / fact[n - k];

}

// Driver Code

int main()

{

int n = 10, k = 2;

cout << "Value of P(" << n << ", "

<< k << ") is "

<< permutationCoeff(n, k);

return 0;

}

**Output :**

Value of P(10, 2) is 90

**A O(n) time and O(1) Extra Space Solution**

// A O(n) time and O(1) extra

// space solution to calculate

// the Permutation Coefficient

#include <iostream>

using namespace std;

int PermutationCoeff(int n, int k)

{

int P = 1;

// Compute n\*(n-1)\*(n-2)....(n-k+1)

for (int i = 0; i < k; i++)

P \*= (n-i) ;

return P;

}

// Driver Code

int main()

{

int n = 10, k = 2;

cout << "Value of P(" << n << ", " << k

<< ") is " << PermutationCoeff(n, k);

return 0;

}

**Output :**

Value of P(10, 2) is 90

# 383. Program for nth Catalan Number

Given an integer N, the task is to find the number of binary strings of size 2\*N in which each prefix of the string has more than or an equal number of 1's than 0's.

**Note:** The answer can be very large. So, output answer modulo 109+7

**Example 1:**

**Input**: N = 2

**Output:** 2

**Explanation**: 1100, 1010 are two

such strings of size 4

**Example 2:**

**Input**: N = 3

**Output:** 5

**Explanation**: 111000 101100 101010 110010

110100 are such 5 strings

**Your Task:**  
You don't need to read input or print anything. Complete the function **prefixStrings()**which takes **N**as input parameter and returns an integer value.  
  
**Expected Time Complexity:** O(**|N|**)  
**Expected Auxiliary Space:** O(**|N|**)  
  
**Constraints:**  
1 ≤ **|N|** ≤ 103

## Solution:

int prefixStrings(int N)

{

long long dp[N+1];

memset(dp, 0, sizeof dp);

int mod = (int)(1e9 + 7);

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

for (int i = 2; i <= N; i++)

{

dp[i] = 0;

for (int j = 0; j < i; j++)

dp[i] =(dp[i]%mod + (dp[j]%mod \* dp[i-j-1]%mod)%mod)%mod;

}

return (int)(dp[N]);

}

**Recursive Solution**   
Catalan numbers satisfy the following recursive formula.



Following is the implementation of above recursive formula.

#include <iostream>

using namespace std;

// A recursive function to find nth catalan number

unsigned long int catalan(unsigned int n)

{

// Base case

if (n <= 1)

return 1;

// catalan(n) is sum of

// catalan(i)\*catalan(n-i-1)

unsigned long int res = 0;

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

res += catalan(i)

\* catalan(n - i - 1);

return res;

}

// Driver code

int main()

{

for (int i = 0; i < 10; i++)

cout << catalan(i) << " ";

return 0;

}

**Output**

1 1 2 5 14 42 132 429 1430 4862

**Time complexity** of above implementation is equivalent to nth catalan number.



The value of nth catalan number is exponential that makes the time complexity exponential.

**Dynamic Programming Solution** : We can observe that the above recursive implementation does a lot of repeated work (we can the same by drawing recursion tree). Since there are overlapping subproblems, we can use dynamic programming for this. Following is a Dynamic programming based implementation .

#include <iostream>

using namespace std;

// A dynamic programming based function to find nth

// Catalan number

unsigned long int catalanDP(unsigned int n)

{

// Table to store results of subproblems

unsigned long int catalan[n + 1];

// Initialize first two values in table

catalan[0] = catalan[1] = 1;

// Fill entries in catalan[] using recursive formula

for (int i = 2; i <= n; i++) {

catalan[i] = 0;

for (int j = 0; j < i; j++)

catalan[i] += catalan[j] \* catalan[i - j - 1];

}

// Return last entry

return catalan[n];

}

// Driver code

int main()

{

for (int i = 0; i < 10; i++)

cout << catalanDP(i) << " ";

return 0;

}

**Output**

1 1 2 5 14 42 132 429 1430 4862

**Time Complexity:** Time complexity of above implementation is O(n2)

**Using Binomial Coefficient**  
We can also use the below formula to find nth Catalan number in O(n) time.

We have discussed a[O(n) approach to find binomial coefficient nCr](https://www.geeksforgeeks.org/space-and-time-efficient-binomial-coefficient/).

// C++ program for nth Catalan Number

#include <iostream>

using namespace std;

// Returns value of Binomial Coefficient C(n, k)

unsigned long int binomialCoeff(unsigned int n,

unsigned int k)

{

unsigned long int res = 1;

// Since C(n, k) = C(n, n-k)

if (k > n - k)

k = n - k;

// Calculate value of [n\*(n-1)\*---\*(n-k+1)] /

// [k\*(k-1)\*---\*1]

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

res \*= (n - i);

res /= (i + 1);

}

return res;

}

// A Binomial coefficient based function to find nth catalan

// number in O(n) time

unsigned long int catalan(unsigned int n)

{

// Calculate value of 2nCn

unsigned long int c = binomialCoeff(2 \* n, n);

// return 2nCn/(n+1)

return c / (n + 1);

}

// Driver code

int main()

{

for (int i = 0; i < 10; i++)

cout << catalan(i) << " ";

return 0;

}

**Output**

1 1 2 5 14 42 132 429 1430 4862

**Time Complexity:**Time complexity of above implementation is O(n).  
We can also use below formula to find nth catalan number in O(n) time.



# 384. Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the **ith** matrix has the dimensions **(arr[i-1] x arr[i])**.

**Example 1:**

**Input:** N = 5

arr = {40, 20, 30, 10, 30}

**Output:** 26000

**Explaination:** There are 4 matrices of dimension

40x20, 20x30, 30x10, 10x30. Say the matrices are

named as A, B, C, D. Out of all possible combinations,

the most efficient way is (A\*(B\*C))\*D.

The number of operations are -

20\*30\*10 + 40\*20\*10 + 40\*10\*30 = 26000.

**Example 2:**

**Input:** N = 4

arr = {10, 30, 5, 60}

**Output:** 4500

**Explaination:** The matrices have dimensions

10\*30, 30\*5, 5\*60. Say the matrices are A, B

and C. Out of all possible combinations,the

most efficient way is (A\*B)\*C. The

number of multiplications are -

10\*30\*5 + 10\*5\*60 = 4500.

**Your Task:**  
You do not need to take input or print anything. Your task is to complete the function **matrixMultiplication()** which takes the value **N** and the array **arr[]** as input parameters and returns the minimum number of multiplication operations needed to be performed.

**Expected Time Complexity:** O(N3)  
**Expected Auxiliary Space:** O(N2)

**Constraints:**   
2 ≤ N ≤ 100  
1 ≤ arr[i] ≤ 500

## Solution:

**1) Optimal Substructure:**   
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.  
Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

**2) Overlapping Subproblems**   
Following is a recursive implementation that simply follows the above optimal substructure property.

Below is the implementation of the above idea:

/\* A naive recursive implementation that simply

follows the above optimal substructure property \*/

#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 min = 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];

if (count < min)

min = count;

}

// Return minimum count

return min;

}

// Driver Code

int main()

{

int arr[] = { 1, 2, 3, 4, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Minimum number of multiplications is "

<< MatrixChainOrder(arr, 1, n - 1);

}

**Output**

Minimum number of multiplications is 30

The time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.

https://media.geeksforgeeks.org/wp-content/uploads/matrixchainmultiplication.png

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Matrix Chain Multiplication problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

**Dynamic Programming Solution**   
Following is the implementation of the Matrix Chain Multiplication problem using Dynamic Programming [(Tabulation vs Memoization)](https://www.geeksforgeeks.org/tabulation-vs-memoization/)

**Using Memoization –**

// 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);

}

**Output**

Minimum number of multiplications is 18

**Time Complexity:** O(n3 )

**Auxiliary Space:** O(n2) ignoring recursion stack space

**Using Tabulation –**

// 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;

}

**Output**

Minimum number of multiplications is 18

**Time Complexity:** O(n3 )  
**Auxiliary Space:**O(n2)

# 385. Edit Distance

## Same as ques 60 of string.

# 386. Subset Sum Problem

## Same as ques 261 of Backtracking.

# 387. Friends Pairing Problem

Given N friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.  
Note: Since answer can be very large, return your answer mod 10^9+7.

**Example 1:**

**Input:**N = 3

**Output:** 4

**Explanation**:

{1}, {2}, {3} : All single

{1}, {2,3} : 2 and 3 paired but 1 is single.

{1,2}, {3} : 1 and 2 are paired but 3 is single.

{1,3}, {2} : 1 and 3 are paired but 2 is single.

Note that {1,2} and {2,1} are considered same.

**Example 2:**

**Input**: N = 2

**Output:** 2

**Explanation**:

{1} , {2} : All single.

{1,2} : 1 and 2 are paired.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **countFriendsPairings()**which accepts an integer n and return number of ways in which friends can remain single or can be paired up.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ N ≤ 104

## Solution:

#define m 1000000007

class Solution

{

public:

// Returns count of ways n people

// can remain single or paired up.

int countFriendsPairings(int n)

{

long long a = 1, b = 2, c = 0;

if (n <= 2) {

return n;

}

for (int i = 3; i <= n; i++)

{

// using modular arithmentic properties.

c = ( b%m + ( ( (i - 1)%m \* a%m )%m ) %m )%m ;

a = b;

b = c;

}

return c;

}

};

**Time Complexity:**O(N)

**Auxiliary Space:**O(1)

For n-th person there are two choices:  
1) n-th person remains single, we recur  
for f(n – 1)  
2) n-th person pairs up with any of the  
remaining n – 1 persons. We get (n – 1) \* f(n – 2)

Therefore we can recursively write f(n) as:  
f(n) = f(n – 1) + (n – 1) \* f(n – 2)

Since the above recursive formula has [overlapping subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/), we can solve it using Dynamic Programming.

// C++ program for solution of

// friends pairing problem

#include <bits/stdc++.h>

using namespace std;

// Returns count of ways n people

// can remain single or paired up.

int countFriendsPairings(int n)

{

int dp[n + 1];

// Filling dp[] in bottom-up manner using

// recursive formula explained above.

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

if (i <= 2)

dp[i] = i;

else

dp[i] = dp[i - 1] + (i - 1) \* dp[i - 2];

}

return dp[n];

}

// Driver code

int main()

{

int n = 4;

cout << countFriendsPairings(n) << endl;

return 0;

}

**Output:**

10

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

**Another approach:**(Using recursion)

// C++ program for solution of friends

// pairing problem Using Recursion

#include <bits/stdc++.h>

using namespace std;

int dp[1000];

// Returns count of ways n people

// can remain single or paired up.

int countFriendsPairings(int n)

{

if (dp[n] != -1)

return dp[n];

if (n > 2)

return dp[n] = countFriendsPairings(n - 1) + (n - 1) \* countFriendsPairings(n - 2);

else

return dp[n] = n;

}

// Driver code

int main()

{

memset(dp, -1, sizeof(dp));

int n = 4;

cout << countFriendsPairings(n) << endl;

}

**Output:**

10

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

Since the above formula is similar to [fibonacci number](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/), we can optimize the space with an iterative solution.

#include <bits/stdc++.h>

using namespace std;

// Returns count of ways n people

// can remain single or paired up.

int countFriendsPairings(int n)

{

int a = 1, b = 2, c = 0;

if (n <= 2) {

return n;

}

for (int i = 3; i <= n; i++) {

c = b + (i - 1) \* a;

a = b;

b = c;

}

return c;

}

// Driver code

int main()

{

int n = 4;

cout << countFriendsPairings(n);

return 0;

}

**Output:**

10

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

# 388. Gold Mine Problem

Given a gold mine called **M** of (**n x m)** dimensions. Each field in this mine contains a positive integer which is the amount of gold in tons. Initially the miner can start from any row in the first column. From a given cell, the miner can move

1. to the cell diagonally up towards the right
2. to the right
3. to the cell diagonally down towards the right

Find out maximum amount of gold which he can collect.

**Example 1:**

**Input:** n = 3, m = 3

M = {{1, 3, 3},

{2, 1, 4},

{0, 6, 4}};

**Output:** 12

**Explaination:**

The path is {(1,0) -> (2,1) -> (2,2)}.

**Example 2:**

**Input:** n = 4, m = 4

M = {{1, 3, 1, 5},

{2, 2, 4, 1},

{5, 0, 2, 3},

{0, 6, 1, 2}};

**Output:** 16

**Explaination:**

The path is {(2,0) -> (3,1) -> (2,2)

-> (2,3)} or {(2,0) -> (1,1) -> (1,2)

-> (0,3)}.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **maxGold()** which takes the values n, m and the mine M as input parameters and returns the maximum amount of gold that can be collected.

**Expected Time Complexity:** O(n\*m)  
**Expected Auxiliary Space:** O(n\*m)

**Constraints:**  
1 ≤ n, m ≤ 50  
1 ≤ M[i][j] ≤ 100

## Solution:

**Method 1:** Recursion

A simple method that is a direct recursive implementation

// C++ program to solve Gold Mine problem

#include<bits/stdc++.h>

using namespace std;

int collectGold(vector<vector<int>> gold, int r, int c, int n, int m) {

// Base condition.

if ((r < 0) || (r == n) || (c == m)) {

return 0;

}

// Right upper diagonal

int rightUpperDiagonal = collectGold(gold, r - 1, c + 1, n, m);

// right

int right = collectGold(gold, r, c + 1, n, m);

// Lower right diagonal

int rightLowerDiagonal = collectGold(gold, r + 1, c + 1, n, m);

// Return the maximum and store the value

return gold[r] + max(max(rightUpperDiagonal, rightLowerDiagonal), right);

}

int getMaxGold(vector<vector<int>> gold, int n, int m)

{

int maxGold = 0;

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

// Recursive function call for ith row.

int goldCollected = collectGold(gold, i, 0, n, m);

maxGold = max(maxGold, goldCollected);

}

return maxGold;

}

// Driver Code

int main()

{

vector<vector<int>> gold { {1, 3, 1, 5},

{2, 2, 4, 1},

{5, 0, 2, 3},

{0, 6, 1, 2}

};

int m = 4, n = 4;

cout << getMaxGold(gold, n, m);

return 0;

}

**Output**

16

**Time complexity**: O(3N\*M)

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

**Method 2:**Memoization

Bottom-Up Approach: The second way is to take an extra space of size m\*n and start computing values of states

of right, right upper diagonal, and right bottom diagonal and store it in the 2d array.

// C++ program to solve Gold Mine problem

#include<bits/stdc++.h>

using namespace std;

int collectGold(vector<vector<int>> gold, int r, int c, int n, int m, vector<vector<int>> &dp) {

// Base condition.

if ((r < 0) || (r == n) || (c == m)) {

return 0;

}

if(dp[r] != -1){

return dp[r] ;

}

// Right upper diagonal

int rightUpperDiagonal = collectGold(gold, r - 1, c + 1, n, m, dp);

// right

int right = collectGold(gold, r, c + 1, n, m, dp);

// Lower right diagonal

int rightLowerDiagonal = collectGold(gold, r + 1, c + 1, n, m, dp);

// Return the maximum and store the value

return dp[r] = gold[r] + max(max(rightUpperDiagonal, rightLowerDiagonal), right);

}

int getMaxGold(vector<vector<int>> gold, int n, int m)

{

int maxGold = 0;

// Initialize the dp vector

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

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

// Recursive function call for ith row.

int goldCollected = collectGold(gold, i, 0, n, m, dp);

maxGold = max(maxGold, goldCollected);

}

return maxGold;

}

// Driver Code

int main()

{

vector<vector<int>> gold { {1, 3, 1, 5},

{2, 2, 4, 1},

{5, 0, 2, 3},

{0, 6, 1, 2}

};

int m = 4, n = 4;

cout << getMaxGold(gold, n, m);

return 0;

}

**Output**

16

**Time Complexity :**O(m\*n)

**Space Complexity :**O(m\*n)

**Method 3**: Using Dp, Tabulation  
Create a 2-D matrix goldTable[][]) of the same as given matrix mat[][]. If we observe the question closely, we can notice following. 

1. Amount of gold is positive, so we would like to cover maximum cells of maximum values under given constraints.
2. In every move, we move one step toward right side. So we always end up in last column. If we are at the last column, then we are unable to move right

If we are at the first row or last column, then we are unable to move right-up so just assign 0 otherwise assign the value of goldTable[row-1][col+1] to right\_up. If we are at the last row or last column, then we are unable to move right down so just assign 0 otherwise assign the value of goldTable[row+1][col+1] to right up.   
Now find the maximum of right, right\_up, and right\_down and then add it with that mat[row][col]. At last, find the maximum of all rows and first column and return it.

// C++ program to solve Gold Mine problem

#include<bits/stdc++.h>

using namespace std;

const int MAX = 100;

// Returns maximum amount of gold that can be collected

// when journey started from first column and moves

// allowed are right, right-up and right-down

int getMaxGold(int gold[][MAX], int m, int n)

{

// Create a table for storing intermediate results

// and initialize all cells to 0. The first row of

// goldMineTable gives the maximum gold that the miner

// can collect when starts that row

int goldTable[m][n];

memset(goldTable, 0, sizeof(goldTable));

for (int col=n-1; col>=0; col--)

{

for (int row=0; row<m; row++)

{

// Gold collected on going to the cell on the right(->)

int right = (col==n-1)? 0: goldTable[row][col+1];

// Gold collected on going to the cell to right up (/)

int right\_up = (row==0 || col==n-1)? 0:

goldTable[row-1][col+1];

// Gold collected on going to the cell to right down (\)

int right\_down = (row==m-1 || col==n-1)? 0:

goldTable[row+1][col+1];

// Max gold collected from taking either of the

// above 3 paths

goldTable[row][col] = gold[row][col] +

max(right, max(right\_up, right\_down));

}

}

// The max amount of gold collected will be the max

// value in first column of all rows

int res = goldTable[0][0];

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

res = max(res, goldTable[i][0]);

return res;

}

// Driver Code

int main()

{

int gold[MAX][MAX]= { {1, 3, 1, 5},

{2, 2, 4, 1},

{5, 0, 2, 3},

{0, 6, 1, 2}

};

int m = 4, n = 4;

cout << getMaxGold(gold, m, n);

return 0;

}

**Output**

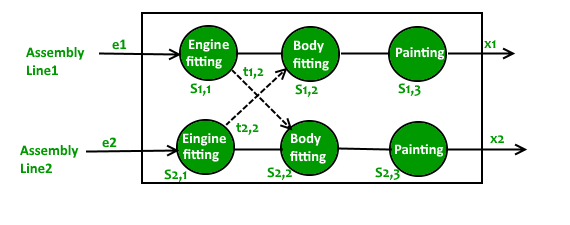
16

**Time Complexity :**O(m\*n)   
**Space Complexity :**O(m\*n)

# 389. Assembly Line Scheduling Problem

A car factory has two assembly lines, each with n stations. A station is denoted by Si,j where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by ai,j. Each station is dedicated to some sort of work like engine fitting, body fitting, painting, and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station Si,j, it will continue to station Si,j+1 unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time ti,j. Each assembly line takes an entry time ei and exit time xi which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture: 



The following information can be extracted from the problem statement to make it simpler:

* Two assembly lines, 1 and 2, each with stations from 1 to n.
* A car chassis must pass through all stations from 1 to n in order(in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.
* The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost ti, j to move to station j from line i. No cost is incurred for movement in same line.
* The time taken in station j on line i is ai, j.
* Si, j represents a station j on line i.

## Solution:

**Breaking the problem into smaller sub-problems:**   
We can easily find the ith factorial if (i-1)th factorial is known. Can we apply the similar funda here?   
If the minimum time taken by the chassis to leave station Si, j-1 is known, the minimum time taken to leave station Si, j can be calculated quickly by combining ai, j and ti, j.  
**T1(j)** indicates the minimum time taken by the car chassis to leave station j on assembly line 1.  
**T2(j)** indicates the minimum time taken by the car chassis to leave station j on assembly line 2.

***Base cases:***  
The entry time ei comes into picture only when the car chassis enters the car factory.  
Time taken to leave the first station in line 1 is given by:   
T1(1) = Entry time in Line 1 + Time spent in station S1,1   
T1(1) = e1 + a1,1   
Similarly, time taken to leave the first station in line 2 is given by:   
T2(1) = e2 + a2,1

***Recursive Relations:***   
If we look at the problem statement, it quickly boils down to the below observations:   
The car chassis at station S1,j can come either from station S1, j-1 or station S2, j-1.

Case #1: Its previous station is S1, j-1   
The minimum time to leave station S1,j is given by:   
T1(j) = Minimum time taken to leave station S1, j-1 + Time spent in station S1, j   
T1(j) = T1(j-1) + a1, j

Case #2: Its previous station is S2, j-1   
The minimum time to leave station S1, j is given by:   
T1(j) = Minimum time taken to leave station S2, j-1 + Extra cost incurred to change the assembly line + Time spent in station S1, j   
T1(j) = T2(j-1) + t2, j + a1, j

The minimum time T1(j) is given by the minimum of the two obtained in cases #1 and #2.   
T1(j) = min((T1(j-1) + a1, j), (T2(j-1) + t2, j + a1, j))

Similarly, the minimum time to reach station S2, j is given by:   
T2(j) = min((T2(j-1) + a2, j), (T1(j-1) + t1, j + a2, j))

The total minimum time taken by the car chassis to come out of the factory is given by:   
Tmin = min(Time taken to leave station Si,n + Time taken to exit the car factory)   
Tmin = min(T1(n) + x1, T2(n) + x2)

**Why dynamic programming?**   
The above recursion exhibits overlapping sub-problems. There are two ways to reach station S1, j:

1. From station S1, j-1
2. From station S2, j-1

So, to find the minimum time to leave station S1, j the minimum time to leave the previous two stations must be calculated(as explained in above recursion).

Similarly, there are two ways to reach station S2, j:

1. From station S2, j-1
2. From station S1, j-1

Please note that the minimum times to leave stations S1, j-1 and S2, j-1 have already been calculated.  
So, we need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in a bottom-up fashion.

**Note:**   
In this post, the word “leave” has been used in place of “reach” to avoid confusion. Since the car chassis must spend a fixed time in each station, the word leave suits better.

**Implementation:**

// A C++ program to find minimum possible

// time by the car chassis to complete

#include <bits/stdc++.h>

using namespace std;

#define NUM\_LINE 2

#define NUM\_STATION 4

// Utility function to find a minimum of two numbers

int min(int a, int b)

{

return a < b ? a : b;

}

int carAssembly(int a[][NUM\_STATION],

int t[][NUM\_STATION],

int \*e, int \*x)

{

int T1[NUM\_STATION], T2[NUM\_STATION], i;

// time taken to leave first station in line 1

T1[0] = e[0] + a[0][0];

// time taken to leave first station in line 2

T2[0] = e[1] + a[1][0];

// Fill tables T1[] and T2[] using the

// above given recursive relations

for (i = 1; i < NUM\_STATION; ++i)

{

T1[i] = min(T1[i - 1] + a[0][i],

T2[i - 1] + t[1][i] + a[0][i]);

T2[i] = min(T2[i - 1] + a[1][i],

T1[i - 1] + t[0][i] + a[1][i]);

}

// Consider exit times and return minimum

return min(T1[NUM\_STATION - 1] + x[0],

T2[NUM\_STATION - 1] + x[1]);

}

// Driver Code

int main()

{

int a[][NUM\_STATION] = {{4, 5, 3, 2},

{2, 10, 1, 4}};

int t[][NUM\_STATION] = {{0, 7, 4, 5},

{0, 9, 2, 8}};

int e[] = {10, 12}, x[] = {18, 7};

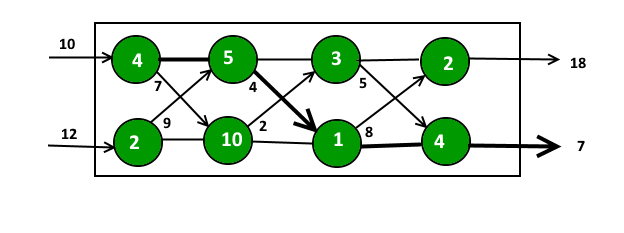
cout << carAssembly(a, t, e, x);

return 0;

}

**Output:**

35



The bold line shows the path covered by the car chassis for given input values. We need only the last two values in the auxiliary arrays. So instead of creating two arrays, we can use two variables.

// A space optimized solution for

// assembly line scheduling

#include <bits/stdc++.h>

using namespace std;

int carAssembly(int a[][4],

int t[][4],

int \*e, int \*x)

{

int first, second, i;

// Time taken to leave first

// station in line 1

first = e[0] + a[0][0];

// Time taken to leave first

// station in line 2

second = e[1] + a[1][0];

// Fill tables T1[] and T2[] using the

// above given recursive relations

for(i = 1; i < 4; ++i)

{

int up = min(first + a[0][i],

second + t[1][i] +

a[0][i]);

int down = min(second + a[1][i],

first + t[0][i] +

a[1][i]);

first = up;

second = down;

}

// Consider exit times and

// return minimum

return min(first + x[0],

second + x[1]);

}

// Driver Code

int main()

{

int a[][4] = { { 4, 5, 3, 2 },

{ 2, 10, 1, 4 } };

int t[][4] = { { 0, 7, 4, 5 },

{ 0, 9, 2, 8 } };

int e[] = { 10, 12 }, x[] = { 18, 7 };

cout << carAssembly(a, t, e, x);

return 0;

}

**Output:**

35

# 390. Painting the Fence Problem

Given a fence with n posts and k colors, find out the number of ways of painting the fence so that not more than two consecutive fences have the same colors. Since the answer can be large return it modulo 10^9 + 7.

**Example 1:**

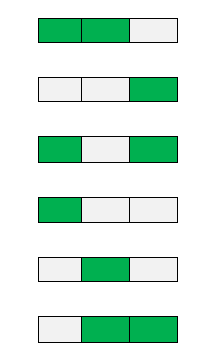
**Input:**

N=3, K=2

**Output:** 6

**Explanation**:

We have following possible combinations:

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/paintFence.png)

**Example 2:**

**Input:**

N=2, K=4

**Output:** 16

**Your Task:**  
Since, this is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function **countWays**() that takes **n and k** as parameters and returns the number of ways in which the fence can be painted.(modulo 109 + 7)

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(N).

**Constraints:**  
1 ≤ N ≤ 5000  
1 ≤ K ≤ 100

## Solution:

**Realizing This is a Dynamic Programming Problem**

There are two parts to this problem that tell us it can be solved with dynamic programming.

First, the question is asking for the "number of ways" to do something.

Second, we need to make decisions that may depend on previously made decisions. In this problem, we need to decide what color we should paint a given post, which may change depending on previous decisions. For example, if we paint the first two posts the same color, then we are not allowed to paint the third post the same color.

Both of these things are characteristic of dynamic programming problems.

**A Framework to Solve Dynamic Programming Problems**

A dynamic programming algorithm typically has 3 components. Learning these components is extremely valuable, as **most dynamic programming problems can be solved this way**.

First, we need some function or array that represents the answer to the problem for a given state. For this problem, let's say that we have a function totalWays, where totalWays(i) returns the number of ways to paint i posts. Because we only have one argument, this is a one-dimensional dynamic programming problem.

Second, we need a way to transition between states, such as totalWays(3) and totalWays(4). This is called a **recurrence relation** and figuring it out is usually the hardest part of solving a problem with dynamic programming. We'll talk about the recurrence relation for this problem below.

The third component is establishing base cases. If we have one post, there are k ways to paint it. If we have two posts, then there are k \* k ways to paint it (since we are allowed to paint have two posts in a row be the same color). Therefore, totalWays(1) = k, totalWays(2) = k \* k.

**Finding The Recurrence Relation**

We know the values for totalWays(1) and totalWays(2), now we need a formula for totalWays(i), where 3 <= i <= n. Let's think about how many ways there are to paint the i^{th}*ith* post. We have two options:

1. Use a different color than the previous post. If we use a different color, then there are k - 1 colors for us to use. This means there are (k - 1) \* totalWays(i - 1) ways to paint the i^{th}*ith* post a different color than the (i - 1)^{th}(*i*−1)*th* post.
2. Use the same color as the previous post. There is only one color for us to use, so there are 1 \* totalWays(i - 1) ways to paint the i^{th}*ith* post the same color as the (i - 1)^{th}(*i*−1)*th* post. However, we have the added restriction of not being allowed to paint three posts in a row the same color. Therefore, we can paint the i^{th}*ith* post the same color as the (i - 1)^{th}(*i*−1)*th* post **only if** the (i - 1)^{th}(*i*−1)*th* post is a different color than the (i - 2)^{th}(*i*−2)*th* post.

So, how many ways are there to paint the (i - 1)^{th}(*i*−1)*th* post a different color than the (i - 2)^{th}(*i*−2)*th* post? Well, as stated in the first option, there are (k - 1) \* totalWays(i - 1) ways to paint the i^{th}*ith* post a different color than the (i - 1)^{th}(*i*−1)*th* post, so that means there are 1 \* (k - 1) \* totalWays(i - 2) ways to paint the (i - 1)^{th}(*i*−1)*th* post a different color than the (i - 2)^{th}(*i*−2)*th* post.

Adding these two scenarios together gives totalWays(i) = (k - 1) \* totalWays(i - 1) + (k - 1) \* totalWays(i - 2), which can be simplified to:

totalWays(i) = (k - 1) \* (totalWays(i - 1) + totalWays(i - 2))

This is our recurrence relation which we can use to solve the problem from the base cases.

Below is the implementation of the problem:

// C++ program for Painting Fence Algorithm

// optimised version

#include <bits/stdc++.h>

using namespace std;

// Returns count of ways to color k posts

long countWays(int n, int k)

{

long dp[n + 1];

memset(dp, 0, sizeof(dp));

long long mod = 1000000007;

dp[1] = k;

dp[2] = k \* k;

for (int i = 3; i <= n; i++) {

dp[i] = ((k - 1) \* (dp[i - 1] + dp[i - 2])) % mod;

}

return dp[n];

}

// Driver code

int main()

{

int n = 3, k = 2;

cout << countWays(n, k) << endl;

return 0;

}

**Output:**

6

**Space optimization :**  
We can optimize the above solution to use one variable instead of a table.  
Below is the implementation of the problem:

long long countWays(int n, int k){

long long a, b, c;

a = k;

b = k\*k;

if(n==1)

return a;

if(n==2)

return b;

for(int i=2;i<n;i++){

c = ((k-1)\*((b+a)%(int)(1e9+7)))%(int)(1e9+7);

a = b;

b = c;

}

return c;

}

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

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

# 391. Maximize The Cut Segments

Given an integer **N** denoting the Length of a line segment. You need to cut the line segment in such a way that the cut length of a line segment each time is either **x** , **y** or **z**. Here x, y, and z are integers.  
After performing all the cut operations, your**total number of cut segments must be maximum**.

**Example 1:**

**Input:**

N = 4

x = 2, y = 1, z = 1

**Output:** 4

**Explanation:**Total length is 4, and the cut

lengths are 2, 1 and 1.  We can make

maximum 4 segments each of length 1.

**Example 2:**

**Input:**

N = 5

x = 5, y = 3, z = 2

**Output:** 2

**Explanation:** Here total length is 5, and

the cut lengths are 5, 3 and 2. We can

make two segments of lengths 3 and 2.

**Your Task:**  
You only need to complete the function **maximizeTheCuts()**that takes n, x, y, z as parameters and returns **max number cuts**.

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

**Constraints**  
1 <= N, x, y, z <= 104

## Solution:

**Approach 1:**

This can be visualized as a classical recursion problem , which further narrows down to **memoization** ( top-down ) method of **Dynamic Programming**. Initially , we have length *l* present with us , we’d have three size choices to cut from this , either we can make a cut of length *p* , or*q* , or *r*. Let’s say we made a cut of length*p* , so the remaining length would be *l-p* and similarly with cuts *q* & *r* resulting in remaining lengths*l-q* & *l-r* respectively. We will call recursive function for the remaining lengths and at any subsequent instance we’ll have these three choices. We will store the answer from all these recursive calls & take the maximum out of them +1 as at any instance we’ll have 1 cut from this particular call as well. Also , note that the recursive call would be made if and only if the available length is greater than length we want to cut i.e. suppose *p=3* , and after certain recursive calls the available length is 2 only , so we can’t cut this line in lengths of*p* anymore.

Below is the **pseudocode** for the same:

if(l==0) // Base Case

return 0;

int a,b,c;

if(p<=l)

a=func(l-p,p,q,r);

if(q<=l)

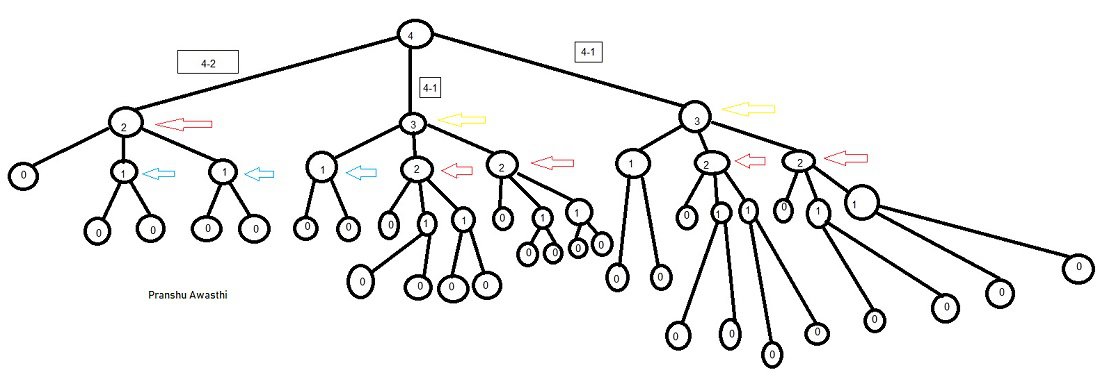
b=func(l-q,p,q,r);

if(r<=l)

c=func(l-r,p,q,r);

return 1+max({a,b,c});

Below is the recursion tree for *l=4,p=2,q=1 and r=1*:



*Recursion Tree for l=4 , p=2 ,q=1 and r=1*

One can clearly observe that at each call , the given length ( 4 initially ) is divided into 3 different subparts. Also , we can see that the recursion is being repeated for certain entries ( *Red* arrow represents repetitive call for *l=2, Yellow* for *l=3* and *Blue* for *l=1).*Therefore , we can **memoize** the results in any container or array , so that repetition of same recursive calls is avoided.

Now , the above **pseudocode** changes to :

vector<int> dp(10005,-1); // Initialise DP Table ( Array can also be used )

if(l==0) // Base Case

return 0;

if(dp[l]!=-1) // If already memoized , return from here only

return dp[l];

int a,b,c;

if(p<=l)

a=func(l-p,p,q,r);

if(q<=l)

b=func(l-q,p,q,r);

if(r<=l)

c=func(l-r,p,q,r);

return dp[l]=1+max({a,b,c}); // Memoize the result in the dp table & return

**Let’s now follow the code for implementation of the above code :**

//C++ Code to find maximum number of cut segments

// Memoization DP

#include <bits/stdc++.h>

using namespace std;

//Function to find the maximum number of cuts.

int dp[10005];

int func(int l, int p, int q, int r)

{

if(l==0)

return 0; // Base Case

if(dp[l]!=-1) // If already memoized

return dp[l];

int a(INT\_MIN),b(INT\_MIN),c(INT\_MIN); // Intitialise a,b,& c with INT\_MIN

if(p<=l) // If Possible to make a cut of length p

a=func(l-p,p,q,r);

if(q<=l) // If possible to make a cut of length q

b=func(l-q,p,q,r);

if(r<=l) // If possible to make a cut of length r

c=func(l-r,p,q,r);

return dp[l]=1+max({a,b,c}); // Memoize & return

}

int maximizeTheCuts(int l, int p, int q, int r)

{

memset(dp,-1,sizeof(dp)); // Set Lookup table to -1

int ans=func(l,p,q,r); // Utility function call

if(ans<0)

return 0; // If returned answer is negative , that means cuts are not possible

return ans;

}

int main()

{

int l,p,q,r;

cout<<"ENTER THE LENGTH OF THE ROD "<<endl;

cin>>l;

cout<<"ENTER THE VALUES OF p,q & r "<<endl;

cin>>p>>q>>r;

cout<<"THE MAXIMUM NUMBER OF SEGMENTS THAT CAN BE CUT OF LENGTH p,q & r FROM A ROD OF LENGTH l are "<<maximizeTheCuts(l,p,q,r)<<endl;

return 0;

}

**Time Complexity** : *O(n)* where n is the length of rod or line segment that has to be cut.

**Space Complexity** : *O(n)* where n is the length of rod or line segment that has to be cut.

**Approach 2:**

As the solution for a maximum number of cuts that can be made in a given length depends on the maximum number of cuts previously made in shorter lengths, this question could be solved by the approach of Dynamic Programming. Suppose we are given a **length ‘l’**. For finding the maximum number of cuts that can be made in **length ‘l’**, find the number of cuts made in shorter previous **length ‘l-p’,** **‘l-q’,** **‘l-r’** lengths respectively. The required answer would be the **max(l-p,l-q,l-r)+1** as one more cut should be needed after this to cut **length ‘l’**. So for solving this problem for a given length, find the maximum number of cuts that can be made in lengths **ranging from ‘1’ to ‘l’**.

**Example:**

*l = 11, p = 2, q = 3, r = 5   
Analysing lengths from 1 to 11:*

1. *Not possible to cut->0*
2. *Possible cut is of lengths 2->1 (2)*
3. *Possible cut is of lengths 3->1 (3)*
4. *Possible cuts are of lengths max(arr[4-2],arr[4-3])+1->2 (2,2)*
5. *Possible cuts are of lengths max(arr[5-2],arr[5-3])+1->2 (2,3)*
6. *Possible cuts are of lengths max(arr[6-2],arr[6-3],arr[6-5])+1->3 (2,2,2)*
7. *Possible cuts are of lengths max(arr[7-2],arr[7-3],arr[7-5])+1->3 (2,3,2) or (2,2,3)*
8. *Possible cuts are of lengths max(arr[8-2],arr[8-3],arr[8-5])+1->4 (2,2,2,2)*
9. *Possible cuts are of lengths max(arr[9-2],arr[9-3],arr[9-5])+1->4 (2,3,2,2) or (2,2,3,2) or (2,2,2,3)*
10. *Possible cuts are of lengths max(arr[10-2],arr[10-3],arr[10-5])+1->5 (2,2,2,2,2)*
11. *Possible cuts are of lengths max(arr[11-2],arr[11-3],arr[11-5])+1->5 (2,3,2,2,2) or (2,2,3,2,2) or (2,2,2,3,2) or (2,2,2,2,3)*

**Algorithm:**

1. Initialise an **array DP[]={-1} and DP[0]=0**.
2. Run a loop from ‘1’ to ‘l’
3. If DP[i]=-1 means it’s not possible to divide it using giving segments p,q,r so continue;
4. DP[i+p]=max(DP[i+p],DP[i]+1)
5. DP[i+q]=max(DP[i+q],DP[i]+1)
6. DP[i+r]=max(DP[i+r],DP[i]+1)
7. print DP[l]

**Pseudo Code:**

DP[l+1]={-1}

DP[0]=0

for(i from 0 to l)

if(DP[i]==-1)

continue

DP[i+p]=max(DP[i+p],DP[i]+1)

DP[i+q]=max(DP[i+q],DP[i]+1)

DP[i+r]=max(DP[i+r],DP[i]+1)

print(DP[l])

**Implementation:**

// C++ program to maximize the number

// of segments of length p, q and r

#include <bits/stdc++.h>

using namespace std;

// Function that returns the maximum number

// of segments possible

int findMaximum(int l, int p, int q, int r)

{

// Array to store the cut at each length

int dp[l + 1];

// All values with -1

memset(dp, -1, sizeof(dp));

// if length of rod is 0 then total cuts will be 0

// so, initialize the dp[0] with 0

dp[0] = 0;

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

// if certain length is not possible

if (dp[i] == -1)

continue;

// if a segment of p is possible

if (i + p <= l)

dp[i + p] = max(dp[i + p], dp[i] + 1);

// if a segment of q is possible

if (i + q <= l)

dp[i + q] = max(dp[i + q], dp[i] + 1);

// if a segment of r is possible

if (i + r <= l)

dp[i + r] = max(dp[i + r], dp[i] + 1);

}

// if no segment can be cut then return 0

if (dp[l] == -1) {

dp[l] = 0;

}

// return value corresponding to length l

return dp[l];

}

// Driver Code

int main()

{

int l = 11, p = 2, q = 3, r = 5;

// Calling Function

int ans = findMaximum(l, p, q, r);

cout << ans;

return 0;

}

**Output**

5

**Complexity Analysis:**

* **Time Complexity:** O(N).   
  Use of a single for-loop till length ‘N’.
* **Auxiliary Space:** O(N).   
  Use of an array ‘DP’ to keep track of segments

**Note:**This problem can also be thought of as a minimum coin change problem because we are given a certain length to acquire which is the same as the value of the amount whose minimum change is needed. Now the x,y,z are the same as the denomination of the coin given. So length is the same as the amount and x y z are the same as denominations, thus we need to change only one condition that is instead of finding minimum we need to find the maximum and we will get the answer. As the minimum coin change problem is the basic dynamic programming question so this will help to solve this question also.

The condition we need to change in minimum coin change problem

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

{

for(ll j=1;j<=3;j++)

{

if(i>=a[j]&&m[i-a[j]]!=-1)

{

dp[i]=max(dp[i],1+dp[i-a[j]]);

}

}

}

# 392. Longest Common Subsequence

## Same as ques 77 of String.

# 393. Longest Repeated Subsequence

## Same as ques 55 of String.

# 394. Longest Increasing Subsequence

Given an array of integers, find the **length**of the**longest (strictly) increasing subsequence** from the given array.

**Example 1:**

**Input:**

N = 16

A[]={0,8,4,12,2,10,6,14,1,9,5

  13,3,11,7,15}

**Output:** 6

**Explanation:**Longest increasing subsequence

0 2 6 9 13 15, which has length **6**

**Example 2:**

**Input:**

N = 6

A[] = {5,8,3,7,9,1}

**Output:** 3

**Explanation:**Longest increasing subsequence

5 7 9, with length **3**

**Your Task:**  
Complete the function **longestSubsequence()** which takes the input array and its size as input parameters and returns the **length**of the**longest increasing subsequence.**

**Expected Time Complexity** : O( N\*log(N) )  
**Expected Auxiliary Space**: O(N)

**Constraints:**  
1 ≤ N ≤ 105  
0 ≤ A[i] ≤ 106

## Solution:

**Method 1:** Recursion.  
***Optimal Substructure:*** Let arr[0..n-1] be the input array and 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.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.  
Formally, the length of the longest increasing subsequence ending at index i, will be 1 greater than the maximum of lengths of all longest increasing subsequences ending at indices before i, where arr[j] < arr[i] (j < i).  
Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

The recursive tree given below will make the approach clearer:

Input : arr[] = {3, 10, 2, 11}

**f(i): Denotes LIS of subarray ending at index 'i'**

(LIS(1)=1)

f(4) {f(4) = 1 + max(f(1), f(2), f(3))}

/ | \

f(1) f(2) f(3) {f(3) = 1, f(2) and f(1) are > f(3)}

| | \

f(1) f(2) f(1) {f(2) = 1 + max(f(1)}

|

f(1) {f(1) = 1}

Below is the implementation of the recursive approach:

/\* A Naive C++ recursive implementation

of LIS problem \*/

#include <iostream>

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]);

cout <<"Length of lis is "<< lis(arr, n);

return 0;

}

**Output:**

Length of lis is 5

**Complexity Analysis:**

* **Time Complexity:** 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 used for storing values apart from the internal stack space.

**Method 2:** Dynamic Programming.  
We can see that there are many subproblems in the above recursive solution which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

The simulation of approach will make things clear:

Input : arr[] = {3, 10, 2, 11}

LIS[] = {1, 1, 1, 1} (initially)

**Iteration-wise simulation :**

1. arr[2] > arr[1] {LIS[2] = max(LIS [2], LIS[1]+1)=2}
2. arr[3] < arr[1] {No change}
3. arr[3] < arr[2] {No change}
4. arr[4] > arr[1] {LIS[4] = max(LIS [4], LIS[1]+1)=2}
5. arr[4] > arr[2] {LIS[4] = max(LIS [4], LIS[2]+1)=3}
6. arr[4] > arr[3] {LIS[4] = max(LIS [4], LIS[3]+1)=3}

We can avoid recomputation of subproblems by using tabulation as shown in the below code:

Below is the implementation of the above approach:

/\* 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]);

printf("Length of lis is %d\n", lis(arr, n));

return 0;

}

**Output**

Length of lis is 5

**Complexity Analysis:**

* **Time Complexity:** O(n2).   
  As nested loop is used.
* **Auxiliary Space:** O(n).   
  Use of any array to store LIS values at each index.

**O(nlogn) solution:**

For the time being, forget about recursive and DP solutions. Let us take small samples and extend the solution to large instances. Even though it may look complex at first time, once if we understood the logic, coding is simple.  
Consider an input array A = {2, 5, 3}. I will extend the array during explanation.  
By observation we know that the LIS is either {2, 3} or {2, 5}. ***Note that I am considering only strictly increasing sequences***.  
Let us add two more elements, say 7, 11 to the array. These elements will extend the existing sequences. Now the increasing sequences are {2, 3, 7, 11} and {2, 5, 7, 11} for the input array {2, 5, 3, 7, 11}.  
Further, we add one more element, say 8 to the array i.e. input array becomes {2, 5, 3, 7, 11, 8}. Note that the latest element 8 is greater than smallest element of any active sequence (*will discuss shortly about active sequences*). How can we extend the existing sequences with 8? First of all, can 8 be part of LIS? If yes, how? If we want to add 8, it should come after 7 (by replacing 11).  
Since the approach is *offline (what we mean by*[*offline*](https://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/)*?)*, we are not sure whether adding 8 will extend the series or not. Assume there is 9 in the input array, say {2, 5, 3, 7, 11, 8, 7, 9 …}. We can replace 11 with 8, as there is potentially *best* candidate (9) that can extend the new series {2, 3, 7, 8} or {2, 5, 7, 8}.  
Our observation is, assume that the end element of largest sequence is E. We can add (replace) current element A[i] to the existing sequence if there is an element A[j] (j > i) such that E < A[i] < A[j] or (E > A[i] < A[j] – for replace). In the above example, E = 11, A[i] = 8 and A[j] = 9.  
In case of our original array {2, 5, 3}, note that we face same situation when we are adding 3 to increasing sequence {2, 5}. I just created two increasing sequences to make explanation simple. Instead of two sequences, 3 can replace 5 in the sequence {2, 5}.  
I know it will be confusing, I will clear it shortly!  
*The question is, when will it be safe to add or replace an element in the existing sequence?*  
Let us consider another sample A = {2, 5, 3}. Say, the next element is 1. How can it extend the current sequences {2, 3} or {2, 5}. Obviously, it can’t extend either. Yet, there is a potential that the new smallest element can be start of an LIS. To make it clear, consider the array is {2, 5, 3, 1, 2, 3, 4, 5, 6}. Making 1 as new sequence will create new sequence which is largest.  
*The observation is, when we encounter new smallest element in the array, it can be a potential candidate to start new sequence.*  
From the observations, we need to maintain lists of increasing sequences.  
In general, we have set of **active lists** of varying length. We are adding an element A[i] to these lists. We scan the lists (for end elements) in decreasing order of their length. We will verify the end elements of all the lists to find a list whose end element is smaller than A[i] (*floor* value).  
Our strategy determined by the following conditions, 

**1. If A[i] is smallest among all *end***

***candidates of active lists, we will start***

***new active list of length 1.***

**2. If A[i] is largest among all *end candidates of***

***active lists, we will clone the largest active***

***list, and extend it by A[i].***

**3. If A[i] is in between, we will find a list with**

***largest end element that is smaller than A[i].***

***Clone and extend this list by A[i]. We will discard all***

***other lists of same length as that of this modified list.***

Note that at any instance during our construction of active lists, the following condition is maintained.  
*“end element of smaller list is smaller than end elements of larger lists”*.  
It will be clear with an example, let us take example from [wiki](http://en.wikipedia.org/wiki/Longest_increasing_subsequence) {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}. 

A[0] = 0. Case 1. There are no active lists, create one.

0.

-----------------------------------------------------------------------------

A[1] = 8. Case 2. Clone and extend.

0.

0, 8.

-----------------------------------------------------------------------------

A[2] = 4. Case 3. Clone, extend and discard.

0.

0, 4.

0, 8. Discarded

-----------------------------------------------------------------------------

A[3] = 12. Case 2. Clone and extend.

0.

0, 4.

0, 4, 12.

-----------------------------------------------------------------------------

A[4] = 2. Case 3. Clone, extend and discard.

0.

0, 2.

0, 4. Discarded.

0, 4, 12.

-----------------------------------------------------------------------------

A[5] = 10. Case 3. Clone, extend and discard.

0.

0, 2.

0, 2, 10.

0, 4, 12. Discarded.

-----------------------------------------------------------------------------

A[6] = 6. Case 3. Clone, extend and discard.

0.

0, 2.

0, 2, 6.

0, 2, 10. Discarded.

-----------------------------------------------------------------------------

A[7] = 14. Case 2. Clone and extend.

0.

0, 2.

0, 2, 6.

0, 2, 6, 14.

-----------------------------------------------------------------------------

A[8] = 1. Case 3. Clone, extend and discard.

0.

0, 1.

0, 2. Discarded.

0, 2, 6.

0, 2, 6, 14.

-----------------------------------------------------------------------------

A[9] = 9. Case 3. Clone, extend and discard.

0.

0, 1.

0, 2, 6.

0, 2, 6, 9.

0, 2, 6, 14. Discarded.

-----------------------------------------------------------------------------

A[10] = 5. Case 3. Clone, extend and discard.

0.

0, 1.

0, 1, 5.

0, 2, 6. Discarded.

0, 2, 6, 9.

-----------------------------------------------------------------------------

A[11] = 13. Case 2. Clone and extend.

0.

0, 1.

0, 1, 5.

0, 2, 6, 9.

0, 2, 6, 9, 13.

-----------------------------------------------------------------------------

A[12] = 3. Case 3. Clone, extend and discard.

0.

0, 1.

0, 1, 3.

0, 1, 5. Discarded.

0, 2, 6, 9.

0, 2, 6, 9, 13.

-----------------------------------------------------------------------------

A[13] = 11. Case 3. Clone, extend and discard.

0.

0, 1.

0, 1, 3.

0, 2, 6, 9.

0, 2, 6, 9, 11.

0, 2, 6, 9, 13. Discarded.

-----------------------------------------------------------------------------

A[14] = 7. Case 3. Clone, extend and discard.

0.

0, 1.

0, 1, 3.

0, 1, 3, 7.

0, 2, 6, 9. Discarded.

0, 2, 6, 9, 11.

----------------------------------------------------------------------------

A[15] = 15. Case 2. Clone and extend.

0.

0, 1.

0, 1, 3.

0, 1, 3, 7.

0, 2, 6, 9, 11.

**0, 2, 6, 9, 11, 15. <-- LIS List**

----------------------------------------------------------------------------

It is required to understand above strategy to devise an algorithm. Also, ensure we have maintained the condition, “*end element of smaller list is smaller than end elements of larger lists*“. Try with few other examples, before reading further. It is important to understand what happening to end elements.  
**Algorithm:**  
Querying length of longest is fairly easy. Note that we are dealing with end elements only. We need not to maintain all the lists. We can store the end elements in an array. Discarding operation can be simulated with replacement, and extending a list is analogous to adding more elements to array.  
We will use an auxiliary array to keep end elements. The maximum length of this array is that of input. In the worst case the array divided into N lists of size one (*note that it doesn’t lead to worst case complexity*). To discard an element, we will trace ceil value of A[i] in auxiliary array (again observe the end elements in your rough work), and replace ceil value with A[i]. We extend a list by adding element to auxiliary array. We also maintain a counter to keep track of auxiliary array length.

Given below is code to find length of LIS

#include <iostream>

#include <vector>

// Binary search (note boundaries in the caller)

int CeilIndex(std::vector<int>& v, int l, int r, int key)

{

while (r - l > 1) {

int m = l + (r - l) / 2;

if (v[m] >= key)

r = m;

else

l = m;

}

return r;

}

int LongestIncreasingSubsequenceLength(std::vector<int>& v)

{

if (v.size() == 0)

return 0;

std::vector<int> tail(v.size(), 0);

int length = 1; // always points empty slot in tail

tail[0] = v[0];

for (size\_t i = 1; i < v.size(); i++) {

// new smallest value

if (v[i] < tail[0])

tail[0] = v[i];

// v[i] extends largest subsequence

else if (v[i] > tail[length - 1])

tail[length++] = v[i];

// v[i] will become end candidate of an existing

// subsequence or Throw away larger elements in all

// LIS, to make room for upcoming greater elements

// than v[i] (and also, v[i] would have already

// appeared in one of LIS, identify the location

// and replace it)

else

tail[CeilIndex(tail, -1, length - 1, v[i])] = v[i];

}

return length;

}

int main()

{

std::vector<int> v{ 2, 5, 3, 7, 11, 8, 10, 13, 6 };

std::cout << "Length of Longest Increasing Subsequence is "

<< LongestIncreasingSubsequenceLength(v) << '\n';

return 0;

}

**Output:**

Length of Longest Increasing Subsequence is 6

**Complexity:**  
The loop runs for N elements. In the worst case (what is worst case input?), we may end up querying ceil value using binary search (log *i*) for many A[i].  
Therefore, T(n) < O( log N! )  = O(N log N). Analyse to ensure that the upper and lower bounds are also O( N log N ). The complexity is THETA (N log N).

**Alternate implementation in various languages using their built in binary search functions are given below:**

#include <bits/stdc++.h>

using namespace std;

int LongestIncreasingSubsequenceLength(std::vector<int>& v)

{

if (v.size() == 0) // boundary case

return 0;

std::vector<int> tail(v.size(), 0);

int length = 1; // always points empty slot in tail

tail[0] = v[0];

for (int i = 1; i < v.size(); i++) {

// Do binary search for the element in

// the range from begin to begin + length

auto b = tail.begin(), e = tail.begin() + length;

auto it = lower\_bound(b, e, v[i]);

// If not present change the tail element to v[i]

if (it == tail.begin() + length)

tail[length++] = v[i];

else

\*it = v[i];

}

return length;

}

int main()

{

std::vector<int> v{ 2, 5, 3, 7, 11, 8, 10, 13, 6 };

std::cout

<< "Length of Longest Increasing Subsequence is "

<< LongestIncreasingSubsequenceLength(v);

return 0;

}

Output:

Length of Longest Increasing Subsequence is 6

# 395. [Space Optimized Solution of LCS](https://www.geeksforgeeks.org/space-optimized-solution-lcs/)

Given two strings, find the length of the longest subsequence present in both of them. 

**Examples:**

LCS for input Sequences “**ABCDGH**” and “**AEDFHR**” is “**ADH**” of length **3**.

LCS for input Sequences “**AGGTAB**” and “**GXTXAYB**” is “**GTAB**” of length **4**

## Solution:

We have discussed a [typical dynamic programming-based solution for LCS](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/). We can optimize the space used by LCS problem. We know the recurrence relationship of the LCS problem is

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs(string &X, string &Y)

{

int m = X.length(), n = Y.length();

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];

}

**How to find the length of LCS is O(n) auxiliary space?**

One important observation in the above simple implementation is, in each iteration of the outer loop we only need **values from all columns of the previous row**. So there is no need to store all rows in our DP matrix, we can just store two rows at a time and use them. In that way, used space will be reduced from L[m+1][n+1] to L[2][n+1]. Below is the implementation of the above idea.

// Space optimized 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)

{

// Find lengths of two strings

int m = X.length(), n = Y.length();

int L[2][n + 1];

// Binary index, used to

// index current row and

// previous row.

bool bi;

for (int i = 0; i <= m; i++)

{

// Compute current

// binary index

bi = i & 1;

for (int j = 0; j <= n; j++)

{

if (i == 0 || j == 0)

L[bi][j] = 0;

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

L[bi][j] = L[1 - bi][j - 1] + 1;

else

L[bi][j] = max(L[1 - bi][j],

L[bi][j - 1]);

}

}

// Last filled entry contains

// length of LCS

// for X[0..n-1] and Y[0..m-1]

return L[bi][n];

}

// Driver code

int main()

{

string X = "AGGTAB";

string Y = "GXTXAYB";

printf("Length of LCS is %d\n", lcs(X, Y));

return 0;

}

**Output:**

Length of LCS is 4

**Time Complexity :**O(m\*n)   
**Auxiliary Space :**O(n)

# 396. LCS (Longest Common Subsequence) of three strings

Given 3 strings A, B and C, the task is to find the longest common sub-sequence in all three given sequences.

**Example 1:**

**Input:**

A = "geeks", B = "geeksfor",

C = "geeksforgeeks"

**Output:** 5

**Explanation**: "geeks"is the longest common

subsequence with length 5.

â€‹**Example 2:**

**Input**:

A = "abcd", B = "efgh", C = "ijkl"

**Output:** 0

**Explanation**: There's no common subsequence

in all the strings.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **LCSof3()**which takes the strings A, B, C and their lengths n1, n2, n3 as input and returns the length of the longest common subsequence in all the 3 strings.

**Expected Time Complexity:**O(n1\*n2\*n3).  
**Expected Auxiliary Space:**O(n1\*n2\*n3).

**Constraints:**  
1<=n1, n2, n3<=20

## Solution:

This problem is simply an extension of [LCS](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/)  
Let the input sequences be X[0..m-1], Y[0..n-1] and Z[0..o-1] of lengths m, n and o respectively. And let L(X[0..m-1], Y[0..n-1], Z[0..o-1]) be the lengths of LCS of the three sequences X, Y and Z. Following is the implementation:

The idea is to take a 3D array to store the

length of common subsequence in all 3 given

sequences i. e., L[m + 1][n + 1][o + 1]

1- If any of the string is empty then there

is no common subsequence at all then

L[i][j][k] = 0

2- If the characters of all sequences match

(or X[i] == Y[j] ==Z[k]) then

L[i][j][k] = 1 + L[i-1][j-1][k-1]

3- If the characters of both sequences do

not match (or X[i] != Y[j] || X[i] != Z[k]

|| Y[j] !=Z[k]) then

L[i][j][k] = max(L[i-1][j][k],

L[i][j-1][k],

L[i][j][k-1])

Below is implementation of above idea.

// C++ program to find LCS of three strings

#include<bits/stdc++.h>

using namespace std;

/\* Returns length of LCS for X[0..m-1], Y[0..n-1]

and Z[0..o-1] \*/

int lcsOf3( string X, string Y, string Z, int m,

int n, int o)

{

int L[m+1][n+1][o+1];

/\* Following steps build L[m+1][n+1][o+1] in

bottom up fashion. Note that L[i][j][k]

contains length of LCS of X[0..i-1] and

Y[0..j-1] and Z[0.....k-1]\*/

for (int i=0; i<=m; i++)

{

for (int j=0; j<=n; j++)

{

for (int k=0; k<=o; k++)

{

if (i == 0 || j == 0||k==0)

L[i][j][k] = 0;

else if (X[i-1] == Y[j-1] && X[i-1]==Z[k-1])

L[i][j][k] = L[i-1][j-1][k-1] + 1;

else

L[i][j][k] = max(max(L[i-1][j][k],

L[i][j-1][k]),

L[i][j][k-1]);

}

}

}

/\* L[m][n][o] contains length of LCS for

X[0..n-1] and Y[0..m-1] and Z[0..o-1]\*/

return L[m][n][o];

}

/\* Driver program to test above function \*/

int main()

{

string X = "AGGT12";

string Y = "12TXAYB";

string Z = "12XBA";

int m = X.length();

int n = Y.length();

int o = Z.length();

cout << "Length of LCS is " << lcsOf3(X, Y,

Z, m, n, o);

return 0;

}

**Output:** 

Length of LCS is 2

**Another approach:**(Using recursion)

// C++ program to find LCS of three strings

#include<bits/stdc++.h>

using namespace std;

string X = "AGGT12";

string Y = "12TXAYB";

string Z = "12XBA";

int dp[100][100][100];

/\* Returns length of LCS for X[0..m-1], Y[0..n-1]

and Z[0..o-1] \*/

int lcsOf3(int i, int j,int k)

{

if(i==-1||j==-1||k==-1)

return 0;

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

return dp[i][j][k];

if(X[i]==Y[j] && Y[j]==Z[k])

return dp[i][j][k] = 1+lcsOf3(i-1,j-1,k-1);

else

return dp[i][j][k] = max(max(lcsOf3(i-1,j,k),

lcsOf3(i,j-1,k)),lcsOf3(i,j,k-1));

}

// Driver code

int main()

{

memset(dp, -1,sizeof(dp));

int m = X.length();

int n = Y.length();

int o = Z.length();

cout << "Length of LCS is " << lcsOf3(m-1,n-1,o-1);

}

**Output:** 

Length of LCS is 2

# 397. Maximum Sum Increasing Subsequence

Given an array of n positive integers. Find the sum of the maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order i.e. increasing subsequence.

**Example 1:**

**Input**: N = 5, arr[] = {1, 101, 2, 3, 100}

**Output:** 106

**Explanation**:The maximum sum of a

increasing sequence is obtained from

{1, 2, 3, 100}

**Example 2:**

**Input**: N = 3, arr[] = {1, 2, 3}

**Output:** 6

**Explanation**:The maximum sum of a

increasing sequence is obtained from

{1, 2, 3}

**Your Task:**  
You don't need to read input or print anything. Complete the function **maxSumIS()**which takes **N**and array **arr**as input parameters and returns the maximum value.

**Expected Time Complexity:** O(**N2**)  
**Expected Auxiliary Space:** O(**N**)

**Constraints:**  
1 ≤ N ≤ 103  
1 ≤ arr[i] ≤ 105

## Solution:

**Solution**   
This problem is a variation of standard [Longest Increasing Subsequence (LIS) problem](https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/). We need a slight change in the Dynamic Programming solution of [LIS problem](https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/). All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are the Dynamic Programming solution to the problem :

int maxSumIS(int arr[], int n)

{

// Your code goes here

int dp[n];

memset(dp, 0, sizeof(dp));

dp[0] = arr[0];

int res = arr[0];

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

dp[i] = arr[i];

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

if(arr[j]<arr[i]){

dp[i] = max(dp[i], dp[j]+arr[i]);

}

}

res = max(res, dp[i]);

}

return res;

}

Time Complexity: O(n^2)

Space Complexity O(n)

# 398. Count all subsequences having product less than K

Given a positive array, find the number of [subsequences](https://www.geeksforgeeks.org/subarraysubstring-vs-subsequence-and-programs-to-generate-them/) having product smaller than K.  
**Examples:** 

Input : [1, 2, 3, 4]

k = 10

Output :11

The subsequences are {1}, {2}, {3}, {4},

{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4},

{1, 2, 3}, {1, 2, 4}

Input : [4, 8, 7, 2]

k = 50

Output : 9

## Solution:

This problem can be solved using dynamic programming where dp[i][j] = number of subsequences having product less than i using first j terms of the array. Which can be obtained by : number of subsequences using first j-1 terms + number of subsequences that can be formed using j-th term.

// CPP program to find number of subarrays having

// product less than k.

#include <bits/stdc++.h>

using namespace std;

// Function to count numbers of such subsequences

// having product less than k.

int productSubSeqCount(vector<int> &arr, int k)

{

int n = arr.size();

int dp[k + 1][n + 1];

memset(dp, 0, sizeof(dp));

for (int i = 1; i <= k; i++) {

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

// number of subsequence using j-1 terms

dp[i][j] = dp[i][j - 1];

// if arr[j-1] > i it will surely make product greater

// thus it won't contribute then

if (arr[j - 1] <= i)

// number of subsequence using 1 to j-1 terms

// and j-th term

dp[i][j] += dp[i/arr[j-1]][j-1] + 1;

}

}

return dp[k][n];

}

// Driver code

int main()

{

vector<int> A;

A.push\_back(1);

A.push\_back(2);

A.push\_back(3);

A.push\_back(4);

int k = 10;

cout << productSubSeqCount(A, k) << endl;

}

**Output:** 

11

# 399. Longest subsequence such that difference between adjacent is one

Given an array A[] of size N, find the longest subsequence such that difference between adjacent elements is one.

**Example 1:**

**Input:** N = 7

A[] = {10, 9, 4, 5, 4, 8, 6}

**Output:** 3

**Explaination:** The three possible subsequences

{10, 9, 8} , {4, 5, 4} and {4, 5, 6}.

**Example 2:**

**Input:** N = 5

A[] = {1, 2, 3, 4, 5}

**Output:** 5

**Explaination:** All the elements can be

included in the subsequence.

**Your Task:**  
You do not need to read input. Your task is to complete the function **longestSubseq()** which takes N and A[] as input parameters and returns the length of the longest such subsequence.

**Expected Time Complexity:** O(N2)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ N ≤ 103  
1 ≤ A[i] ≤ 103

## Solution:

This problem is based upon the concept of [Longest Increasing Subsequence Problem](https://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/).

Let arr[0..n-1] be the input array and

dp[i] be the length of the longest subsequence (with

differences one) ending at index i such that arr[i]

is the last element of the subsequence.

Then, dp[i] can be recursively written as:

dp[i] = 1 + max(dp[j]) where 0 < j < i and

[arr[j] = arr[i] -1 or arr[j] = arr[i] + 1]

dp[i] = 1, if no such j exists.

To find the result for a given array, we need

to return max(dp[i]) where 0 < i < n.

Following is a Dynamic Programming based implementation. It follows the recursive structure discussed above.

// C++ program to find the longest subsequence such

// the difference between adjacent elements of the

// subsequence is one.

#include <bits/stdc++.h>

using namespace std;

// Function to find the length of longest subsequence

int longestSubseqWithDiffOne(int arr[], int n)

{

// Initialize the dp[] array with 1 as a

// single element will be of 1 length

int dp[n];

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

dp[i] = 1;

// Start traversing the given array

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

// Compare with all the previous elements

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

// If the element is consecutive then

// consider this subsequence and update

// dp[i] if required.

if ((arr[i] == arr[j] + 1) || (arr[i] == arr[j] - 1))

dp[i] = max(dp[i], dp[j] + 1);

}

}

// Longest length will be the maximum value

// of dp array.

int result = 1;

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

if (result < dp[i])

result = dp[i];

return result;

}

// Driver code

int main()

{

// Longest subsequence with one difference is

// {1, 2, 3, 4, 3, 2}

int arr[] = { 1, 2, 3, 4, 5, 3, 2 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << longestSubseqWithDiffOne(arr, n);

return 0;

}

**Output:**

6

**Time Complexity:**O(n2)   
**Auxiliary Space:**O(n)

**Efficient Approach**

int longestSubsequence(int N, int A[])

{

// code here

if(N==1)

return 1;

unordered\_map<int,int> mapp;

int res = 1;

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

if(mapp.count(A[i]+1) >0 || mapp.count(A[i]-1)>0){

mapp[A[i]]=1+max(mapp[A[i]+1],mapp[A[i]-1]);

}

else

mapp[A[i]]=1;

res = max(res, mapp[A[i]]);

}

return res;

}

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

# 400. Maximum subsequence sum such that no three are consecutive

Given a sequence of positive numbers, find the maximum sum that can be formed which has no three consecutive elements present.  
**Examples :**

**Input:** arr[] = {1, 2, 3}

**Output:** 5

We can't take three of them, so answer is

2 + 3 = 5

**Input:** arr[] = {3000, 2000, 1000, 3, 10}

**Output:** 5013

3000 + 2000 + 3 + 10 = 5013

**Input:** arr[] = {100, 1000, 100, 1000, 1}

**Output:** 2101

100 + 1000 + 1000 + 1 = 2101

**Input:** arr[] = {1, 1, 1, 1, 1}

**Output:** 4

**Input:** arr[] = {1, 2, 3, 4, 5, 6, 7, 8}

**Output:** 27

## Solution:

This problem is mainly an extension of below problem.  
[Maximum sum such that no two elements are adjacent](https://www.geeksforgeeks.org/maximum-sum-such-that-no-two-elements-are-adjacent/)  
We maintain an auxiliary array sum[] (of same size as input array) to find the result.

sum[i] : Stores result for subarray arr[0..i], i.e.,

maximum possible sum in subarray arr[0..i]

such that no three elements are consecutive.

sum[0] = arr[0]

// Note : All elements are positive

sum[1] = arr[0] + arr[1]

// We have three cases

// 1) Exclude arr[2], i.e., sum[2] = sum[1]

// 2) Exclude arr[1], i.e., sum[2] = sum[0] + arr[2]

// 3) Exclude arr[0], i.e., sum[2] = arr[1] + arr[2]

sum[2] = max(sum[1], arr[0] + arr[2], arr[1] + arr[2])

In general,

// We have three cases

// 1) Exclude arr[i], i.e., sum[i] = sum[i-1]

// 2) Exclude arr[i-1], i.e., sum[i] = sum[i-2] + arr[i]

// 3) Exclude arr[i-2], i.e., sum[i-3] + arr[i] + arr[i-1]

sum[i] = max(sum[i-1], sum[i-2] + arr[i],

sum[i-3] + arr[i] + arr[i-1])

Below is implementation of above idea.

// C++ program to find the maximum sum such that

// no three are consecutive

#include <bits/stdc++.h>

using namespace std;

// Returns maximum subsequence sum such that no three

// elements are consecutive

int maxSumWO3Consec(int arr[], int n)

{

// Stores result for subarray arr[0..i], i.e.,

// maximum possible sum in subarray arr[0..i]

// such that no three elements are consecutive.

int sum[n];

// Base cases (process first three elements)

if (n >= 1)

sum[0] = arr[0];

if (n >= 2)

sum[1] = arr[0] + arr[1];

if (n > 2)

sum[2] = max(sum[1], max(arr[1] +

arr[2], arr[0] + arr[2]));

// Process rest of the elements

// We have three cases

// 1) Exclude arr[i], i.e., sum[i] = sum[i-1]

// 2) Exclude arr[i-1], i.e., sum[i] = sum[i-2] + arr[i]

// 3) Exclude arr[i-2], i.e., sum[i-3] + arr[i] + arr[i-1]

for (int i = 3; i < n; i++)

sum[i] = max(max(sum[i - 1], sum[i - 2] + arr[i]),

arr[i] + arr[i - 1] + sum[i - 3]);

return sum[n - 1];

}

// Driver code

int main()

{

int arr[] = { 100, 1000 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << maxSumWO3Consec(arr, n);

return 0;

}

**Output:**

2101

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

**Another approach:** (Using recursion)

// C++ program to find the maximum sum such that

// no three are consecutive using recursion.

#include<bits/stdc++.h>

using namespace std;

int arr[] = {100, 1000, 100, 1000, 1};

int sum[10000];

// Returns maximum subsequence sum such that no three

// elements are consecutive

int maxSumWO3Consec(int n)

{

if(sum[n]!=-1)

return sum[n];

//Base cases (process first three elements)

if(n==0)

return sum[n] = 0;

if(n==1)

return sum[n] = arr[0];

if(n==2)

return sum[n] = arr[1]+arr[0];

// Process rest of the elements

// We have three cases

return sum[n] = max(max(maxSumWO3Consec(n-1),

maxSumWO3Consec(n-2) + arr[n]),

arr[n] + arr[n-1] + maxSumWO3Consec(n-3));

}

// Driver code

int main()

{

int n = sizeof(arr) / sizeof(arr[0]);

memset(sum,-1,sizeof(sum));

cout << maxSumWO3Consec(n);

// this code is contributed by Kushdeep Mittal

return 0;

}

**Output:**

2101

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

# 401. Egg Dropping Problem

You are given **N** identical eggs and you have access to a **K**-floored building from **1** to **K**.

There exists a floor **f** where **0** <= **f** <= **K**such that any egg dropped at a floor higher than **f** will break, and any egg dropped **at or below**floor **f**will **not break**. There are few rules given below.

* 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 the egg doesn't break at a certain floor, it will not break at any floor below.
* If the eggs breaks at a certain floor, it will break at any floor above.

Return the minimum number of moves that you need to determine with certainty what the value of **f** is.

For more description on this problem see [wiki page](http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle)

**Example 1:**

**Input:**

**N** = 1**, K** = 2

**Output:** 2

**Explanation:**

1. Drop the egg from floor 1. If it

  breaks, we know that f = 0.

2. Otherwise, drop the egg from floor 2.

  If it breaks, we know that f = 1.

3. If it does not break, then we know f = 2.

4. Hence, we need at minimum 2 moves to

  determine with certainty what the value of f is.

**Example 2:**

**Input:**

N = 2, K = 10

**Output:** 4

**Your Task:**  
Complete the function **eggDrop()** which takes two positive integer N and K as input parameters and returns the minimum number of attempts you need in order to find the critical floor.

**Expected Time Complexity** : O(N\*(K^2))  
**Expected Auxiliary Space**: O(N\*K)

**Constraints:**  
1<=N<=200  
1<=K<=200

## Solution:

**Method 1:** Recursion.   
In this post, we will discuss a solution to a general problem with ‘n’ eggs and ‘k’ floors. 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 floor numbers of every trial also.  
Meaning of a 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 a 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 floor should exist lower than ‘x’ in which 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 *worst*case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials. 

*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:*** *For example :   
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: 

#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 program to test

// to printDups

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;

}

**Output**

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

It should be noted that 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 many repeated subproblems when you draw the complete recursion tree even for small values of n and k. 

E(2, 4)

|

-------------------------------------

| | | |

| | | |

x=1/ x=2/ x=3/ x=4/

/ / .... ....

/ /

E(1, 0) E(2, 3) E(1, 1) E(2, 2)

/ /... /

x=1/ .....

/

E(1, 0) E(2, 2)

/

......

Partial recursion tree for 2 eggs and 4 floors.

**Complexity Analysis:** 

* **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.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.  
**Method 2:** Dynamic Programming.  
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 which will store the results of sub-problems so that to solve a sub-problem, it would only require a look-up from the table which will take **constant time**, which earlier took **exponential time**.  
Formally for filling DP[i][j] state where ‘i’ is the number of eggs and ‘j’ is the number of floors: 

* We have to traverse for each floor ‘x’ from ‘1’ to ‘j’ and find minimum of:

(1 + max( DP[i-1][j-1], DP[i][j-x] )).

This simulation will make things clear: 

*i => Number of eggs   
j => Number of floors   
Look up find maximum   
Lets fill the table for the following case:   
Floors = ‘4’   
Eggs = ‘2’  
1 2 3 4  
1 2 3 4 => 1   
1 2 2 3 => 2   
For ‘egg-1’ each case is the base case so the   
number of attempts is equal to floor number.  
For ‘egg-2’ it will take ‘1’ attempt for 1st   
floor which is base case.  
For floor-2 =>   
Taking 1st floor 1 + max(0, DP[1][1])   
Taking 2nd floor 1 + max(DP[1][1], 0)   
DP[2][2] = min(1 + max(0, DP[1][1]), 1 + max(DP[1][1], 0))  
For floor-3 =>   
Taking 1st floor 1 + max(0, DP[2][2])   
Taking 2nd floor 1 + max(DP[1][1], DP[2][1])   
Taking 3rd floor 1 + max(0, DP[2][2])   
DP[2][3]= min(‘all three floors’) = 2  
For floor-4 =>   
Taking 1st floor 1 + max(0, DP[2][3])   
Taking 2nd floor 1 + max(DP[1][1], DP[2][2])   
Taking 3rd floor 1 + max(DP[1][2], DP[2][1])   
Taking 4th floor 1 + max(0, DP[2][3])   
DP[2][4]= min(‘all four floors’) = 3*

// A Dynamic Programming based for

// the Egg Dropping Puzzle

#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)

{

/\* A 2D table where entry

eggFloor[i][j] will represent

minimum number of trials needed for

i eggs and j floors. \*/

int eggFloor[n + 1][k + 1];

int res;

int i, j, x;

// We need one trial for one floor and 0

// trials for 0 floors

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

eggFloor[i][1] = 1;

eggFloor[i][0] = 0;

}

// We always need j trials for one egg

// and j floors.

for (j = 1; j <= k; j++)

eggFloor[1][j] = j;

// Fill rest of the entries in table using

// optimal substructure property

for (i = 2; i <= n; i++) {

for (j = 2; j <= k; j++) {

eggFloor[i][j] = INT\_MAX;

for (x = 1; x <= j; x++) {

res = 1 + max(

eggFloor[i - 1][x - 1],

eggFloor[i][j - x]);

if (res < eggFloor[i][j])

eggFloor[i][j] = res;

}

}

}

// eggFloor[n][k] holds the result

return eggFloor[n][k];

}

/\* Driver program to test to printDups\*/

int main()

{

int n = 2, k = 36;

cout << "\nMinimum number of trials "

"in worst case with " << n<< " eggs and "<< k<<

" floors is "<< eggDrop(n, k);

return 0;

}

**Output**

Minimum number of trials in worst case with 2 eggs and 36 floors is 8

**Complexity Analysis:** 

* **Time Complexity:** O(n\*k^2).   
  Where ‘n’ is the number of eggs and ‘k’ is the number of floors, as we use a nested for loop ‘k^2’ times for each egg
* **Auxiliary Space:** O(n\*k).   
  As a 2-D array of size ‘n\*k’ is used for storing elements.

**Method 3:** Dynamic Programming using memoization.

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

vector<vector<int>> memo(MAX, vector<int> (MAX, -1));

int solveEggDrop(int n, int k) {

if(memo[n][k] != -1) { return memo[n][k];}

if (k == 1 || k == 0)

return k;

if (n == 1)

return k;

int min = INT\_MAX, x, res;

for (x = 1; x <= k; x++) {

res = max(

solveEggDrop(n - 1, x - 1),

solveEggDrop(n, k - x));

if (res < min)

min = res;

}

memo[n][k] = min+1;

return min + 1;

}

int main() {

int n = 2, k = 36;

cout<<solveEggDrop(n, k);

return 0;

}

**Output**

8

# 402. Maximum Length Chain of Pairs

You are given N pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed in this fashion. You have to find the longest chain which can be formed from the given set of pairs. 

**Example 1:**

**Input:**

N = 5

P[] = {5  24 , 39 60 , 15 28 , 27 40 , 50 90}

**Output:** 3

**Explanation**: The given pairs are { {5, 24}, {39, 60},

{15, 28}, {27, 40}, {50, 90} },the longest chain that

can be formed is of length 3, and the chain is

{{5, 24}, {27, 40}, {50, 90}}

â€‹**Example 2:**

**Input**:

N = 2

P[] = {5 10 , 1 11}

**Output:** 1

**Explanation**:The max length chain possible is only of

length one.

**Your Task:**  
You don't need to read input or print anything. Your task is to Complete the function **maxChainLen()** which takes a structure p[] denoting the pairs and n as the number of pairs and returns the length of the longest chain formed.

**Expected Time Complexity**: O(N\*N)  
**Expected Auxiliary Space**: O(N)  
  
**Constraints:**  
1<=N<=100

## Solution:

This problem is a variation of standard [Longest Increasing Subsequence](https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/) problem. Following is a simple two step process.   
1) Sort given pairs in increasing order of first (or smaller) element. Why do not need sorting? Consider the example {{6, 8}, {3, 4}} to understand the need of sorting. If we proceed to second step without sorting, we get output as 1. But the correct output is 2.   
2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.   
The following code is a slight modification of method 2 of [this post](https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/).

// CPP program for above approach

#include <bits/stdc++.h>

using namespace std;

// Structure for a Pair

class Pair

{

public:

int a;

int b;

};

// This function assumes that arr[]

// is sorted in increasing order

// according the first

// (or smaller) values in Pairs.

int maxChainLength( Pair arr[], int n)

{

int i, j, max = 0;

int \*mcl = new int[sizeof( int ) \* n ];

/\* Initialize MCL (max chain length)

values for all indexes \*/

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

mcl[i] = 1;

/\* Compute optimized chain

length values in bottom up manner \*/

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

for ( j = 0; j < i; j++ )

if ( arr[i].a > arr[j].b &&

mcl[i] < mcl[j] + 1)

mcl[i] = mcl[j] + 1;

// mcl[i] now stores the maximum

// chain length ending with Pair i

/\* Pick maximum of all MCL values \*/

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

if ( max < mcl[i] )

max = mcl[i];

/\* Free memory to avoid memory leak \*/

return max;

}

/\* Driver code \*/

int main()

{

Pair arr[] = { {5, 24}, {15, 25},

{27, 40}, {50, 60} };

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Length of maximum size chain is "

<< maxChainLength( arr, n );

return 0;

}

**Output**

Length of maximum size chain is 3

**Time Complexity:** O(n^2) where n is the number of pairs.

The given problem is also a variation of [**Activity Selection problem**](https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/)and can be solved in (nLogn) time. To solve it as a activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time.

**Another approach( Top-down Dynamic programming):**  
Now we will explore the way of solving this problem using the top-down approach of dynamic programming ***(***recursion + memorization).   
Since we are going to solve the above problem using top down method our first step is to figure out the recurrence relation. The best and the easiest way to get the recurrence relation is to think about the choices that we have at each state or position.   
If we look at the above problem carefully, we find two choices to be present at each position/index. The two choices are:   
***Choice 1:*** To select the element at the particular position and explore the rest, *(or)*   
***Choice 2:*** To leave the element at that position and explore the rest.   
Please note here that we can select the element at a particular position only if first element at that position is greater than the second element that we have previously chosen (this is a constraint given in the question). Hence, in the recursion we maintain a variable which would tell us the previous element that we picked.   
Also, we have to maximize our answer. Hence, we have to find out the maximum resulting option by exploring the above two choices at each position.   
The resulting recurrence relation would be:

**T(n) = max( maxlenchain(p,n,p[pos].second,0)+1,maxlenchain(p,n,prev\_choosen\_ele,pos+1) )**   
Please note the function signature is as follows:   
int cal(struct val p[],int n,int prev\_choosen\_ele,int pos);

Nevertheless, we should not forget our base condition in recursion. If not, our code would enjoy a vacation by just executing forever and not stopping at all.   
So, our base condition for this problem is quite simple. If we reach the end of our exploration, we just return 0, as no more chains would be possible.

**if(pos >= n) return 0;**

To avoid the repetitive task, we do the dynamic programming magic (It is a magic to reduce your time complexity). We store the position and previous element in a map. If we ever happened to come to the same position with the same previous element we do not recompute again. We just return the answer from the map.  
Below is the implementation of the above approach:

// CPP program for above approach

#include <bits/stdc++.h>

using namespace std;

// Structure val

struct val

{

int first;

int second;

};

map<pair<int, int>, int> m;

// Memoisation function

int findMaxChainLen(struct val p[], int n,

int prev, int pos)

{

// Check if pair { pos, prev } exists

// in m

if (m.find({ pos, prev }) != m.end())

{

return m[{ pos, prev }];

}

// Check if pos is >=n

if (pos >= n)

return 0;

// Check if p[pos].first is

// less than prev

if (p[pos].first <= prev)

{

return findMaxChainLen(p, n, prev,

pos + 1);

}

else

{

int ans = max(findMaxChainLen(p, n,

p[pos].second, 0) + 1,

findMaxChainLen(p, n,

prev, pos + 1));

m[{ pos, prev }] = ans;

return ans;

}

}

// Function to calculate maximum

// chain length

int maxChainLen(struct val p[], int n)

{

m.clear();

// Call memoisation function

int ans = findMaxChainLen(p, n, 0, 0);

return ans;

}

// Driver Code

int main()

{

int n = 5;

val p[n];

p[0].first = 5;

p[0].second = 24;

p[1].first = 39;

p[1].second = 60;

p[2].first = 15;

p[2].second = 28;

p[3].first = 27;

p[3].second = 40;

p[4].first = 50;

p[4].second = 90;

// Function Call

cout << maxChainLen(p, n) << endl;

return 0;

}

**Output**

3

# 403. Maximum size square sub-matrix with all 1s

Given a binary matrix **mat** of size **n** \* **m**, find out the maximum size square sub-matrix with all 1s.

**Example 1:**

**Input:** n = 2, m = 2

mat = {{1, 1},

  {1, 1}}

**Output:** 2

**Explaination:** The maximum size of the square

sub-matrix is 2. The matrix itself is the

maximum sized sub-matrix in this case.

**Example 2:**

**Input:** n = 2, m = 2

mat = {{0, 0},

  {0, 0}}

**Output:** 0

**Explaination:** There is no 1 in the matrix.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **maxSquare()** which takes n, m and mat as input parameters and returns the size of the maximum square sub-matrix of given matrix.

**Expected Time Complexity:** O(n\*m)  
**Expected Auxiliary Space:** O(n\*m)

**Constraints:**  
1 ≤ n, m ≤ 50  
0 ≤ mat[i][j] ≤ 1

## Solution:

Algorithm:   
Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents the size of the square sub-matrix with all 1s including M[i][j] where M[i][j] is the rightmost and bottom-most entry in sub-matrix.

1) Construct a sum matrix S[R][C] for the given M[R][C].

a) Copy first row and first columns as it is from M[][] to S[][]

b) For other entries, use following expressions to construct S[][]

If M[i][j] is 1 then

S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1

Else /\*If M[i][j] is 0\*/

S[i][j] = 0

2) Find the maximum entry in S[R][C]

3) Using the value and coordinates of maximum entry in S[i], print

sub-matrix of M[][]

For the given M[R][C] in the above example, constructed S[R][C] would be:

0 1 1 0 1

1 1 0 1 0

0 1 1 1 0

1 1 2 2 0

1 2 2 3 1

0 0 0 0 0

The value of the maximum entry in the above matrix is 3 and the coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

// C++ code for Maximum size square

// sub-matrix with all 1s

#include <bits/stdc++.h>

#define bool int

#define R 6

#define C 5

using namespace std;

void printMaxSubSquare(bool M[R][C])

{

int i,j;

int S[R][C];

int max\_of\_s, max\_i, max\_j;

/\* Set first column of S[][]\*/

for(i = 0; i < R; i++)

S[i][0] = M[i][0];

/\* Set first row of S[][]\*/

for(j = 0; j < C; j++)

S[0][j] = M[0][j];

/\* Construct other entries of S[][]\*/

for(i = 1; i < R; i++)

{

for(j = 1; j < C; j++)

{

if(M[i][j] == 1)

S[i][j] = min({S[i][j-1], S[i-1][j],

S[i-1][j-1]}) + 1; //better of using min in case of arguments more than 2

else

S[i][j] = 0;

}

}

/\* Find the maximum entry, and indexes of maximum entry

in S[][] \*/

max\_of\_s = S[0][0]; max\_i = 0; max\_j = 0;

for(i = 0; i < R; i++)

{

for(j = 0; j < C; j++)

{

if(max\_of\_s < S[i][j])

{

max\_of\_s = S[i][j];

max\_i = i;

max\_j = j;

}

}

}

cout<<"Maximum size sub-matrix is: \n";

for(i = max\_i; i > max\_i - max\_of\_s; i--)

{

for(j = max\_j; j > max\_j - max\_of\_s; j--)

{

cout << M[i][j] << " ";

}

cout << "\n";

}

}

/\* Driver code \*/

int main()

{

bool M[R][C] = {{0, 1, 1, 0, 1},

{1, 1, 0, 1, 0},

{0, 1, 1, 1, 0},

{1, 1, 1, 1, 0},

{1, 1, 1, 1, 1},

{0, 0, 0, 0, 0}};

printMaxSubSquare(M);

}

**Output:**

Maximum size sub-matrix is:

1 1 1

1 1 1

1 1 1

**Time Complexity:** O(m\*n) where m is the number of rows and n is the number of columns in the given matrix.   
**Auxiliary Space:** O(m\*n) where m is the number of rows and n is the number of columns in the given matrix.   
Algorithmic Paradigm: Dynamic Programming

**Space Optimized Solution:**In order to compute an entry at any position in the matrix we only need the current row and the previous row.

// C++ code for Maximum size square

// sub-matrix with all 1s

// (space optimized solution)

#include <bits/stdc++.h>

using namespace std;

#define R 6

#define C 5

void printMaxSubSquare(bool M[R][C])

{

int S[2][C], Max = 0;

// set all elements of S to 0 first

memset(S, 0, sizeof(S));

// Construct the entries

for (int i = 0; i < R;i++)

for (int j = 0; j < C;j++){

// Compute the entrie at the current position

int Entrie = M[i][j];

if(Entrie)

if(j)

Entrie = 1 + min(S[1][j - 1], min(S[0][j - 1], S[1][j]));

// Save the last entrie and add the new one

S[0][j] = S[1][j];

S[1][j] = Entrie;

// Keep track of the max square length

Max = max(Max, Entrie);

}

// Print the square

cout << "Maximum size sub-matrix is: \n";

for (int i = 0; i < Max; i++, cout << '\n')

for (int j = 0; j < Max;j++)

cout << "1 ";

}

// Driver code

int main ()

{

bool M[R][C] = {{0, 1, 1, 0, 1},

{1, 1, 0, 1, 0},

{0, 1, 1, 1, 0},

{1, 1, 1, 1, 0},

{1, 1, 1, 1, 1},

{0, 0, 0, 0, 0}};

printMaxSubSquare(M);

return 0;

// This code is contributed

// by Gatea David

}

**Output**

Maximum size sub-matrix is:

1 1 1

1 1 1

1 1 1

**Time Complexity**: O(m\*n) where m is the number of rows and n is the number of columns in the given matrix. **Auxiliary space:** O(n) where n is the number of columns in the given matrix.

# 404. Maximum sum of pairs with specific difference

Given an array of integers, **arr[]** and a number, **K**.You can pair two numbers of the array if the difference between them is strictly less than **K**. The task is to find the maximum possible sum of such disjoint pairs (i.e., each element of the array can be used at most once). The Sum of **P** pairs is the sum of all **2P** elements of pairs.

**Example 1:**

**Input :**

arr[] = {3, 5, 10, 15, 17, 12, 9}

K = 4

**Output :**

62

**Explanation :**

Then disjoint pairs with difference less

than K are, (3, 5), (10, 12), (15, 17)

max sum which we can get is

3 + 5 + 10 + 12 + 15 + 17 = 62

Note that an alternate way to form

disjoint pairs is,(3, 5), (9, 12), (15, 17)

but this pairing produces less sum.

**Example 2:**

**Input :**

arr[] = {5, 15, 10, 300}

K = 12

**Output :**

25

**Your Task:**  
You don't need to read, input, or print anything. Your task is to complete the function **maxSumPairWithDifferenceLessThanK()** which takes the array **arr[]**, its size **N,**and an integer **K**as inputs and returns the maximum possible sum of disjoint pairs.

**Expected Time Complexity:** O(N. log(N))  
**Expected Auxiliary Space:**O(N)

**Constraints:**  
1 ≤ N ≤ 105  
0 ≤ K ≤ 105  
1 ≤ arr[i] ≤ 104

## Solution:

**Approach:**  
First, we sort the given array in increasing order. Once array is sorted, we traverse the array. For every element, we try to pair it with its previous element first. Why do we prefer previous element? Let arr[i] can be paired with arr[i-1] and arr[i-2] (i.e. arr[i] – arr[i-1] < K and arr[i]-arr[i-2] < K). Since the array is sorted, value of arr[i-1] would be more than arr[i-2]. Also, we need to pair with difference less than k, it means if arr[i-2] can be paired, then arr[i-1] can also be paired in a sorted array.   
Now observing the above facts, we can formulate our dynamic programming solution as below,   
Let dp[i] denotes the maximum disjoint pair sum we can achieve using first i elements of the array. Assume currently, we are at i’th position, then there are two possibilities for us.

Pair up i with (i-1)th element, i.e.

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

Don't pair up, i.e.

dp[i] = dp[i-1]

Above iteration takes O(N) time and sorting of array will take O(N log N) time so total time complexity of the solution will be O(N log N)

// C++ program to find maximum pair sum whose

// difference is less than K

#include <bits/stdc++.h>

using namespace std;

// method to return maximum sum we can get by

// finding less than K difference pair

int maxSumPairWithDifferenceLessThanK(int arr[], int N, int K)

{

// Sort input array in ascending order.

sort(arr, arr+N);

// dp[i] denotes the maximum disjoint pair sum

// we can achieve using first i elements

int dp[N];

// if no element then dp value will be 0

dp[0] = 0;

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

{

// first give previous value to dp[i] i.e.

// no pairing with (i-1)th element

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

// if current and previous element can form a pair

if (arr[i] - arr[i-1] < K)

{

// update dp[i] by choosing maximum between

// pairing and not pairing

if (i >= 2)

dp[i] = max(dp[i], dp[i-2] + arr[i] + arr[i-1]);

else

dp[i] = max(dp[i], arr[i] + arr[i-1]);

}

}

// last index will have the result

return dp[N - 1];

}

// Driver code to test above methods

int main()

{

int arr[] = {3, 5, 10, 15, 17, 12, 9};

int N = sizeof(arr)/sizeof(int);

int K = 4;

cout << maxSumPairWithDifferenceLessThanK(arr, N, K);

return 0;

}

**Output**

62

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

An optimised solution is given below,

// C++ program to find maximum pair sum whose

// difference is less than K

#include <bits/stdc++.h>

using namespace std;

// Method to return maximum sum we can get by

// finding less than K difference pairs

int maxSumPair(int arr[], int N, int k)

{

int maxSum = 0;

// Sort elements to ensure every i and i-1 is closest

// possible pair

sort(arr, arr + N);

// To get maximum possible sum,

// iterate from largest to

// smallest, giving larger

// numbers priority over smaller

// numbers.

for (int i = N - 1; i > 0; --i)

{

// Case I: Diff of arr[i] and arr[i-1]

// is less then K,add to maxSum

// Case II: Diff between arr[i] and arr[i-1] is not

// less then K, move to next i since with

// sorting we know, arr[i]-arr[i-1] <

// rr[i]-arr[i-2] and so on.

if (arr[i] - arr[i - 1] < k)

{

// Assuming only positive numbers.

maxSum += arr[i];

maxSum += arr[i - 1];

// When a match is found skip this pair

--i;

}

}

return maxSum;

}

// Driver code

int main()

{

int arr[] = { 3, 5, 10, 15, 17, 12, 9 };

int N = sizeof(arr) / sizeof(int);

int K = 4;

cout << maxSumPair(arr, N, K);

return 0;

}

**Output**

62

***Time complexity:****O(N Log N)*  
***Auxiliary Space:****O(1)*

# 405. Min Cost Path Problem

Given a NxN matrix of positive integers. There are only three possible moves from a cell **Matrix[r][c]**.

1. Matrix [r+1] [c]
2. Matrix [r+1] [c-1]
3. Matrix [r+1] [c+1]

Starting from any column in row 0 return the largest sum of any of the paths up to row N-1.  
  
**Example 1:**

**Input:** N = 2

Matrix = {{348, 391},

{618, 193}}

**Output:** 1009

**Explaination:** The best path is 391 -> 618.

It gives the sum = 1009.

**Example 2:**

**Input:** N = 2

Matrix = {{2, 2},

{2, 2}}

**Output:** 4

**Explaination:** No matter which path is

chosen, the output is 4.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **maximumPath()**which takes the size N and the Matrix as input parameters and returns the highest maximum path sum.

**Expected Time Complexity:** O(N\*N)  
**Expected Auxiliary Space:** O(N\*N)

**Constraints:**  
1 ≤ N ≤ 500  
1 ≤ Matrix[i][j] ≤ 1000

## Solution:

int maximumPath(int N, vector<vector<int>> Matrix)

{

// code here

int dp[2][N];

memset(dp, 0, sizeof(dp));

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

dp[(N-1)%2][i] = Matrix[N-1][i];

}

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

for(int j=0;j<N;j++){

dp[i%2][j] = Matrix[i][j] + dp[1-(i%2)][j];

if(j!=0)

dp[i%2][j] = max(dp[i%2][j], Matrix[i][j] + dp[1-(i%2)][j-1]);

if(j<N-1)

dp[i%2][j] = max(dp[i%2][j], Matrix[i][j] + dp[1-(i%2)][j+1]);

}

}

int res = 0;

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

res = max(res, dp[0][i]);

}

return res;

}

# 406. [Maximum difference of zeros and ones in binary string](https://practice.geeksforgeeks.org/problems/maximum-difference-of-zeros-and-ones-in-binary-string4111/1)

Given a binary string **S** consisting of 0s and 1s. The task is to find the **maximum difference** of the number of **0s** and the number of **1s** (number of 0s – number of 1s) in the substrings of a string.

**Note:** In the case of all 1s, the answer will be -1.

**Example 1:**

**Input** : S = "11000010001"

**Output** : 6

**Explanatio:** From index 2 to index 9,

there are 7 0s and 1 1s, so number

of 0s - number of 1s is 6.

**Example 2:**

**Input:** S = "111111"

**Output:** -1

**Explanation:** S contains 1s only

**Your task:**  
You do not need to read any input or print anything. The task is to complete the function **maxSubstring()**, which takes a string as input and returns an integer.

**Expected Time Complexity:** O(|S|)  
**Expected Auxiliary Space:** O(|S|)

**Constraints:**  
1 ≤ |S| ≤ 105  
S contains 0s and 1s only

## Solution:

In the post we seen an efficient method that work in O(n) time and in O(1) extra space. Idea behind that if we convert all zeros into 1 and all ones into -1.now our problem reduces to find out the maximum sum sub\_array Using [Kadane’s Algorithm](https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/).

Input : S = "11000010001"

After converting '0' into 1 and

'1' into -1 our **S** Look Like

S = -1 -1 1 1 1 1 -1 1 1 1 -1

Now we have to find out Maximum Sum sub\_array

that is : 6 is that case

Output : 6

Below is the implementation of above idea.

// CPP Program to find the length of

// substring with maximum difference of

// zeros and ones in binary string.

#include <iostream>

using namespace std;

// Returns the length of substring with

// maximum difference of zeroes and ones

// in binary string

int findLength(string str, int n)

{

int current\_sum = 0;

int max\_sum = 0;

// traverse a binary string from left

// to right

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

// add current value to the current\_sum

// according to the Character

// if it's '0' add 1 else -1

current\_sum += (str[i] == '0' ? 1 : -1);

if (current\_sum < 0)

current\_sum = 0;

// update maximum sum

max\_sum = max(current\_sum, max\_sum);

}

// return -1 if string does not contain

// any zero that means all ones

// otherwise max\_sum

return max\_sum == 0 ? -1 : max\_sum;

}

// Driven Program

int main()

{

string s = "11000010001";

int n = 11;

cout << findLength(s, n) << endl;

return 0;

}

**Output:**

6

**Time Complexity :** O(n)   
**Space complexity :** O(1)

# 407. Minimum number of jumps to reach end

## Same as ques 10 of array.

# 408. Minimum cost to fill given weight in a bag

Given an array **cost[]** of positive integers of size **N** and an integer **W**, cost[i] represents the cost of **‘i’** kg packet of oranges, the task is to find the minimum cost to buy **W** kgs of oranges. If it is not possible to buy exactly **W** kg oranges then the output will be -1

**Note:**  
1. cost[i] = -1 means that ‘i’ kg packet of orange is unavailable  
2. It may be assumed that there is infinite supply of all available packet types.

**Example 1:**

**Input**: N = 5, arr[] = {20, 10, 4, 50, 100}

W = 5

**Output:** 14

**Explanation**: choose two oranges to minimize

cost. First orange of 2Kg and cost 10.

Second orange of 3Kg and cost 4.

**Example 2:**

**Input**: N = 5, arr[] = {-1, -1, 4, 3, -1}

W = 5

**Output:** -1

**Explanation**: It is not possible to buy 5

kgs of oranges

**Your Task:**  
You don't need to read input or print anything. Complete the function **minimumCost()**which takes **N, W,**and array **cost**as input parameters and returns the minimum value.  
  
**Expected Time Complexity:** O(**N\*W**)  
**Expected Auxiliary Space:** O(**N\*W**)  
  
**Constraints:**  
1 ≤ N, W ≤ 2\*102  
-1 ≤ cost[i] ≤ 105  
cost[i] ≠ 0

## Solution:

int minimumCost(int cost[], int N, int W)

{

// Your code goes here

int dp[W];

memset(dp, 0, sizeof(dp));

for(int i=1;i<=W;i++){

int t = INT\_MAX;

for(int j=1;j<=(i)/2;j++){

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

t = min(t, dp[j-1]+dp[i-j-1]);

}

if(i<=N && cost[i-1]!=-1)

t = min(t, cost[i-1]);

if(t==INT\_MAX)

t = -1;

dp[i-1] = t;

}

return dp[W-1];

}

# 409. Minimum removals from array to make max – min <= K

Given N integers and K, find the minimum number of elements that should be removed, such that Amax-Amin<=K. After the removal of elements, Amax and Amin is considered among the remaining elements.

**Examples:**

Input : a[] = {1, 3, 4, 9, 10, 11, 12, 17, 20}

k = 4

Output : 5

Explanation: Remove 1, 3, 4 from beginning

and 17, 20 from the end.

Input : a[] = {1, 5, 6, 2, 8} K=2

Output : 3

Explanation: There are multiple ways to

remove elements in this case.

One among them is to remove 5, 6, 8.

The other is to remove 1, 2, 5

## Solution:

**Approach:** Sort the given elements. Using the greedy approach, the best way is to remove the minimum element or the maximum element and then check if**Amax-Amin <= K**. There are various combinations of removals that have to be considered. We write a recurrence relation to try every possible combination. There will be two possible ways of removal, either we remove the minimum or we remove the maximum. **Let(i…j) be the index of elements left after removal of elements**. Initially, we start with i=0 and j=n-1 and the number of elements removed is 0 at the beginning. **We only remove an element if a[j]-a[i]>k,**the two possible ways of removal are **(i+1…j) or (i…j-1)**. The minimum of the two is considered.   
Let DPi, j be the number of elements that need to be removed so that after removal a[j]-a[i]<=k.

**Recurrence relation for sorted array:**

**DPi, j = 1+ (min(countRemovals(i+1, j), countRemovals(i, j-1))**

Below is the implementation of the above idea:

// CPP program to find minimum removals

// to make max-min <= K

#include <bits/stdc++.h>

using namespace std;

#define MAX 100

int dp[MAX][MAX];

// function to check all possible combinations

// of removal and return the minimum one

int countRemovals(int a[], int i, int j, int k)

{

// base case when all elements are removed

if (i >= j)

return 0;

// if condition is satisfied, no more

// removals are required

else if ((a[j] - a[i]) <= k)

return 0;

// if the state has already been visited

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

return dp[i][j];

// when Amax-Amin>d

else if ((a[j] - a[i]) > k) {

// minimum is taken of the removal

// of minimum element or removal

// of the maximum element

dp[i][j] = 1 + min(countRemovals(a, i + 1, j, k),

countRemovals(a, i, j - 1, k));

}

return dp[i][j];

}

// To sort the array and return the answer

int removals(int a[], int n, int k)

{

// sort the array

sort(a, a + n);

// fill all stated with -1

// when only one element

memset(dp, -1, sizeof(dp));

if (n == 1)

return 0;

else

return countRemovals(a, 0, n - 1, k);

}

// Driver Code

int main()

{

int a[] = { 1, 3, 4, 9, 10, 11, 12, 17, 20 };

int n = sizeof(a) / sizeof(a[0]);

int k = 4;

cout << removals(a, n, k);

return 0;

}

**Output**

5

**Time Complexity :**O(n2)   
**Auxiliary Space:**O(n2)

The solution could be further optimized. The idea is to sort the array in increasing order and traverse through all the elements (let’s say index i) and find the maximum element on its right (index j) such that arr[j] – arr[i] <= k. Thus, the number of elements to be removed becomes n-(j-i+1). The minimum number of elements can be found by taking the minimum of n-(j-i-1) overall i. The value of index j can be found through a binary search between start = i+1 and end = n-1;

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to find

// rightmost index

// which satisfies the condition

// arr[j] - arr[i] <= k

int findInd(int key, int i,

int n, int k, int arr[])

{

int start, end, mid, ind = -1;

// Initialising start to i + 1

start = i + 1;

// Initialising end to n - 1

end = n - 1;

// Binary search implementation

// to find the rightmost element

// that satisfy the condition

while (start < end)

{

mid = start + (end - start) / 2;

// Check if the condition satisfies

if (arr[mid] - key <= k)

{

// Store the position

ind = mid;

// Make start = mid + 1

start = mid + 1;

}

else

{

// Make end = mid

end = mid;

}

}

// Return the rightmost position

return ind;

}

// Function to calculate

// minimum number of elements

// to be removed

int removals(int arr[], int n, int k)

{

int i, j, ans = n - 1;

// Sort the given array

sort(arr, arr + n);

// Iterate from 0 to n-1

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

{

// Find j such that

// arr[j] - arr[i] <= k

j = findInd(arr[i], i, n, k, arr);

// If there exist such j

// that satisfies the condition

if (j != -1)

{

// Number of elements

// to be removed for this

// particular case is

// (j - i + 1)

ans = min(ans, n - (j - i + 1));

}

}

// Return answer

return ans;

}

// Driver Code

int main()

{

int a[] = {1, 3, 4, 9, 10,

11, 12, 17, 20};

int n = sizeof(a) / sizeof(a[0]);

int k = 4;

cout << removals(a, n, k);

return 0;

}

**Output**

5

**Time Complexity :**O(nlogn)

**Auxiliary Space:** O(n)

**Approach:**

1. The solution could be further optimized. The idea is to sort the array in increasing order and traverse through all the elements (let’s say index j) and find the minimum element on its left (index i) such that arr[j] – arr[i] <= k and store it in dp[j].
2. Thus, the number of elements to be removed becomes n-(j-i+1). The minimum number of elements can be found by taking the minimum of n-(j-i-1) overall j. The value of index i can be found through its previous dp array element value.

Below is the implementation of the approach:

// C++ program for the above approach

#include<bits/stdc++.h>

using namespace std;

// To sort the array and return the answer

int removals(int arr[], int n, int k)

{

// sort the array

sort(arr, arr + n);

int dp[n];

// fill all stated with -1

// when only one element

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

dp[i] = -1;

// as dp[0] = 0 (base case) so min

// no of elements to be removed are

// n-1 elements

int ans = n - 1;

dp[0] = 0;

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

{

dp[i] = i;

int j = dp[i - 1];

while (j != i && arr[i] - arr[j] > k)

{

j++;

}

dp[i] = min(dp[i], j);

ans = min(ans, (n - (i - j + 1)));

}

return ans;

}

// Driver code

int main()

{

int a[] = { 1, 3, 4, 9, 10, 11, 12, 17, 20 };

int n = sizeof(a) / sizeof(a[0]);

int k = 4;

cout<<removals(a, n, k);

return 0;

}

**Output**

5

**Time Complexity**: O(nlog n). As outer loop is going to make n iterations. And the inner loop iterates at max n times for all outer iterations. Because we start value of j from dp[i-1] and loops it until it reaches i and then for the next element we again start from the previous dp[i] value. So the total time complexity is O(n) if we don’t consider the complexity of the sorting as it is not considered in the above solution as well.

**Auxiliary Space**: O(n)

**Space Optimised Approach**

The solution could be further optimized. It can be done in **Auxiliary Space: O(1).**The idea is to first sort the array in ascending order. The keep 2 pointers, say **i**and **j,**where j iterates from index 1 to index n-1 and keeps track of ending subarray with the difference in max and min less than k, and i keeps the track of starting index of the subarray. If we find that a[j] – a[i[ <=k (means the i to j subarray is valid) we update the length from i to j in a variable to track of max length so far. Else, we update the starting index i with i+1.

At first it seems that subarrays from i to j are ignored, but obviously their lengths are less than i to j subarray, so we don’t care about them.

// C++ program for the above approach

#include<bits/stdc++.h>

using namespace std;

int removal(int a[], int n, int k)

{

// Sort the Array; Time complexity:O(nlogn)

sort(a, a + n);

// to store the max length of

// array with difference <= k

int maxLen = INT\_MIN;

int i = 0;

// J goes from 1 to n-1 in n-2 iterations

// Thus time complexity of below loop is O(n)

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

// if the subarray from i to j index is valid

// the store it's length

if (a[j] - a[i] <= k) {

maxLen = max(maxLen, j - i + 1);

}

// if subarray difference exceeds k

// change starting position, i.e. i

else {

i++;

}

}

// no. of elements need to be removed is

// Length of array - max subarray with diff <= k

int removed = n - maxLen;

return removed;

}

//Driver Code

int main()

{

int a[] = { 1, 3, 4, 9, 10, 11, 12, 17, 20 };

int n = sizeof(a) / sizeof(a[0]);

int k = 4;

cout << removal(a, n, k);

return 0;

}

**Time Complexity: O(nlogn)**For sorting the array, we require O(nlogn) time, and no extra space. And for calculating we only traverse the loop once, thus it has O(n) time complexity. So, overall time complexity is O(nlogn).

**Auxiliary Space: O(1)**

# 410. Longest Common Substring

Given two strings. The task is to find the length of the longest common substring.

**Example 1:**

**Input:** S1 = "ABCDGH", S2 = "ACDGHR"

**Output:** 4

**Explanation**: The longest common substring

is "CDGH" which has length 4.

**Example 2:**

**Input**: S1 = "ABC", S2 "ACB"

**Output:** 1

**Explanation**: The longest common substrings

are "A", "B", "C" all having length 1.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **longestCommonSubstr()**which takes the string S1, string S2 and their length n and m as inputs and returns the length of the longest common substring in S1 and S2.

**Expected Time Complexity:**O(n\*m).  
**Expected Auxiliary Space:**O(n\*m).

**Constraints:**  
1<=n, m<=1000

## Solution:

**Approach:**  
Let m and n be the lengths of the first and second strings respectively.

A **simple solution** is to one by one consider all substrings of the first string and for every substring check if it is a substring in the second string. Keep track of the maximum length substring. There will be O(m^2) substrings and we can find whether a string is substring on another string in O(n) time (See [this](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/)). So overall time complexity of this method would be O(n \* m2)

**Dynamic Programming**can be used to find the longest common substring in O(m\*n) time. The idea is to find the length of the longest common suffix for all substrings of both strings and store these lengths in a table.

*The longest common suffix has following optimal substructure property.   
If last characters match, then we reduce both lengths by 1   
LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 if X[m-1] = Y[n-1]   
If last characters do not match, then result is 0, i.e.,   
LCSuff(X, Y, m, n) = 0 if (X[m-1] != Y[n-1])  
Now we consider suffixes of different substrings ending at different indexes.   
The maximum length Longest Common Suffix is the longest common substring.   
LCSubStr(X, Y, m, n) = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m and 1 <= j <= n*

Following is the iterative implementation of the above solution.

/\* Dynamic Programming solution to

find length of the

longest common substring \*/

#include <iostream>

#include <string.h>

using namespace std;

/\* Returns length of longest

common substring of X[0..m-1]

and Y[0..n-1] \*/

int LCSubStr(char\* X, char\* Y, int m, int n)

{

// Create a table to store

// lengths of longest

// common suffixes of substrings.

// Note that LCSuff[i][j] contains

// length of longest common suffix

// of X[0..i-1] and Y[0..j-1].

int LCSuff[m + 1][n + 1];

int result = 0; // To store length of the

// longest common substring

/\* Following steps build LCSuff[m+1][n+1] in

bottom up fashion. \*/

for (int i = 0; i <= m; i++)

{

for (int j = 0; j <= n; j++)

{

// The first row and first column

// entries have no logical meaning,

// they are used only for simplicity

// of program

if (i == 0 || j == 0)

LCSuff[i][j] = 0;

else if (X[i - 1] == Y[j - 1]) {

LCSuff[i][j] = LCSuff[i - 1][j - 1] + 1;

result = max(result, LCSuff[i][j]);

}

else

LCSuff[i][j] = 0;

}

}

return result;

}

// Driver code

int main()

{

char X[] = "OldSite:GeeksforGeeks.org";

char Y[] = "NewSite:GeeksQuiz.com";

int m = strlen(X);

int n = strlen(Y);

cout << "Length of Longest Common Substring is "

<< LCSubStr(X, Y, m, n);

return 0;

}

**Output**

Length of Longest Common Substring is 10

**Time Complexity:** O(m\*n)   
**Auxiliary Space:** O(m\*n)

**Another approach:**(Space optimized approach).  
In the above approach, we are only using the last row of the 2-D array only, hence we can optimize the space by using   
a 2-D array of dimension 2\*(min(n,m)).

Below is the implementation of the above approach:

// C++ implementation of the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to find the length of the

// longest LCS

int LCSubStr(string s, string t, int n, int m)

{

// Create DP table

int dp[2][m + 1];

int res = 0;

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

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

if (s[i - 1] == t[j - 1]) {

dp[i % 2][j] = dp[(i - 1) % 2][j - 1] + 1;

if (dp[i % 2][j] > res)

res = dp[i % 2][j];

}

else

dp[i % 2][j] = 0;

}

}

return res;

}

// Driver Code

int main()

{

string X = "OldSite:GeeksforGeeks.org";

string Y = "NewSite:GeeksQuiz.com";

int m = X.length();

int n = Y.length();

cout << LCSubStr(X, Y, m, n);

return 0;

cout << "GFG!";

return 0;

}

**Output**

10

**Time Complexity:** O(n\*m)  
**Auxiliary Space:**O(min(m,n))

# 411. Count number of ways to reach a given score in a game

Consider a game where a player can score **3** or **5** or **10** points in a move. Given a total score **n**, find number of distinct combinations to reach the given score.

**Example:**

**Input**

3

8

20

13

**Output**

1

4

2

**Explanation**

For 1st example when n = 8

{ 3, 5 } and {5, 3} are the

two possible permutations but

these represent the same

cobmination. Hence output is 1.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **count( )** which takes **N as** input parameter and returns the answer to the problem.

**Constraints:**  
1 ≤ **T** ≤ 100  
1 ≤ **n** ≤ 1000

## Solution:

This problem is a variation of [coin change problem](https://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/) and can be solved in O(n) time and O(n) auxiliary space.  
The idea is to create a table of size n+1 to store counts of all scores from 0 to n. For every possible move (3, 5 and 10), increment values in table.

// A C++ program to count number of

// possible ways to a given score

// can be reached in a game where a

// move can earn 3 or 5 or 10

#include <iostream>

using namespace std;

// Returns number of ways

// to reach score n

int count(int n)

{

// table[i] will store count

// of solutions for value i.

int table[n + 1], i;

// Initialize all table

// values as 0

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

table[j] = 0;

// Base case (If given value is 0)

table[0] = 1;

// One by one consider given 3 moves

// and update the table[] values after

// the index greater than or equal to

// the value of the picked move

for (i = 3; i <= n; i++)

table[i] += table[i - 3];

for (i = 5; i <= n; i++)

table[i] += table[i - 5];

for (i = 10; i <= n; i++)

table[i] += table[i - 10];

return table[n];

}

// Driver Code

int main(void)

{

int n = 20;

cout << "Count for " << n

<< " is " << count(n) << endl;

n = 13;

cout <<"Count for "<< n<< " is "

<< count(n) << endl;

return 0;

}

**Output:**

Count for 20 is 4

Count for 13 is 2

# 412. Count Balanced Binary Trees of Height h

Given a height h, count the maximum number of balanced binary trees possible with height h. Print the result modulo **109 + 7**.  
**Note :**A balanced binary tree is one in which for every node, the difference between heights of left and right subtree is not more than 1.  
  
**Example 1:**

**Input**: h = 2

**Output:** 3

**Explanation**: The maximum number of balanced

binary trees possible with height 2 is 3.

**Example 2:**

**Input:** h = 3

**Output:**15

**Explanation**: The maximum number of balanced

binary trees possible with height 3 is 15.

**Your Task:**  
You dont need to read input or print anything. Complete the function **countBT()**which takes h as input parameter and returns the maximum number of balanced binary trees possible with height h.   
  
**Expected Time Complexity:** O(h)  
**Expected Auxiliary Space:** O(h)  
  
**Constraints:**  
1<= n <=103

## Solution:

Following are the balanced binary trees of height 3. 

https://media.geeksforgeeks.org/wp-content/uploads/TreeNodes-1.png

Height of tree, h = 1 + max(left height, right height)  
Since the difference between the heights of left and right subtree is not more than one, possible heights of left and right part can be one of the following: 

1. (h-1), (h-2)
2. (h-2), (h-1)
3. (h-1), (h-1)

count(h) = count(h-1) \* count(h-2) +

count(h-2) \* count(h-1) +

count(h-1) \* count(h-1)

= 2 \* count(h-1) \* count(h-2) +

count(h-1) \* count(h-1)

= count(h-1) \* (2\*count(h - 2) +

count(h - 1))

Hence we can see that the problem has optimal substructure property.  
A **recursive function** to count no of balanced binary trees of height h is: 

int countBT(int h)

{

// One tree is possible with height 0 or 1

if (h == 0 || h == 1)

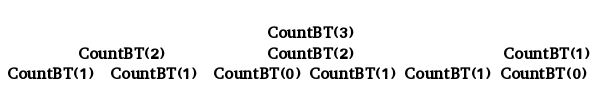
return 1;

return countBT(h-1) \* (2 \*countBT(h-2) +

countBT(h-1));

}

The time complexity of this recursive approach will be exponential. The recursion tree for the problem with h = 3 looks like : 



As we can see, sub-problems are solved repeatedly. Therefore we store the results as we compute them.   
An efficient dynamic programming approach will be as follows :   
Below is the implementation of above approach:

// C++ program to count number of balanced

// binary trees of height h.

#include <bits/stdc++.h>

#define mod 1000000007

using namespace std;

long long int countBT(int h) {

long long int dp[h + 1];

//base cases

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

for(int i = 2; i <= h; i++) {

dp[i] = (dp[i - 1] \* ((2 \* dp [i - 2])%mod + dp[i - 1])%mod) % mod;

}

return dp[h];

}

// Driver program

int main()

{

int h = 3;

cout << "No. of balanced binary trees"

" of height h is: "

<< countBT(h) << endl;

}

**Output**

No. of balanced binary trees of height h is: 15

**Time Complexity : O(n)**

**Space Complexity : O(n)**

**Memory efficient Dynamic Programming approach :**

If we observe carefully, in the previous approach to calculate dp[i] we are using dp[i-1] and dp[i-2] only and dp[0] to dp[i-3] are no longer required. Hence we can replace dp[i],dp[i-1] and dp[i-2] with dp2, dp1 and dp0 respectively.

// C++ program to count number of balanced

// binary trees of height h.

#include <bits/stdc++.h>

using namespace std;

long long int countBT(int h) {

if(h<2){

return 1;

}

const int BIG\_PRIME = 1000000007;

long long int dp0 = 1, dp1 = 1,dp2;

for(int i = 2; i <= h; i++) {

dp2 = (dp1 \* ((2 \* dp0)%BIG\_PRIME + dp1)%BIG\_PRIME) % BIG\_PRIME;

// update dp1 and dp0

dp0 = dp1;

dp1 = dp2;

// Don't commit following simple mistake

// dp1 = dp0;

// dp0 = dp1;

}

return dp2;

}

// Driver program

int main()

{

int h = 3;

cout << "No. of balanced binary trees"

" of height h is: "

<< countBT(h) << endl;

}

**Output**

No. of balanced binary trees of height h is: 15

**Time Complexity : O(n)**

**Space Complexity : O(1)**

# 413. LargestSum Contiguous Subarray [V>V>V>V IMP ]

## Same as ques 13 of array.

# 414. Smallest sum contiguous subarray

Given an array containing **n** integers. The problem is to find the sum of the elements of the contiguous subarray having the smallest(minimum) sum.  
Examples: 

Input : arr[] = {3, -4, 2, -3, -1, 7, -5}

Output : -6

Subarray is **{-4, 2, -3, -1}** = -6

Input : arr = {2, 6, 8, 1, 4}

Output : 1

## Solution:

**Naive Approach:** Consider all the contiguous subarrays of different sizes and find their sum. The subarray having the smallest(minimum) sum is the required answer.  
**Efficient Approach:** It is a variation to the problem of finding the [largest sum contiguous subarray](https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/) based on the idea of Kadane’s algorithm.   
**Algorithm:** 

**smallestSumSubarr(arr, n)**

Initialize **min\_ending\_here** = INT\_MAX

Initialize **min\_so\_far** = INT\_MAX

for **i** = 0 to n-1

if min\_ending\_here > 0

min\_ending\_here = arr[i]

else

min\_ending\_here += arr[i]

min\_so\_far = min(min\_so\_far, min\_ending\_here)

return min\_so\_far

// C++ implementation to find the smallest sum

// contiguous subarray

#include <bits/stdc++.h>

using namespace std;

// function to find the smallest sum contiguous subarray

int smallestSumSubarr(int arr[], int n)

{

// to store the minimum value that is ending

// up to the current index

int min\_ending\_here = INT\_MAX;

// to store the minimum value encountered so far

int min\_so\_far = INT\_MAX;

// traverse the array elements

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

{

// if min\_ending\_here > 0, then it could not possibly

// contribute to the minimum sum further

if (min\_ending\_here > 0)

min\_ending\_here = arr[i];

// else add the value arr[i] to min\_ending\_here

else

min\_ending\_here += arr[i];

// update min\_so\_far

min\_so\_far = min(min\_so\_far, min\_ending\_here);

}

// required smallest sum contiguous subarray value

return min\_so\_far;

}

// Driver program to test above

int main()

{

int arr[] = {3, -4, 2, -3, -1, 7, -5};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Smallest sum: "

<< smallestSumSubarr(arr, n);

return 0;

}

**Output:**

Smallest sum: -6

Time Complexity: O(n)

# 415. [Unbounded Knapsack (Repetition of items allowed)](https://practice.geeksforgeeks.org/problems/knapsack-with-duplicate-items4201/1)

Given a set of **N** items, each with a weight and a value, represented by the array **w[]** and **val[]** respectively. Also, a knapsack with weight limit **W**.  
The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.  
Note: Each item can be taken any number of times.

**Example 1:**

**Input:** N = 2, W = 3

val[] = {1, 1}

wt[] = {2, 1}

**Output:** 3

**Explanation:**

1.Pick the 2nd element thrice.

2.Total profit = 1 + 1 + 1 = 3. Also the total

 weight = 1 + 1 + 1 = 3 which is <= W.

**Example 2:**

**Input:** N = 4, W = 8

val[] = {1, 4, 5, 7}

wt[] = {1, 3, 4, 5}

**Output:** 11

**Explanation:** The optimal choice is to

pick the 2nd and 4th element.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **knapSack()** which takes the values N, W and the arrays val[] and wt[] as input parameters and returns the maximum possible value.

**Expected Time Complexity:** O(N\*W)  
**Expected Auxiliary Space:**O(W)

**Constraints:**  
1 ≤ N, W ≤ 1000  
1 ≤ val[i], wt[i] ≤ 100

## Solution:

Its an unbounded knapsack problem as we can use 1 or more instances of any resource. A simple 1D array, say dp[W+1] can be used such that dp[i] stores the maximum value which can achieved using all items and i capacity of knapsack. Note that we use 1D array here which is different from classical knapsack where we used 2D array. Here number of items never changes. We always have all items available.  
We can recursively compute dp[] using below formula

dp[i] = 0

dp[i] = max(dp[i], dp[i-wt[j]] + val[j]

where j varies from 0

to n-1 such that:

wt[j] <= i

result = d[W]

Below is the implementation of above idea.

// C++ program to find maximum achievable value

// with a knapsack of weight W and multiple

// instances allowed.

#include<bits/stdc++.h>

using namespace std;

// Returns the maximum value with knapsack of

// W capacity

int unboundedKnapsack(int W, int n,

int val[], int wt[])

{

// dp[i] is going to store maximum value

// with knapsack capacity i.

int dp[W+1];

memset(dp, 0, sizeof dp);

// Fill dp[] using above recursive formula

for (int i=0; i<=W; i++)

for (int j=0; j<n; j++)

if (wt[j] <= i)

dp[i] = max(dp[i], dp[i-wt[j]] + val[j]);

return dp[W];

}

// Driver program

int main()

{

int W = 100;

int val[] = {10, 30, 20};

int wt[] = {5, 10, 15};

int n = sizeof(val)/sizeof(val[0]);

cout << unboundedKnapsack(W, n, val, wt);

return 0;

}

**Output:**

300

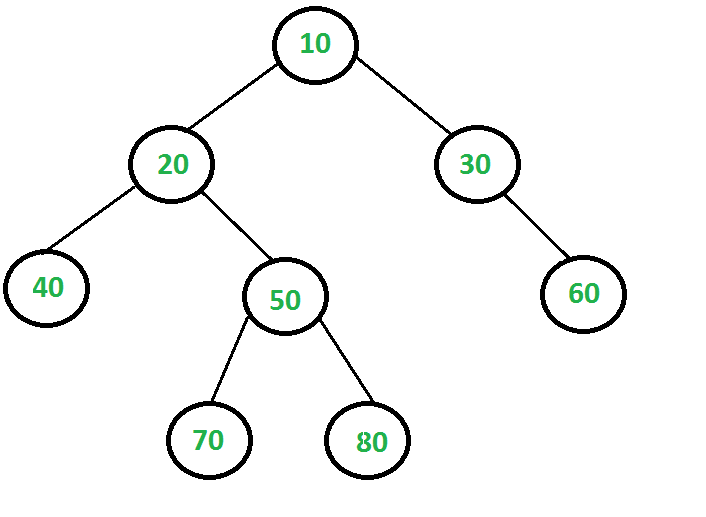
# 416. Word Break Problem

## Same as ques 63 of string.

# 417. Largest Independent Set Problem

Given a Binary Tree, find size of the **L**argest **I**ndependent **S**et(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



## Solution:

A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

**1) Optimal Substructure:**   
Let LISS(X) indicates size of largest independent set of a tree with root X.

LISS(X) = MAX { (1 + sum of LISS for all grandchildren of X),

(sum of LISS for all children of X) }

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

**2) Overlapping Subproblems**   
Following is recursive implementation that simply follows the recursive structure mentioned above.

// A naive recursive implementation of

// Largest Independent Set problem

#include <bits/stdc++.h>

using namespace std;

// A utility function to find

// max of two integers

int max(int x, int y)

{

return (x > y) ? x : y;

}

/\* A binary tree node has data,

pointer to left child and a

pointer to right child \*/

class node

{

public:

int data;

node \*left, \*right;

};

// The function returns size of the

// largest independent set in a given

// binary tree

int LISS(node \*root)

{

if (root == NULL)

return 0;

// Calculate size excluding the current node

int size\_excl = LISS(root->left) +

LISS(root->right);

// Calculate size including the current node

int size\_incl = 1;

if (root->left)

size\_incl += LISS(root->left->left) +

LISS(root->left->right);

if (root->right)

size\_incl += LISS(root->right->left) +

LISS(root->right->right);

// Return the maximum of two sizes

return max(size\_incl, size\_excl);

}

// A utility function to create a node

node\* newNode( int data )

{

node\* temp = new node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver Code

int main()

{

// Let us construct the tree

// given in the above diagram

node \*root = newNode(20);

root->left = newNode(8);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

root->right = newNode(22);

root->right->right = newNode(25);

cout << "Size of the Largest"

<< " Independent Set is "

<< LISS(root);

return 0;

}

**Output:**

Size of the Largest Independent Set is 5

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems,](https://www.geeksforgeeks.org/archives/tag/dynamic-programming) recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following are implementation of Dynamic Programming based solution. In the following solution, an additional field ‘liss’ is added to tree nodes. The initial value of ‘liss’ is set as 0 for all nodes. The recursive function LISS() calculates ‘liss’ for a node only if it is not already set.

/\* Dynamic programming based program

for Largest Independent Set problem \*/

#include <bits/stdc++.h>

using namespace std;

// A utility function to find max of two integers

int max(int x, int y) { return (x > y)? x: y; }

/\* A binary tree node has data, pointer

to left child and a pointer to

right child \*/

class node

{

public:

int data;

int liss;

node \*left, \*right;

};

// A memoization function returns size

// of the largest independent set in

// a given binary tree

int LISS(node \*root)

{

if (root == NULL)

return 0;

if (root->liss)

return root->liss;

if (root->left == NULL && root->right == NULL)

return (root->liss = 1);

// Calculate size excluding the current node

int liss\_excl = LISS(root->left) + LISS(root->right);

// Calculate size including the current node

int liss\_incl = 1;

if (root->left)

liss\_incl += LISS(root->left->left) + LISS(root->left->right);

if (root->right)

liss\_incl += LISS(root->right->left) + LISS(root->right->right);

// Maximum of two sizes is LISS, store it for future uses.

root->liss = max(liss\_incl, liss\_excl);

return root->liss;

}

// A utility function to create a node

node\* newNode(int data)

{

node\* temp = new node();

temp->data = data;

temp->left = temp->right = NULL;

temp->liss = 0;

return temp;

}

// Driver code

int main()

{

// Let us construct the tree

// given in the above diagram

node \*root = newNode(20);

root->left = newNode(8);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

root->right = newNode(22);

root->right->right = newNode(25);

cout << "Size of the Largest Independent Set is " << LISS(root);

return 0;

}

**Output:**

Size of the Largest Independent Set is 5

**Time Complexity:** O(n) where n is the number of nodes in given Binary tree.

# 418. Partition problem

Given an array **arr[]** of size **N**, check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

**Example 1:**

**Input:** N = 4

arr = {1, 5, 11, 5}

**Output:** YES

**Explaination:**

The two parts are {1, 5, 5} and {11}.

**Example 2:**

**Input:** N = 3

arr = {1, 3, 5}

**Output:** NO

**Explaination:** This array can never be

partitioned into two such parts.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **equalPartition()** which takes the value N and the array as input parameters and returns 1 if the partition is possible. Otherwise, returns 0.

**Expected Time Complexity:** O(N\*sum of elements)  
**Expected Auxiliary Space:** O(N\*sum of elements)

**Constraints:**  
1 ≤ N ≤ 100  
1 ≤ arr[i] ≤ 1000

## Solution:

class Solution{

public:

int fun(int N, int arr[], int target, int sum, int ind, vector<vector<int>> &dp){

if(dp[sum][ind]!=-1)

return dp[sum][ind];

if(sum==target)

return dp[sum][ind] = 1;

if(ind==N || sum>target)

return dp[sum][ind] = 0;

int excl = fun(N, arr, target, sum, ind+1, dp);

int incl = fun(N, arr, target, sum+arr[ind], ind+1, dp);

if(excl==1||incl==1)

return dp[sum][ind] = 1;

return dp[sum][ind] = 0;

}

int equalPartition(int N, int arr[])

{

int target = 0;

for(int i=0;i<N;i++)

target += arr[i];

if(target%2!=0)

return 0;

vector<vector<int>> dp(target+1, vector<int> (N+1, -1));

return fun(N, arr, target/2, 0, 0, dp);

}

};

**For Space Optimization, use below concept**

We have discussed a [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/)based solution in below post.   
[Dynamic Programming | Set 25 (Subset Sum Problem)](https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/)  
The solution discussed above requires O(n \* sum) space and O(n \* sum) time. We can optimize space. We create a boolean 2D array subset[2][sum+1]. Using bottom up manner we can fill up this table. The idea behind using **2 in “subset[2][sum+1]”** is that for filling a row only the values from previous row is required. So alternate rows are used either making the first one as current and second as previous or the first as previous and second as current.

// Returns true if there exists a subset

// with given sum in arr[]

#include <iostream>

using namespace std;

bool isSubsetSum(int arr[], int n, int sum)

{

// The value of subset[i%2][j] will be true

// if there exists a subset of sum j in

// arr[0, 1, ...., i-1]

bool subset[2][sum + 1];

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

for (int j = 0; j <= sum; j++) {

// A subset with sum 0 is always possible

if (j == 0)

subset[i % 2][j] = true;

// If there exists no element no sum

// is possible

else if (i == 0)

subset[i % 2][j] = false;

else if (arr[i - 1] <= j)

subset[i % 2][j] = subset[(i + 1) % 2]

[j - arr[i - 1]] || subset[(i + 1) % 2][j];

else

subset[i % 2][j] = subset[(i + 1) % 2][j];

}

}

return subset[n % 2][sum];

}

// Driver code

int main()

{

int arr[] = { 6, 2, 5 };

int sum = 7;

int n = sizeof(arr) / sizeof(arr[0]);

if (isSubsetSum(arr, n, sum) == true)

cout <<"There exists a subset with given sum";

else

cout <<"No subset exists with given sum";

return 0;

}

**Output**

There exists a subset with given sum

**Another Approach:** To further reduce space complexity, we create a boolean 1D array subset[sum+1]. Using bottom up manner we can fill up this table. The idea is that we can check if the sum till position “i” is possible then if the current element in the array at position j is x, then sum i+x is also possible. We traverse the sum array from back to front so that we don’t count any element twice.

Here’s the code for the given approach:

#include <iostream>

using namespace std;

bool isPossible(int elements[], int sum, int n)

{

int dp[sum + 1];

// Initializing with 1 as sum 0 is

// always possible

dp[0] = 1;

// Loop to go through every element of

// the elements array

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

{

// To change the values of all possible sum

// values to 1

for(int j = sum; j >= elements[i]; j--)

{

if (dp[j - elements[i]] == 1)

dp[j] = 1;

}

}

// If sum is possible then return 1

if (dp[sum] == 1)

return true;

return false;

}

// Driver code

int main()

{

int elements[] = { 6, 2, 5 };

int n = sizeof(elements) / sizeof(elements[0]);

int sum = 7;

if (isPossible(elements, sum, n))

cout << ("YES");

else

cout << ("NO");

return 0;

}

**Output**

YES

**Time Complexity:** O(N\*K) where N is the number of elements in the array and K is total sum.  
**Space Complexity:** O(K)

# 419. Longest Palindromic Subsequence

Given a String, find the longest palindromic subsequence.

**Example 1:**

**Input:**

S = "bbabcbcab"

**Output:** 7

**Explanation**: Subsequence "babcbab" is the

longest subsequence which is also a palindrome.

**Example 2:**

**Input**:

S = "abcd"

**Output:** 1

**Explanation**: "a", "b", "c" and "d" are

palindromic and all have a length 1.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **longestPalinSubseq()**which takes the string S as input and returns an integer denoting the length of the longest palindromic subsequence of S.

**Expected Time Complexity:**O(|S|\*|S|).  
**Expected Auxiliary Space:**O(|S|\*|S|).

**Constraints:**  
1 ≤ |S| ≤ 1000

## Solution:

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. 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:**  
Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].   
If last and first characters of X are same, then L(0, n-1) = L(1, n-2) + 2.   
Else L(0, n-1) = MAX (L(1, n-1), L(0, n-2)).

Following is a general recursive solution with all cases handled. 

// Every single character is a palindrome of length 1

L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same

If (X[i] != X[j]) L(i, j) = max{L(i + 1, j),L(i, j - 1)}

// If there are only 2 characters and both are same

Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last

// characters are same

Else L(i, j) = L(i + 1, j - 1) + 2

**2) Overlapping Subproblems**   
Following is a simple recursive implementation of the LPS problem. The implementation simply follows the recursive structure mentioned above.

// C++ program of above approach

#include<bits/stdc++.h>

using namespace std;

// A utility function to get max of two integers

int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq

int lps(char \*seq, int i, int j)

{

// Base Case 1: If there is only 1 character

if (i == j)

return 1;

// Base Case 2: If there are only 2

// characters and both are same

if (seq[i] == seq[j] && i + 1 == j)

return 2;

// If the first and last characters match

if (seq[i] == seq[j])

return lps (seq, i+1, j-1) + 2;

// If the first and last characters do not match

return max( lps(seq, i, j-1), lps(seq, i+1, j) );

}

/\* Driver program to test above functions \*/

int main()

{

char seq[] = "GEEKSFORGEEKS";

int n = strlen(seq);

cout << "The length of the LPS is "

<< lps(seq, 0, n-1);

return 0;

}

**Output**

The length of the LPS is 5

Considering the above implementation, the following is a partial recursion tree for a sequence of length 6 with all different characters. 

L(0, 5)

/ \

/ \

L(1,5) L(0,4)

/ \ / \

/ \ / \

L(2,5) L(1,4) L(1,4) L(0,3)

In the above partial recursion tree, L(1, 4) is being solved twice. If we draw the complete recursion tree, then 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 LPS problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of the same subproblems can be avoided by constructing a temporary array L[][] in a bottom-up manner.

**Dynamic Programming Solution**

// A Dynamic Programming based C++ program for LPS problem

// Returns the length of the longest palindromic subsequence in seq

#include<stdio.h>

#include<string.h>

// A utility function to get max of two integers

int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq

int lps(char \*str)

{

int n = strlen(str);

int i, j, cl;

int L[n][n]; // Create a table to store results of subproblems

// Strings of length 1 are palindrome of length 1

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

L[i][i] = 1;

// Build the table. Note that the lower diagonal values of table are

// useless and not filled in the process. The values are filled in a

// manner similar to Matrix Chain Multiplication DP solution (See

// https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/). cl is length of

// substring

for (cl=2; cl<=n; cl++)

{

for (i=0; i<n-cl+1; i++)

{

j = i+cl-1;

if (str[i] == str[j] && cl == 2)

L[i][j] = 2;

else if (str[i] == str[j])

L[i][j] = L[i+1][j-1] + 2;

else

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

}

}

return L[0][n-1];

}

/\* Driver program to test above functions \*/

int main()

{

char seq[] = "GEEKS FOR GEEKS";

int n = strlen(seq);

printf ("The length of the LPS is %d", lps(seq));

getchar();

return 0;

}

**Output**

The length of the LPS is 7

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.

**Using Memoization Technique of Dynamic programming**

The idea used here is to reverse the given input string and check the length of the [longest common subsequence](https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/). That would be the answer for the longest palindromic subsequence.

// A Dynamic Programming based C++ program for LPS problem

// Returns the length of the longest palindromic subsequence

// in seq

#include <bits/stdc++.h>

using namespace std;

int dp[1001][1001];

// Returns the length of the longest palindromic subsequence

// in seq

int lps(string& s1, string& s2, int n1, int n2)

{

if (n1 == 0 || n2 == 0) {

return 0;

}

if (dp[n1][n2] != -1) {

return dp[n1][n2];

}

if (s1[n1 - 1] == s2[n2 - 1]) {

return dp[n1][n2] = 1 + lps(s1, s2, n1 - 1, n2 - 1);

}

else {

return dp[n1][n2] = max(lps(s1, s2, n1 - 1, n2),

lps(s1, s2, n1, n2 - 1));

}

}

/\* Driver program to test above functions \*/

int main()

{

string seq = "GEEKS FOR GEEKS";

int n = seq.size();

dp[n][n];

memset(dp, -1, sizeof(dp));

string s2 = seq;

reverse(s2.begin(), s2.end());

cout << "The length of the LPS is "

<< lps(s2, seq, n, n) << endl;

return 0;

}

**Output**

The length of the LPS is 7

**Time Complexity:** O(n\*n)

# 420. Longest Palindromic Substring

Given a string str of length N, you have to find number of palindromic subsequence (need not necessarily be distinct) which could be formed from the string str.  
Note: You have to return the answer module 109+7;

**Example 1:**

**Input:**

Str = "abcd"

**Output:**

4

**Explanation:**

palindromic subsequence are : "a" ,"b", "c" ,"d"

**Example 2:**

**Input:**

Str = "aab"

**Output:**

4

**Explanation:**

palindromic subsequence are :"a", "a", "b", "aa"

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **countPs()** which takes a string str as input parameter and returns the number of palindromic subsequence.

**Expected Time Complexity:** O(N\*N)  
**Expected Auxiliary Space:** O(N\*N)

**Constraints:**  
1<=length of string str <=1000

## Solution:

The above problem can be recursively defined.

Initial Values : i= 0, j= n-1;

CountPS(i,j)

// Every single character of a string is a palindrome

// subsequence

if i == j

return 1 // palindrome of length 1

// If first and last characters are same, then we

// consider it as palindrome subsequence and check

// for the rest subsequence (i+1, j), (i, j-1)

Else if (str[i] == str[j)]

return countPS(i+1, j) + countPS(i, j-1) + 1;

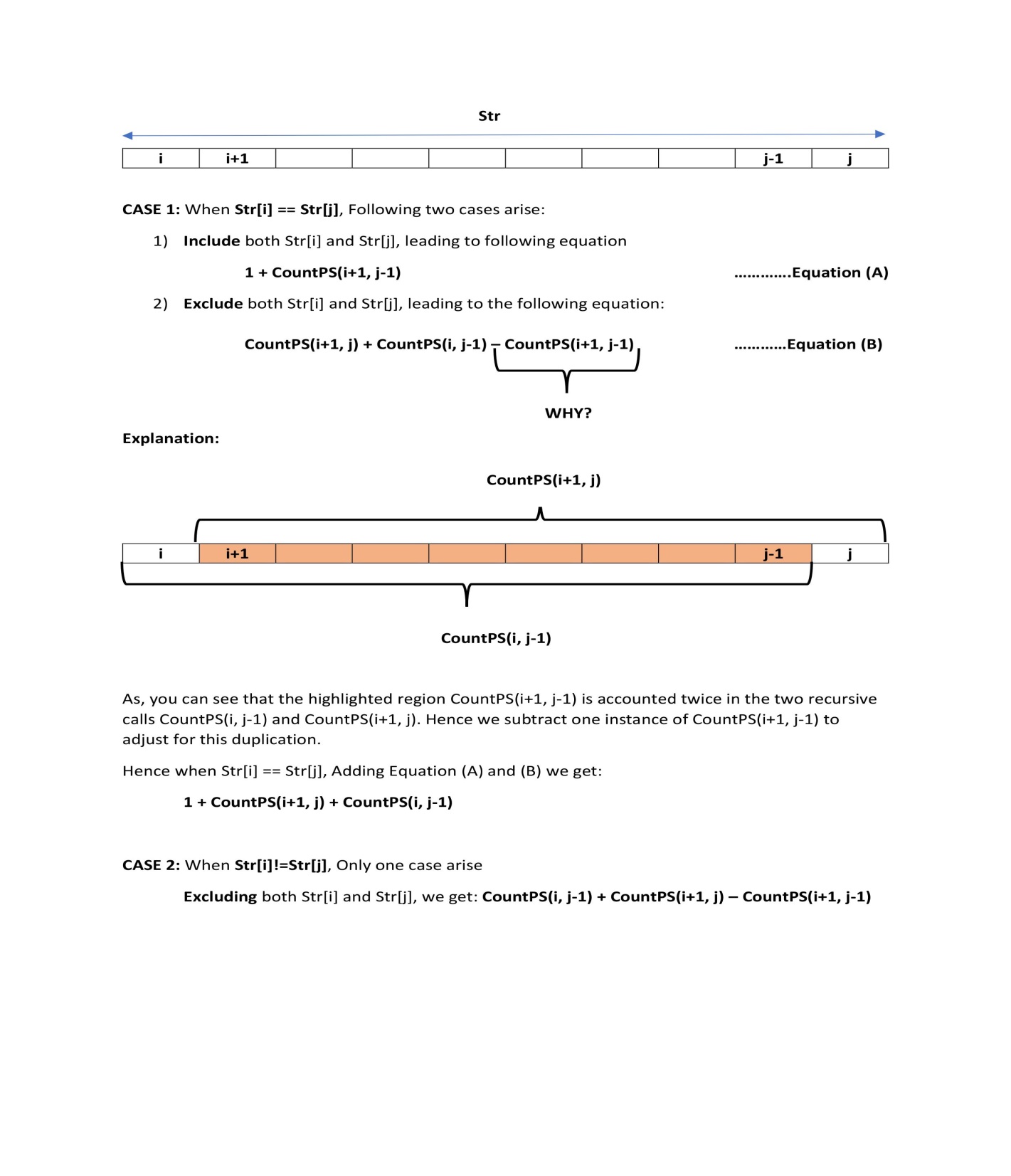
else

// check for rest sub-sequence and remove common

// palindromic subsequences as they are counted

// twice when we do countPS(i+1, j) + countPS(i,j-1)

return countPS(i+1, j) + countPS(i, j-1) - countPS(i+1, j-1)



If we draw recursion tree of above recursive solution, we can observe [overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/). Since the problem has overlapping subproblems, we can solve it efficiently using Dynamic Programming. Below is Dynamic Programming based solution.

// Counts Palindromic Subsequence in a given String

#include <cstring>

#include <iostream>

using namespace std;

// Function return the total palindromic subsequence

int countPS(string str)

{

int N = str.length();

// create a 2D array to store the count of palindromic

// subsequence

int cps[N + 1][N + 1];

memset(cps, 0, sizeof(cps));

// palindromic subsequence of length 1

for (int i = 0; i < N; i++)

cps[i][i] = 1;

// check subsequence of length L is palindrome or not

for (int L = 2; L <= N; L++) {

for (int i = 0; i <= N-L; i++) {

int k = L + i - 1;

if (str[i] == str[k])

cps[i][k]

= cps[i][k - 1] + cps[i + 1][k] + 1;

else

cps[i][k] = cps[i][k - 1] + cps[i + 1][k]

- cps[i + 1][k - 1];

}

}

// return total palindromic subsequence

return cps[0][N - 1];

}

// Driver program

int main()

{

string str = "abcb";

cout << "Total palindromic subsequence are : "

<< countPS(str) << endl;

return 0;

}

**Output:**

Total palindromic subsequence are : 6

**Time Complexity :** O(N2)

**Another approach:**(Using recursion)

// C++ program to counts Palindromic Subsequence

// in a given String using recursion

#include <bits/stdc++.h>

using namespace std;

int n, dp[1000][1000];

string str = "abcb";

// Function return the total

// palindromic subsequence

int countPS(int i, int j)

{

if (i > j)

return 0;

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

return dp[i][j];

if (i == j)

return dp[i][j] = 1;

else if (str[i] == str[j])

return dp[i][j]

= countPS(i + 1, j) +

countPS(i, j - 1) + 1;

else

return dp[i][j] = countPS(i + 1, j) +

countPS(i, j - 1) -

countPS(i + 1, j - 1);

}

// Driver code

int main()

{

memset(dp, -1, sizeof(dp));

n = str.size();

cout << "Total palindromic subsequence are : "

<< countPS(0, n - 1) << endl;

return 0;

}

**Output:**

Total palindromic subsequence are : 6

# 421. Longest Palindromic Substring

Given a string s, return *the longest palindromic substring* in s.

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Explanation:** "aba" is also a valid answer.

**Example 2:**

**Input:** s = "cbbd"

**Output:** "bb"

**Constraints:**

* 1 <= s.length <= 1000
* s consist of only digits and English letters.

## Solution:

**Approach 1: Longest Common Substring**

**Common mistake**

Some people will be tempted to come up with a quick solution, which is unfortunately flawed (however can be corrected easily):

Reverse S*S* and become S'*S*′. Find the longest common substring between S*S* and S'*S*′, which must also be the longest palindromic substring.

This seemed to work, let’s see some examples below.

For example, S*S* = "caba", S'*S*′ = "abac".

The longest common substring between S*S* and S'*S*′ is "aba", which is the answer.

Let’s try another example: S*S* = "abacdfgdcaba", S'*S*′ = "abacdgfdcaba".

The longest common substring between S*S* and S'*S*′ is "abacd". Clearly, this is not a valid palindrome.

**Algorithm**

We could see that the longest common substring method fails when there exists a reversed copy of a non-palindromic substring in some other part of S*S*. To rectify this, each time we find a longest common substring candidate, we check if the substring’s indices are the same as the reversed substring’s original indices. If it is, then we attempt to update the longest palindrome found so far; if not, we skip this and find the next candidate.

This gives us an O(n^2)*O*(*n*2) Dynamic Programming solution which uses O(n^2)*O*(*n*2) space (could be improved to use O(n)*O*(*n*) space). Please read more about Longest Common Substring [here](http://en.wikipedia.org/wiki/Longest_common_substring).

**Approach 2: Brute Force**

The obvious brute force solution is to pick all possible starting and ending positions for a substring, and verify if it is a palindrome.

**Complexity Analysis**

* Time complexity : O(n^3)*O*(*n*3). Assume that n*n* is the length of the input string, there are a total of \binom{n}{2} = \frac{n(n-1)}{2}(2*n*​)=2*n*(*n*−1)​ such substrings (excluding the trivial solution where a character itself is a palindrome). Since verifying each substring takes O(n)*O*(*n*) time, the run time complexity is O(n^3)*O*(*n*3).
* Space complexity : O(1)*O*(1).

**Approach 3: Dynamic Programming**

To improve over the brute force solution, we first observe how we can avoid unnecessary re-computation while validating palindromes. Consider the case "ababa". If we already knew that "bab" is a palindrome, it is obvious that "ababa" must be a palindrome since the two left and right end letters are the same.

We define P(i,j)*P*(*i*,*j*) as following:

P(i,j) = \begin{cases} \text{true,} &\quad\text{if the substring } S\_i \dots S\_j \text{ is a palindrome}\\ \text{false,} &\quad\text{otherwise.} \end{cases}*P*(*i*,*j*)={true,false,​if the substring *Si*​…*Sj*​ is a palindromeotherwise.​

Therefore,

P(i, j) = ( P(i+1, j-1) \text{ and } S\_i == S\_j )*P*(*i*,*j*)=(*P*(*i*+1,*j*−1) and *Si*​==*Sj*​)

The base cases are:

P(i, i) = true*P*(*i*,*i*)=*true*

P(i, i+1) = ( S\_i == S\_{i+1} )*P*(*i*,*i*+1)=(*Si*​==*Si*+1​)

This yields a straight forward DP solution, which we first initialize the one and two letters palindromes, and work our way up finding all three letters palindromes, and so on...

string longestPalindrome(string s) {

int n = s.size(), max = 1, x = 0;

bool dp[n][n];

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

dp[i][i] = true;

for(int l=2;l<=n;l++){

for(int i=0;i<n-l+1;i++){

int j = i+l-1;

if(s[i]!=s[j]){

dp[i][j] = false;

}

else{

if(l>2 && dp[i+1][j-1]==false)

dp[i][j] = false;

else{

if(max<l){

max = l;

x = i;

}

dp[i][j] = true;

}

}

}

}

return s.substr(x, max);

}

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). This gives us a runtime complexity of O(n^2)*O*(*n*2).
* Space complexity : O(n^2)*O*(*n*2). It uses O(n^2)*O*(*n*2) space to store the table.

**Approach 4: Expand Around Center**

In fact, we could solve it in O(n^2)*O*(*n*2) time using only constant space.

We observe that a palindrome mirrors around its center. Therefore, a palindrome can be expanded from its center, and there are only 2n - 12*n*−1 such centers.

You might be asking why there are 2n - 12*n*−1 but not n*n* centers? The reason is the center of a palindrome can be in between two letters. Such palindromes have even number of letters (such as "abba") and its center are between the two 'b's.

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) return "";

int start = 0, end = 0;

for (int i = 0; i < s.length(); i++) {

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

int L = left, R = right;

while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {

L--;

R++;

}

return R - L - 1;

}

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). Since expanding a palindrome around its center could take O(n)*O*(*n*) time, the overall complexity is O(n^2)*O*(*n*2).
* Space complexity : O(1)*O*(1).

# 422. Longest alternating subsequence

A sequence {x1, x2, .. xn} is alternating sequence if its elements satisfy one of the following relations :  
x1 < x2 > x3 < x4 > x5..... or  x1 >x2 < x3 > x4 < x5.....  
Your task is to find the longest such sequence.  
  
**Example 1:**

**Input:** nums = {1,5,4}

**Output:** 3

**Explanation:** The entire sequenece is a

alternating sequence.

**Examplae 2:**

**Input:** nums = {1,17,5,10,13,15,10,5,16,8}

**Output:** 7

**Explanation:** There are several subsequences

that achieve this length.

One is {1,17,10,13,10,16,8}.

**Your Task:**  
You don't need to read or print anyhting. Your task is to complete the function **AlternatingaMaxLength()**which takes the sequence  nums as input parameter and returns the maximum length of alternating sequence.  
  
**Expected Time Complexity:**O(n)  
**Expected Space Complexity:**O(1)  
  
**Constraints:**  
1 <= n <= 105

## Solution:

This problem is an extension of [longest increasing subsequence problem](https://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/), but requires more thinking for finding optimal substructure property in this.  
We will solve this problem by dynamic Programming method, Let A is given array of length n of integers. We define a 2D array las[n][2] such that las[i][0] contains longest alternating subsequence ending at index i and last element is greater than its previous element and las[i][1] contains longest alternating subsequence ending at index i and last element is smaller than its previous element, then we have following recurrence relation between them,

**las[i][0]** = Length of the longest alternating subsequence

ending at index i and last element is greater

than its previous element

**las[i][1]** = Length of the longest alternating subsequence

ending at index i and last element is smaller

than its previous element

**Recursive Formulation:**

las[i][0] = max (las[i][0], las[j][1] + 1);

for all j < i and A[j] < A[i]

las[i][1] = max (las[i][1], las[j][0] + 1);

for all j < i and A[j] > A[i]

The first recurrence relation is based on the fact that, If we are at position i and this element has to bigger than its previous element then for this sequence (upto i) to be bigger we will try to choose an element j ( < i) such that A[j] < A[i] i.e. A[j] can become A[i]’s previous element and las[j][1] + 1 is bigger than las[i][0] then we will update las[i][0].   
Remember we have chosen las[j][1] + 1 not las[j][0] + 1 to satisfy alternate property because in las[j][0] last element is bigger than its previous one and A[i] is greater than A[j] which will break the alternating property if we update. So above fact derives first recurrence relation, similar argument can be made for second recurrence relation also.

// C++ program to find longest alternating

// subsequence in an array

#include<iostream>

using namespace std;

// Function to return max of two numbers

int max(int a, int b)

{

return (a > b) ? a : b;

}

// Function to return longest alternating

// subsequence length

int zzis(int arr[], int n)

{

/\*las[i][0] = Length of the longest

alternating subsequence ending at

index i and last element is greater

than its previous element

las[i][1] = Length of the longest

alternating subsequence ending

at index i and last element is

smaller than its previous element \*/

int las[n][2];

// Initialize all values from 1

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

las[i][0] = las[i][1] = 1;

// Initialize result

int res = 1;

// Compute values in bottom up manner

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

{

// Consider all elements as

// previous of arr[i]

for(int j = 0; j < i; j++)

{

// If arr[i] is greater, then

// check with las[j][1]

if (arr[j] < arr[i] &&

las[i][0] < las[j][1] + 1)

las[i][0] = las[j][1] + 1;

// If arr[i] is smaller, then

// check with las[j][0]

if(arr[j] > arr[i] &&

las[i][1] < las[j][0] + 1)

las[i][1] = las[j][0] + 1;

}

// Pick maximum of both values at index i

if (res < max(las[i][0], las[i][1]))

res = max(las[i][0], las[i][1]);

}

return res;

}

// Driver code

int main()

{

int arr[] = { 10, 22, 9, 33,

49, 50, 31, 60 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Length of Longest alternating "

<< "subsequence is " << zzis(arr, n);

return 0;

}

**Output:**

Length of Longest alternating subsequence is 6

**Time Complexity:** O(n2)   
**Auxiliary Space:** O(n)

**Efficient Solution:**  
In the above approach, at any moment we are keeping track of two values (Length of the longest alternating subsequence ending at index i, and last element is smaller than or greater than previous element), for every element on array. To optimise space, we only need to store two variables for element at any index i.

inc = Length of longest alternative subsequence so far with current value being greater than it’s previous value.  
dec = Length of longest alternative subsequence so far with current value being smaller than it’s previous value.  
The tricky part of this approach is to update these two values.

“inc” should be increased, if and only if the last element in the alternative sequence was smaller than it’s previous element.  
“dec” should be increased, if and only if the last element in the alternative sequence was greater than it’s previous element.

// C++ program for above approach

#include <bits/stdc++.h>

using namespace std;

// Function for finding

// longest alternating

// subsequence

int LAS(int arr[], int n)

{

// "inc" and "dec" initialized as 1

// as single element is still LAS

int inc = 1;

int dec = 1;

// Iterate from second element

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

{

if (arr[i] > arr[i - 1])

{

// "inc" changes iff "dec"

// changes

inc = dec + 1;

}

else if (arr[i] < arr[i - 1])

{

// "dec" changes iff "inc"

// changes

dec = inc + 1;

}

}

// Return the maximum length

return max(inc, dec);

}

// Driver Code

int main()

{

int arr[] = { 10, 22, 9, 33, 49,

50, 31, 60 };

int n = sizeof(arr) / sizeof(arr[0]);

// Function Call

cout << LAS(arr, n) << endl;

return 0;

}

**Output:**

6

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

# 423. Weighted Job Scheduling

Given N jobs where every job is represented by following three elements of it.

1. Start Time
2. Finish Time
3. Profit or Value Associated (>= 0)

Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

**Example:**

Input: Number of Jobs n = 4

Job Details {Start Time, Finish Time, Profit}

Job 1: {1, 2, 50}

Job 2: {3, 5, 20}

Job 3: {6, 19, 100}

Job 4: {2, 100, 200}

Output: The maximum profit is 250.

We can get the maximum profit by scheduling jobs 1 and 4.

Note that there is longer schedules possible Jobs 1, 2 and 3

but the profit with this schedule is 20+50+100 which is less than 2

## Solution:

A simple version of this problem is discussed [here](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)where every job has the same profit or value. The [Greedy Strategy for activity selection](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/) doesn’t work here as a schedule with more jobs may have smaller profit or value.

The above problem can be solved using the following recursive solution.

**1)** First sort jobs according to finish time.

**2)** Now apply following recursive process.

// Here arr[] is array of n jobs

findMaximumProfit(arr[], n)

{

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

b) Return the maximum of following two profits.

(i) Maximum profit by excluding current job, i.e.,

findMaximumProfit(arr, n-1)

(ii) Maximum profit by including the current job

}

**How to find the profit including current job?**

The idea is to find the latest job before the current job (in

sorted array) that doesn't conflict with current job 'arr[n-1]'.

Once we find such a job, we recur for all jobs till that job and

add profit of current job to result.

In the above example, "job 1" is the latest non-conflicting

for "job 4" and "job 2" is the latest non-conflicting for "job 3".

The following is the implementation of the above naive recursive method.

// C++ program for weighted job scheduling using Naive Recursive Method

#include <iostream>

#include <algorithm>

using namespace std;

// A job has start time, finish time and profit.

struct Job

{

int start, finish, profit;

};

// A utility function that is used for sorting events

// according to finish time

bool jobComparator(Job s1, Job s2)

{

return (s1.finish < s2.finish);

}

// Find the latest job (in sorted array) that doesn't

// conflict with the job[i]. If there is no compatible job,

// then it returns -1.

int latestNonConflict(Job arr[], int i)

{

for (int j=i-1; j>=0; j--)

{

if (arr[j].finish <= arr[i-1].start)

return j;

}

return -1;

}

// A recursive function that returns the maximum possible

// profit from given array of jobs. The array of jobs must

// be sorted according to finish time.

int findMaxProfitRec(Job arr[], int n)

{

// Base case

if (n == 1) return arr[n-1].profit;

// Find profit when current job is included

int inclProf = arr[n-1].profit;

int i = latestNonConflict(arr, n);

if (i != -1)

inclProf += findMaxProfitRec(arr, i+1);

// Find profit when current job is excluded

int exclProf = findMaxProfitRec(arr, n-1);

return max(inclProf, exclProf);

}

// The main function that returns the maximum possible

// profit from given array of jobs

int findMaxProfit(Job arr[], int n)

{

// Sort jobs according to finish time

sort(arr, arr+n, jobComparator);

return findMaxProfitRec(arr, n);

}

// Driver program

int main()

{

Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "The optimal profit is " << findMaxProfit(arr, n);

return 0;

}

**Output:**

The optimal profit is 250

The above solution may contain many overlapping subproblems. For example, if lastNonConflicting() always returns the previous job, then findMaxProfitRec(arr, n-1) is called twice and the time complexity becomes O(n\*2n). As another example when lastNonConflicting() returns previous to the previous job, there are two recursive calls, for n-2 and n-1. In this example case, recursion becomes the same as Fibonacci Numbers.

So this problem has both properties of Dynamic Programming, [Optimal Substructure](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/),and [Overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/).   
Like other Dynamic Programming Problems, we can solve this problem by making a table that stores solutions of subproblems.

Below is an implementation based on Dynamic Programming.

// C++ program for weighted job scheduling using Dynamic

// Programming.

#include <algorithm>

#include <iostream>

using namespace std;

// A job has start time, finish time and profit.

struct Job {

int start, finish, profit;

};

// A utility function that is used for sorting events

// according to finish time

bool jobComparator(Job s1, Job s2)

{

return (s1.finish < s2.finish);

}

// Find the latest job (in sorted array) that doesn't

// conflict with the job[i]

int latestNonConflict(Job arr[], int i)

{

for (int j = i - 1; j >= 0; j--) {

if (arr[j].finish <= arr[i].start)

return j;

}

return -1;

}

// The main function that returns the maximum possible

// profit from given array of jobs

int findMaxProfit(Job arr[], int n)

{

// Sort jobs according to finish time

sort(arr, arr + n, jobComparator);

// Create an array to store solutions of subproblems.

// table[i] stores the profit for jobs till arr[i]

// (including arr[i])

int\* table = new int[n];

table[0] = arr[0].profit;

// Fill entries in M[] using recursive property

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

// Find profit including the current job

int inclProf = arr[i].profit;

int l = latestNonConflict(arr, i);

if (l != -1)

inclProf += table[l];

// Store maximum of including and excluding

table[i] = max(inclProf, table[i - 1]);

}

// Store result and free dynamic memory allocated for

// table[]

int result = table[n - 1];

delete[] table;

return result;

}

// Driver program

int main()

{

Job arr[] = { { 3, 10, 20 },

{ 1, 2, 50 },

{ 6, 19, 100 },

{ 2, 100, 200 } };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "The optimal profit is "

<< findMaxProfit(arr, n);

return 0;

}

**Output:**

The optimal profit is 250

Time Complexity of the above Dynamic Programming Solution is O(n2). Note that the above solution can be optimized to O(nLogn) using Binary Search in latestNonConflict() instead of linear search. Thanks to Garvit for suggesting this optimization.

We have discussed recursive and Dynamic Programming based approaches in the [previous article](https://www.geeksforgeeks.org/weighted-job-scheduling/). The implementations discussed in above post uses linear search to find the previous non-conflicting job. In this post, Binary Search based solution is discussed. The time complexity of Binary Search based solution is O(n Log n).  
The algorithm is: 

1. Sort the jobs by non-decreasing finish times.
2. For each i from 1 to n, determine the maximum value of the schedule from the subsequence of jobs[0..i]. Do this by comparing the inclusion of job[i] to the schedule to the exclusion of job[i] to the schedule, and then taking the max.

To find the profit with inclusion of job[i]. we need to find the latest job that doesn’t conflict with job[i]. The idea is to use Binary Search to find the latest non-conflicting job.

// C++ program for weighted job scheduling using Dynamic

// Programming and Binary Search

#include <iostream>

#include <algorithm>

using namespace std;

// A job has start time, finish time and profit.

struct Job

{

int start, finish, profit;

};

// A utility function that is used for sorting events

// according to finish time

bool myfunction(Job s1, Job s2)

{

return (s1.finish < s2.finish);

}

// A Binary Search based function to find the latest job

// (before current job) that doesn't conflict with current

// job. "index" is index of the current job. This function

// returns -1 if all jobs before index conflict with it.

// The array jobs[] is sorted in increasing order of finish

// time.

int binarySearch(Job jobs[], int index)

{

// Initialize 'lo' and 'hi' for Binary Search

int lo = 0, hi = index - 1;

// Perform binary Search iteratively

while (lo <= hi)

{

int mid = (lo + hi) / 2;

if (jobs[mid].finish <= jobs[index].start)

{

if (jobs[mid + 1].finish <= jobs[index].start)

lo = mid + 1;

else

return mid;

}

else

hi = mid - 1;

}

return -1;

}

// The main function that returns the maximum possible

// profit from given array of jobs

int findMaxProfit(Job arr[], int n)

{

// Sort jobs according to finish time

sort(arr, arr+n, myfunction);

// Create an array to store solutions of subproblems. table[i]

// stores the profit for jobs till arr[i] (including arr[i])

int \*table = new int[n];

table[0] = arr[0].profit;

// Fill entries in table[] using recursive property

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

{

// Find profit including the current job

int inclProf = arr[i].profit;

int l = binarySearch(arr, i);

if (l != -1)

inclProf += table[l];

// Store maximum of including and excluding

table[i] = max(inclProf, table[i-1]);

}

// Store result and free dynamic memory allocated for table[]

int result = table[n-1];

delete[] table;

return result;

}

// Driver program

int main()

{

Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Optimal profit is " << findMaxProfit(arr, n);

return 0;

}

Output:

Optimal profit is 250

# 424. Coin game winner where every player has three choices

Given three numbers **N**, **X**, and **Y**. Geek and his friend playing a coin game. At the beginning, there are**N** coins. In each move, a player can pick **X,** **Y**, or **1** coin. Geek always starts the game. The player who picks the last coin wins the game. The task is to check whether Geek will win the game or not if both are playing optimally.

**Example 1:**

**Input**: N = 5, X = 3, Y = 4

**Output:** 1

**Explanation**: There are 5 coins, every

player can pick 1 or 3 or 4 coins on

his/her turn. Geek can win by picking

3 coins in first chance. Now 2 coins

will be left so his frined will pick

one coin and now Geek can win by

picking the last coin.

**Example 2:**

**Input**: N = 2, X = 3, Y = 4

**Output:** 0

**Explanation**: Geek picks 1 and then

his friend picks 1

**Your Task:**  
You don't need to read input or print anything. Complete the function **findWinner()**which takes **N, X,**and**Y**as input parameters and returns 1 if Geek can win otherwise 0.  
  
**Expected Time Complexity:** O(**|N|**)  
**Expected Auxiliary Space:** O(**|N|**)  
  
**Constraints:**  
1 ≤ **|N|** ≤ 106

## Solution:

Let us take few example values of n for x = 3, y = 4.   
n = 0 A can not pick any coin so he losses   
n = 1 A can pick 1 coin and win the game   
n = 2 A can pick only 1 coin. Now B will pick 1 coin and win the game   
n = 3 4 A will win the game by picking 3 or 4 coins   
n = 5, 6 A will choose 3 or 4 coins. Now B will have to choose from 2 coins so A will win.  
We can observe that A wins game for n coins only when B loses for coins n-1 or n-x or n-y.

// C++ program to find winner of game

// if player can pick 1, x, y coins

#include <bits/stdc++.h>

using namespace std;

// To find winner of game

bool findWinner(int x, int y, int n)

{

// To store results

int dp[n + 1];

// Initial values

dp[0] = false;

dp[1] = true;

// Computing other values.

for (int i = 2; i <= n; i++) {

// If A losses any of i-1 or i-x

// or i-y game then he will

// definitely win game i

if (i - 1 >= 0 and !dp[i - 1])

dp[i] = true;

else if (i - x >= 0 and !dp[i - x])

dp[i] = true;

else if (i - y >= 0 and !dp[i - y])

dp[i] = true;

// Else A loses game.

else

dp[i] = false;

}

// If dp[n] is true then A will

// game otherwise he losses

return dp[n];

}

// Driver program to test findWinner();

int main()

{

int x = 3, y = 4, n = 5;

if (findWinner(x, y, n))

cout << 'A';

else

cout << 'B';

return 0;

}

**Output:**

A

# 425. Count Derangements (Permutation such that no element appears in its original position) [ IMPORTANT ]

A Derangement is a permutation of n elements, such that no element appears in its original position. For example, a derangement of {0, 1, 2, 3} is {2, 3, 1, 0}.  
Given a number n, find the total number of Derangements of a set of n elements.

**Examples :**

**Input:** n = 2

**Output:** 1

For two elements say {0, 1}, there is only one

possible derangement {1, 0}

**Input:** n = 3

**Output:** 2

For three elements say {0, 1, 2}, there are two

possible derangements {2, 0, 1} and {1, 2, 0}

**Input:** n = 4

**Output:** 9

For four elements say {0, 1, 2, 3}, there are 9

possible derangements {1, 0, 3, 2} {1, 2, 3, 0}

{1, 3, 0, 2}, {2, 3, 0, 1}, {2, 0, 3, 1}, {2, 3,

1, 0}, {3, 0, 1, 2}, {3, 2, 0, 1} and {3, 2, 1, 0}

## Solution:

Let countDer(n) be count of derangements for n elements. Below is the recursive relation to it.

countDer(n) = (n - 1) \* [countDer(n - 1) + countDer(n - 2)]

**How does above recursive relation work?**

There are n – 1 way for element 0 (this explains multiplication with n – 1).   
Let ***0 be placed at index i***. There are now two possibilities, depending on whether or not element i is placed at 0 in return.

1. ***i is placed at 0:*** This case is equivalent to solving the problem for n-2 elements as two elements have just swapped their positions.
2. ***i is not placed at 0:***This case is equivalent to solving the problem for n-1 elements as now there are n-1 elements, n-1 positions and every element has n-2 choices

Below is the simple solution based on the above recursive formula:

// A Naive Recursive C++ program

// to count derangements

#include <bits/stdc++.h>

using namespace std;

int countDer(int n)

{

// Base cases

if (n == 1) return 0;

if (n == 2) return 1;

// countDer(n) = (n-1)[countDer(n-1) + der(n-2)]

return (n - 1) \* (countDer(n - 1) + countDer(n - 2));

}

// Driver Code

int main()

{

int n = 4;

cout << "Count of Derangements is "

<< countDer(n);

return 0;

}

**Output**

Count of Derangements is 9

**Time Complexity:**O(2^n) since **T(n) = T(n-1) + T(n-2)** which is exponential.

***Auxiliary Space:*** O(h) where **h= log n**is the maximum height of the tree.

We can observe that this implementation does repetitive work. For example, see recursion tree for countDer(5), countDer(3) is being evaluated twice.

cdr() ==> countDer()

cdr(5)

/ \

cdr(4) cdr(3)

/ \ / \

cdr(3) cdr(2) cdr(2) cdr(1)

An **Efficient Solution** is to use Dynamic Programming to store results of subproblems in an array and build the array in bottom-up manner.

// A Dynamic programming based C++

// program to count derangements

#include <bits/stdc++.h>

using namespace std;

int countDer(int n)

{

// Create an array to store

// counts for subproblems

int der[n + 1] = {0};

// Base cases

der[1] = 0;

der[2] = 1;

// Fill der[0..n] in bottom up manner

// using above recursive formula

for (int i = 3; i <= n; ++i)

der[i] = (i - 1) \* (der[i - 1] +

der[i - 2]);

// Return result for n

return der[n];

}

// Driver code

int main()

{

int n = 4;

cout << "Count of Derangements is "

<< countDer(n);

return 0;

}

**Output**

Count of Derangements is 9

***Time Complexity :****O(n)*  
***Auxiliary Space :****O(n)*  
Thanks to Utkarsh Trivedi for suggesting the above solution.

A **More** **Efficient** Solution Without using **Extra Space**.

As we only need to remember only two previous values So, instead of Storing the values in an array two variables can be used to just store the required previous only.

Below is the implementation of the above approach:

// C++ implementation of the above

// approach

#include <iostream>

using namespace std;

int countDer(int n)

{

// base case

if (n == 1 or n == 2) {

return n - 1;

}

// Variable for just storing

// previous values

int a = 0;

int b = 1;

// using above recursive formula

for (int i = 3; i <= n; ++i) {

int cur = (i - 1) \* (a + b);

a = b;

b = cur;

}

// Return result for n

return b;

}

// Driver Code

int main()

{

cout << "Count of Derangements is " << countDer(4);

return 0;

}

**Output**

Count of Derangements is 9

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