# 426. Maximum profit by buying and selling a share at most twice [ IMP ]

In daily share trading, a buyer buys shares in the morning and sells them on the same day. If the trader is allowed to make at most 2 transactions in a day, whereas the second transaction can only start after the first one is complete (Buy->sell->Buy->sell). The stock prices throughout the day  are represented in the form of an array **price**.

Given an array **price** of size **N**, find out the **maximum** profit that a share trader could have made.

**Example 1:**

**Input:**

6

10 22 5 75 65 80

**Output:**

87

**Explanation:**

Trader earns 87 as sum of 12, 75

Buy at 10, sell at 22,

Buy at 5 and sell at 80

**Example 2:**

**Input:**

7

2 30 15 10 8 25 80

**Output:**

100

**Explanation:**

Trader earns 100 as sum of 28 and 72

Buy at price 2, sell at 30,

Buy at 8 and sell at 80

**Your Task:**

Complete the function **maxProfit()** which takes an integer array **price** as the only argument and returns an integer, representing the maximum profit, if only two transactions are allowed.

**Expected Time Complexity:** O(N)

**Expected Space Complexity:** O(1)

**Constraints:**

* 1 <= N <= 105
* 1 <= price[i] <= 105

## Solution:

A **Simple Solution** is to consider every index ‘i’ and do the following

*Max profit with at most two transactions =   
MAX {max profit with one transaction and subarray price[0..i] +   
max profit with one transaction and subarray price[i+1..n-1] }   
i varies from 0 to n-1.*

Maximum possible using one transaction can be calculated using the following O(n) algorithm   
[The maximum difference between two elements such that](https://www.geeksforgeeks.org/maximum-difference-between-two-elements/)the [larger element appears after the smaller number](https://www.geeksforgeeks.org/maximum-difference-between-two-elements/)  
The time complexity of the above simple solution is O(n2).

We can do this O(n) using the following **Efficient Solution**. The idea is to store the maximum possible profit of every subarray and solve the problem in the following two phases.

**1)** Create a table profit[0..n-1] and initialize all values in it 0.  
**2)** Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]  
**3)**Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].  
**4)**Return profit[n-1]

To do step 2, we need to keep track of the maximum price from right to left side, and to do step 3, we need to keep track of the minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in the third step, we use the same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After iteration i, the array profit[0..i] contains the maximum profit with 2 transactions, and profit[i+1..n-1] contains profit with two transactions.

Below are the implementations of the above idea.

// C++ program to find maximum

// possible profit with at most

// two transactions

#include <bits/stdc++.h>

using namespace std;

// Returns maximum profit with

// two transactions on a given

// list of stock prices, price[0..n-1]

int maxProfit(int price[], int n)

{

// Create profit array and

// initialize it as 0

int\* profit = new int[n];

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

profit[i] = 0;

/\* Get the maximum profit with

only one transaction

allowed. After this loop,

profit[i] contains maximum

profit from price[i..n-1]

using at most one trans. \*/

int max\_price = price[n - 1];

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

// max\_price has maximum

// of price[i..n-1]

if (price[i] > max\_price)

max\_price = price[i];

// we can get profit[i] by taking maximum of:

// a) previous maximum, i.e., profit[i+1]

// b) profit by buying at price[i] and selling at

// max\_price

profit[i]

= max(profit[i + 1], max\_price - price[i]);

}

/\* Get the maximum profit with two transactions allowed

After this loop, profit[n-1] contains the result \*/

int min\_price = price[0];

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

// min\_price is minimum price in price[0..i]

if (price[i] < min\_price)

min\_price = price[i];

// Maximum profit is maximum of:

// a) previous maximum, i.e., profit[i-1]

// b) (Buy, Sell) at (min\_price, price[i]) and add

// profit of other trans. stored in profit[i]

profit[i] = max(profit[i - 1],

profit[i] + (price[i] - min\_price));

}

int result = profit[n - 1];

delete[] profit; // To avoid memory leak

return result;

}

// Driver code

int main()

{

int price[] = { 2, 30, 15, 10, 8, 25, 80 };

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

cout << "Maximum Profit = " << maxProfit(price, n);

return 0;

}

**Output**

Maximum Profit = 100

The time complexity of the above solution is O(n).

Algorithmic Paradigm: Dynamic Programming

**Another approach:**

Initialize four variables for taking care of the first buy, first sell, second buy, second sell. Set first buy and second buy as INT\_MIN and first and second sell as 0. This is to ensure to get profit from transactions. Iterate through the array and return the second sell as it will store maximum profit.

#include <iostream>

#include<climits>

using namespace std;

int maxtwobuysell(int arr[],int size) {

int first\_buy = INT\_MIN;

int first\_sell = 0;

int second\_buy = INT\_MIN;

int second\_sell = 0;

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

first\_buy = max(first\_buy,-arr[i]);//we set prices to negative, so the calculation of profit will be convenient

first\_sell = max(first\_sell,first\_buy+arr[i]);

second\_buy = max(second\_buy,first\_sell-arr[i]);//we can buy the second only after first is sold

second\_sell = max(second\_sell,second\_buy+arr[i]);

}

return second\_sell;

}

int main() {

int arr[] = {2, 30, 15, 10, 8, 25, 80};

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

cout<<maxtwobuysell(arr,size);

return 0;

}

**Output**

100

**Time Complexity:** O(N)

**Auxiliary Space:** O(1)

# 427. [Optimal Strategy for a Game](https://practice.geeksforgeeks.org/problems/optimal-strategy-for-a-game/0)

You are given an array **A of size N**. The array contains integers and is of **even length**. The elements of the array represent N **coin**of **values V1, V2, ....Vn**. You play against an opponent in an **alternating**way.

In each **turn**, a player selects either the **first or last coin** from the **row**, removes it from the row permanently, and **receives the value** of the coin.

You need to determine the **maximum possible amount of money**you can win if you **go first**.  
**Note:** Both the players are playing optimally.

**Example 1:**

**Input:**

N = 4

A[] = {5,3,7,10}

**Output:** 15

**Explanation:** The user collects maximum

value as 15(10 + 5)

**Example 2:**

**Input:**

N = 4

A[] = {8,15,3,7}

**Output:** 22

**Explanation:** The user collects maximum

value as 22(7 + 15)

**Your Task:**  
Complete the function **maximumAmount()** which takes an array arr[] (represent values of N coins) and N as number of coins as a parameter and returns the **maximum possible amount of money**you can win if you **go first**.

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

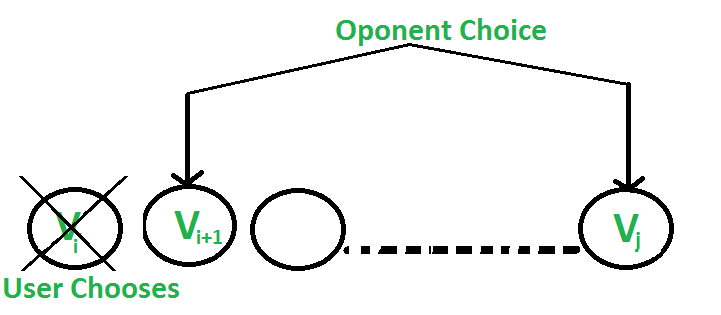
**Constraints:**  
2 <= N <= 103  
1 <= Ai <= 106

## Solution:

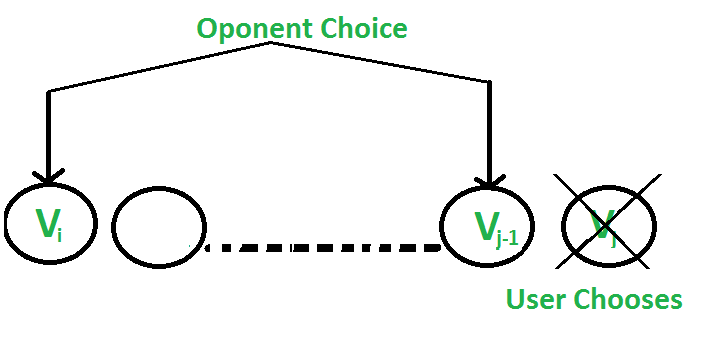
**Approach:** As both the players are equally strong, both will try to reduce the possibility of winning of each other. Now let’s see how the opponent can achieve this.

There are two choices:

* The user chooses the ‘ith’ coin with value ‘Vi’: The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with **minimum value**.   
  i.e. The user can collect the value **Vi + min(F(i+2, j), F(i+1, j-1) )**.



* The user chooses the ‘jth’ coin with value ‘Vj’: The opponent either chooses ‘ith’ coin or ‘(j-1)th’ coin. The opponent intends to choose the coin which leaves the user with minimum value, i.e. the user can collect the value **Vj + min(F(i+1, j-1), F(i, j-2) )**.



Following is the recursive solution that is based on the above two choices. We take a maximum of two choices.

F(i, j) represents the maximum value the user

can collect from i'th coin to j'th coin.

F(i, j) = Max(Vi + min(F(i+2, j), F(i+1, j-1) ),

Vj + min(F(i+1, j-1), F(i, j-2) ))

As user wants to maximise the number of coins.

Base Cases

F(i, j) = Vi If j == i

F(i, j) = max(Vi, Vj) If j == i + 1

class Solution{

public:

long long fun(int arr[], int n, vector<vector<long long>> &dp, int i, int j){

if(i>=n || j<0)

return 0;

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

return dp[i][j];

if(i==j)

return dp[i][j] = arr[i];

if(i>j)

return dp[i][j] = 0;

long long val1 = arr[i] + min(fun(arr, n, dp, i+2, j), fun(arr, n, dp, i+1, j-1));

long long val2 = arr[j] + min(fun(arr, n, dp, i+1, j-1), fun(arr, n, dp, i, j-2));

return dp[i][j] = max(val1, val2);

}

long long maximumAmount(int arr[], int n){

// Your code here

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

return fun(arr, n, dp, 0, n-1);

}

};

**Iterative:**

// C++ program to find out

// maximum value from a given

// sequence of coins

#include <bits/stdc++.h>

using namespace std;

// Returns optimal value possible

// that a player can collect from

// an array of coins of size n.

// Note than n must be even

int optimalStrategyOfGame(

int\* arr, int n)

{

// Create a table to store

// solutions of subproblems

int table[n][n];

// Fill table using above

// recursive formula. Note

// that the table is filled

// in diagonal fashion (similar

// to http:// goo.gl/PQqoS),

// from diagonal elements to

// table[0][n-1] which is the result.

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

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

// Here x is value of F(i+2, j),

// y is F(i+1, j-1) and

// z is F(i, j-2) in above recursive

// formula

int x = ((i + 2) <= j)

? table[i + 2][j]

: 0;

int y = ((i + 1) <= (j - 1))

? table[i + 1][j - 1]

: 0;

int z = (i <= (j - 2))

? table[i][j - 2]

: 0;

table[i][j] = max(

arr[i] + min(x, y),

arr[j] + min(y, z));

}

}

return table[0][n - 1];

}

// Driver program to test above function

int main()

{

int arr1[] = { 8, 15, 3, 7 };

int n = sizeof(arr1) / sizeof(arr1[0]);

printf("%d\n",

optimalStrategyOfGame(arr1, n));

int arr2[] = { 2, 2, 2, 2 };

n = sizeof(arr2) / sizeof(arr2[0]);

printf("%d\n",

optimalStrategyOfGame(arr2, n));

int arr3[] = { 20, 30, 2, 2, 2, 10 };

n = sizeof(arr3) / sizeof(arr3[0]);

printf("%d\n",

optimalStrategyOfGame(arr3, n));

return 0;

}

**Output:**

22

4

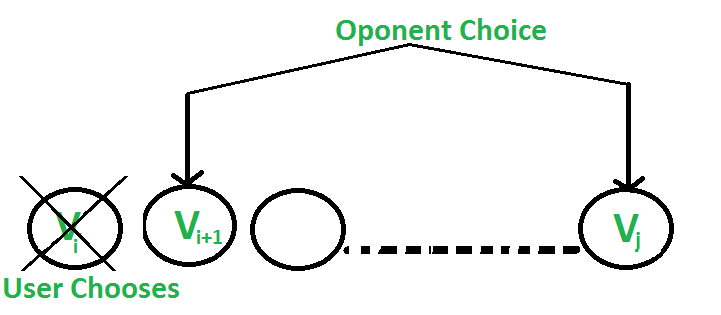
42

**Complexity Analysis:**

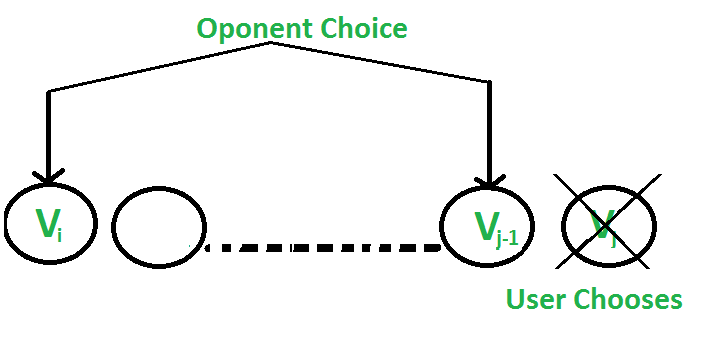
* **Time Complexity:** O(n2).   
  Use of a nested for loop brings the time complexity to n2.
* **Auxiliary Space:** O(n2).   
  As a 2-D table is used for storing states.

**Note:** The above solution can be optimized by using less number of comparisons for every choice.

We have discussed an [approach that makes 4 recursive calls](https://www.geeksforgeeks.org/optimal-strategy-for-a-game-dp-31/). In this post, a new approach is discussed that makes two recursive calls.  
There are two choices:   
**1.** The user chooses the ith coin with value Vi: The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with minimum value.   
i.e. The user can collect the value Vi + (Sum – Vi) – F(i+1, j, Sum – Vi) where Sum is sum of coins from index i to j. The expression can be simplified to Sum – F(i+1, j, Sum – Vi) 



**2.** The user chooses the jth coin with value Vj: The opponent either chooses ith coin or (j-1)th coin. The opponent intends to choose the coin which leaves the user with minimum value.   
i.e. The user can collect the value Vj + (Sum – Vj) – F(i, j-1, Sum – Vj) where Sum is sum of coins from index i to j. The expression can be simplified to Sum – F(i, j-1, Sum – Vj) 



Following is recursive solution that is based on above two choices. We take the maximum of two choices.

F(i, j) represents the maximum value the user can collect from

i'th coin to j'th coin.

arr[] represents the list of coins

F(i, j) = Max(Sum - F(i+1, j, Sum-arr[i]),

Sum - F(i, j-1, Sum-arr[j]))

Base Case

F(i, j) = max(arr[i], arr[j]) If j == i+1

**Memoization Based Solution**

// C++ program to find out maximum value from a

// given sequence of coins

#include <bits/stdc++.h>

using namespace std;

const int MAX = 100;

int memo[MAX][MAX];

int oSRec(int arr[], int i, int j, int sum)

{

if (j == i + 1)

return max(arr[i], arr[j]);

if (memo[i][j] != -1)

return memo[i][j];

// For both of your choices, the opponent

// gives you total sum minus maximum of

// his value

memo[i][j] = max((sum - oSRec(arr, i + 1, j, sum - arr[i])),

(sum - oSRec(arr, i, j - 1, sum - arr[j])));

return memo[i][j];

}

// Returns optimal value possible that a player can

// collect from an array of coins of size n. Note

// than n must be even

int optimalStrategyOfGame(int\* arr, int n)

{

// Compute sum of elements

int sum = 0;

sum = accumulate(arr, arr + n, sum);

// Initialize memoization table

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

return oSRec(arr, 0, n - 1, sum);

}

// Driver program to test above function

int main()

{

int arr1[] = { 8, 15, 3, 7 };

int n = sizeof(arr1) / sizeof(arr1[0]);

printf("%d\n", optimalStrategyOfGame(arr1, n));

int arr2[] = { 2, 2, 2, 2 };

n = sizeof(arr2) / sizeof(arr2[0]);

printf("%d\n", optimalStrategyOfGame(arr2, n));

int arr3[] = { 20, 30, 2, 2, 2, 10 };

n = sizeof(arr3) / sizeof(arr3[0]);

printf("%d\n", optimalStrategyOfGame(arr3, n));

return 0;

}

**Output:**

22

4

42

# 428. Optimal Binary Search Tree

Given a sorted array key *[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches for *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.  
Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

**Examples:**

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs

10 12

\ /

12 10

I II

Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is 34\*1 + 50\*2 = 134

The cost of tree II is 50\*1 + 34\*2 = 118

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs

10 12 20 10 20

\ / \ / \ /

12 10 20 12 20 10

\ / / \

20 10 12 12

I II III IV V

Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1\*50 + 2\*34 + 3\*8 = 142

## Solution:

**1) Optimal Substructure:**   
The optimal cost for freq[i..j] can be recursively calculated using the following formula.   
  
We need to calculate ***optCost(0, n-1)*** to find the result.   
The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.   
We add sum of frequencies from i to j (see first term in the above formula)

**The reason for adding the sum of frequencies from i to j:**

This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements from i to j except r. The term freq[r] is added because it is going to be root and that means level of 1, so freq[r]\*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining elements because as we take r as root then all the elements other than that are going 1 level down than that is calculated in the subproblem. Let me put it in a more clear way, for calculating optcost(i,j) we assume that the r is taken as root and calculate min of opt(i,r-1)+opt(r+1,j) for all i<=r<=j. Here for every subproblem we are  choosing one node as a root. But in reality the level of subproblem root and all its descendant nodes will be 1 greater than the level of the parent problem root. Therefore the frequency of all the nodes except r should be added which accounts to the descend in their level compared to level assumed in subproblem.  
**2) Overlapping Subproblems**   
Following is recursive implementation that simply follows the recursive structure mentioned above.

// A naive recursive implementation of

// optimal binary search tree problem

#include <bits/stdc++.h>

using namespace std;

// A utility function to get sum of

// array elements freq[i] to freq[j]

int sum(int freq[], int i, int j);

// A recursive function to calculate

// cost of optimal binary search tree

int optCost(int freq[], int i, int j)

{

// Base cases

if (j < i) // no elements in this subarray

return 0;

if (j == i) // one element in this subarray

return freq[i];

// Get sum of freq[i], freq[i+1], ... freq[j]

int fsum = sum(freq, i, j);

// Initialize minimum value

int min = INT\_MAX;

// One by one consider all elements

// as root and recursively find cost

// of the BST, compare the cost with

// min and update min if needed

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

{

int cost = optCost(freq, i, r - 1) +

optCost(freq, r + 1, j);

if (cost < min)

min = cost;

}

// Return minimum value

return min + fsum;

}

// The main function that calculates

// minimum cost of a Binary Search Tree.

// It mainly uses optCost() to find

// the optimal cost.

int optimalSearchTree(int keys[],

int freq[], int n)

{

// Here array keys[] is assumed to be

// sorted in increasing order. If keys[]

// is not sorted, then add code to sort

// keys, and rearrange freq[] accordingly.

return optCost(freq, 0, n - 1);

}

// A utility function to get sum of

// array elements freq[i] to freq[j]

int sum(int freq[], int i, int j)

{

int s = 0;

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

s += freq[k];

return s;

}

// Driver Code

int main()

{

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

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

cout << "Cost of Optimal BST is "

<< optimalSearchTree(keys, freq, n);

return 0;

}

**Output:**

Cost of Optimal BST is 142

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. We can see many subproblems being repeated in the following recursion tree for freq[1..4]. 

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

Since same subproblems are called again, this problem has Overlapping Subproblems property. So optimal BST 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 cost[][] in bottom up manner.  
**Dynamic Programming Solution**   
Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner> The idea used in the implementation is same as [Matrix Chain Multiplication problem](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/), we use a variable ‘L’ for chain length and increment ‘L’, one by one. We calculate column number ‘j’ using the values of ‘i’ and ‘L’.

// Dynamic Programming code for Optimal Binary Search

// Tree Problem

#include <bits/stdc++.h>

using namespace std;

// A utility function to get sum of array elements

// freq[i] to freq[j]

int sum(int freq[], int i, int j);

/\* A Dynamic Programming based function that calculates

minimum cost of a Binary Search Tree. \*/

int optimalSearchTree(int keys[], int freq[], int n)

{

/\* Create an auxiliary 2D matrix to store results

of subproblems \*/

int cost[n][n];

/\* cost[i][j] = Optimal cost of binary search tree

that can be formed from keys[i] to keys[j].

cost[0][n-1] will store the resultant cost \*/

// For a single key, cost is equal to frequency of the key

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

cost[i][i] = freq[i];

// Now we need to consider chains of length 2, 3, ... .

// L is chain length.

for (int L = 2; L <= n; L++)

{

// i is row number in cost[][]

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

{

// Get column number j from row number i and

// chain length L

int j = i+L-1;

cost[i][j] = INT\_MAX;

// Try making all keys in interval keys[i..j] as root

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

{

// c = cost when keys[r] becomes root of this subtree

int c = ((r > i)? cost[i][r-1]:0) +

((r < j)? cost[r+1][j]:0) +

sum(freq, i, j);

if (c < cost[i][j])

cost[i][j] = c;

}

}

}

return cost[0][n-1];

}

// A utility function to get sum of array elements

// freq[i] to freq[j]

int sum(int freq[], int i, int j)

{

int s = 0;

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

s += freq[k];

return s;

}

// Driver code

int main()

{

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

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

cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);

return 0;

}

**Output:** 

Cost of Optimal BST is 142

**Notes**   
**1)** The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.  
**2)** In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen ‘r’ in the innermost loop.

# 429. Palindrome Partitioning Problem

Given a string **str**, a partitioning of the string is a *palindrome partitioning* if every sub-string of the partition is a palindrome. Determine the fewest cuts needed for palindrome partitioning of given string.

**Example 1:**

**Input:** str = "ababbbabbababa"

**Output:** 3

**Explaination:** After 3 partitioning substrings

are "a", "babbbab", "b", "ababa".

**Example 2:**

**Input:** str = "aaabba"

**Output:** 1

**Explaination:** The substrings after 1

partitioning are "aa" and "abba".

**Your Task:**  
You do not need to read input or print anything, Your task is to complete the function **palindromicPartition()** which takes the string str as input parameter and returns minimum number of partitions required.

**Expected Time Complexity:** O(n\*n) [n is the length of the string str]  
**Expected Auxiliary Space:** O(n\*n)

**Constraints:**  
1 ≤ length of str ≤ 500

## Solution:

**My approach:**

class Solution{

public:

int fun(vector<int> &dp, vector<vector<bool>> &ispallin, string str, int ind){

if(ind>=str.size())

return 0;

if(dp[ind]!=0)

return dp[ind];

int res = str.size()-ind+1;

for(int i=ind;i<str.size();i++){

if(ispallin[ind][i]==true){

res = min(res, 1+fun(dp, ispallin, str, i+1));

}

}

return dp[ind] = res;

}

int palindromicPartition(string str)

{

// code here

int n = str.size();

vector<vector<bool>> ispallin(n, vector<bool> (n));

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

ispallin[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(str[i]!=str[j])

ispallin[i][j] = false;

else if(l==2)

ispallin[i][j] = true;

else

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

}

}

vector<int> dp(n, 0);

int ans = fun(dp, ispallin, str, 0);

return ans-1;

}

};

**Time Complexity:** O(n\*n) [n is the length of the string str]  
**Auxiliary Space:** O(n\*n)

This problem is a variation of [Matrix Chain Multiplication](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/) problem. If the string is a palindrome, then we simply return 0. Else, like the Matrix Chain Multiplication problem, we try making cuts at all possible places, recursively calculate the cost for each cut and return the minimum value.   
Let the given string be str and minPalPartion() be the function that returns the fewest cuts needed for palindrome partitioning. following is the optimal substructure property.

**Using Recursion**

// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1

minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.

minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be

// calculated recursively using the following formula.

minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +

minPalPartion(str, k+1, j) }

where k varies from i to j-1

// C++ Code for Palindrome Partitioning

// Problem

#include <bits/stdc++.h>

using namespace std;

bool isPalindrome(string String, int i, int j)

{

while(i < j)

{

if(String[i] != String[j])

return false;

i++;

j--;

}

return true;

}

int minPalPartion(string String, int i, int j)

{

if( i >= j || isPalindrome(String, i, j) )

return 0;

int ans = INT\_MAX, count;

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

{

count = minPalPartion(String, i, k) +

minPalPartion(String, k + 1, j) + 1;

ans = min(ans, count);

}

return ans;

}

// Driver code

int main() {

string str = "ababbbabbababa";

cout << "Min cuts needed for " <<

"Palindrome Partitioning is " <<

minPalPartion(str, 0, str.length() - 1) << endl;

return 0;

}

**Output:**

Min cuts needed for Palindrome Partitioning is 3

**An optimization to above approach**   
In the above approach, we can calculate the minimum cut while finding all palindromic substring. If we find all palindromic substring 1st and then we calculate minimum cut, time complexity will reduce to O(n2).

// Dynamic Programming Solution for Palindrome Partitioning Problem

#include <iostream>

#include <bits/stdc++.h>

#include <string.h>

using namespace std;

// A utility function to get minimum of two integers

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

// Returns the minimum number of cuts needed to partition a string

// such that every part is a palindrome

int minPalPartion(char\* str)

{

// Get the length of the string

int n = strlen(str);

/\* Create two arrays to build the solution in bottom-up manner

C[i] = Minimum number of cuts needed for a palindrome partitioning

of substring str[0..i]

P[i][j] = true if substring str[i..j] is palindrome, else false

Note that C[i] is 0 if P[0][i] is true \*/

int C[n];

bool P[n][n];

int i, j, k, L; // different looping variables

// Every substring of length 1 is a palindrome

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

P[i][i] = true;

}

/\* L is substring length. Build the solution in bottom up manner by

considering all substrings of length starting from 2 to n. \*/

for (L = 2; L <= n; L++) {

// For substring of length L, set different possible starting indexes

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

j = i + L - 1; // Set ending index

// If L is 2, then we just need to compare two characters. Else

// need to check two corner characters and value of P[i+1][j-1]

if (L == 2)

P[i][j] = (str[i] == str[j]);

else

P[i][j] = (str[i] == str[j]) && P[i + 1][j - 1];

}

}

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

if (P[0][i] == true)

C[i] = 0;

else {

C[i] = INT\_MAX;

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

if (P[j + 1][i] == true && 1 + C[j] < C[i])

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

}

}

}

// Return the min cut value for complete string. i.e., str[0..n-1]

return C[n - 1];

}

// Driver program to test above function

int main()

{

char str[] = "ababbbabbababa";

cout <<"Min cuts needed for Palindrome Partitioning is " << minPalPartion(str);

return 0;

}

**Output:**

Min cuts needed for Palindrome Partitioning is 3

**Time Complexity:** O(n2)

**Using Memorization to solve this problem.**   
The basic idea is to cache the intermittent results calculated in recursive functions. We can put these results into a hashmap/unordered\_map.   
To calculate the keys for the Hashmap we will use the starting and end index of the string as the key i.e. [“start\_index”.append(“end\_index”)] would be the key for the Hashmap.

Below is the implementation of above approach :

// Using memoizatoin to solve the partition problem.

#include <bits/stdc++.h>

using namespace std;

// Function to check if input string is palindrome or not

bool ispalindrome(string input, int start, int end)

{

// Using two pointer technique to check palindrome

while (start < end) {

if (input[start] != input[end])

return false;

start++;

end--;

}

return true;

}

// Function to find keys for the Hashmap

string convert(int a, int b)

{

return to\_string(a) + "" + to\_string(b);

}

// Returns the minimum number of cuts needed to partition a string

// such that every part is a palindrome

int minpalparti\_memo(string input, int i, int j, unordered\_map<string, int>& memo)

{

if (i > j)

return 0;

// Key for the Input String

string ij = convert(i, j);

// If the no of partitions for string "ij" is already calculated

// then return the calculated value using the Hashmap

if (memo.find(ij) != memo.end()) {

return memo[ij];

}

// Every String of length 1 is a palindrome

if (i == j) {

memo[ij] = 0;

return 0;

}

if (ispalindrome(input, i, j)) {

memo[ij] = 0;

return 0;

}

int minimum = INT\_MAX;

// Make a cut at every possible location starting from i to j

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

int left\_min = INT\_MAX;

int right\_min = INT\_MAX;

string left = convert(i, k);

string right = convert(k + 1, j);

// If left cut is found already

if (memo.find(left) != memo.end()) {

left\_min = memo[left];

}

// If right cut is found already

if (memo.find(right) != memo.end()) {

right\_min = memo[right];

}

// Recursively calculating for left and right strings

if (left\_min == INT\_MAX)

left\_min = minpalparti\_memo(input, i, k, memo);

if (right\_min == INT\_MAX)

right\_min = minpalparti\_memo(input, k + 1, j, memo);

// Taking minimum of all k possible cuts

minimum = min(minimum, left\_min + 1 + right\_min);

}

memo[ij] = minimum;

// Return the min cut value for complete string.

return memo[ij];

}

int main()

{

string input = "ababbbabbababa";

unordered\_map<string, int> memo;

cout << minpalparti\_memo(input, 0, input.length() - 1, memo) << endl;

return 0;

}

**Time Complexity:** O(n2)

# 430. Word Wrap Problem

## Same as ques 59 of String.

# 431. Mobile Numeric Keypad Problem [ IMP ]

Given the mobile numeric keypad. You can only press buttons that are up, left, right, or down to the current button. You are not allowed to press bottom row corner buttons (i.e. \* and # ). Given a number **N**, the task is to find out the number of possible numbers of the given length.

**Example 1:**

**Input**: 1

**Output:** 10

**Explanation**: Number of possible numbers

would be 10 (0, 1, 2, 3, …., 9)

**Example 2:**

**Input:** N = 2

**Output:** 36

**Explanation**: Possible numbers: 00, 08, 11,

12, 14, 22, 21, 23, 25 and so on.

If we start with 0, valid numbers

will be 00, 08 (count: 2)

If we start with 1, valid numbers

will be 11, 12, 14 (count: 3)

If we start with 2, valid numbers

will be 22, 21, 23,25 (count: 4)

If we start with 3, valid numbers

will be 33, 32, 36 (count: 3)

If we start with 4, valid numbers

will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers

will be 55,54,52,56,58 (count: 5)

and so on..

**Your Task:**  
You don't need to read input or print anything. Complete the function **getCount()**which takes **N** as input parameter and returns the integer value  
  
**Expected Time Complexity:** O(**N**)  
**Expected Auxiliary Space:** O(**N**)  
  
**Constraints:**  
1 ≤ **N** ≤ 25

## Solution:

**My Solution:**

public:

vector<vector<int>> arr;

long long dp[26][10];

long long count;

long long fun(int n, int num){

if(n==1){

return 1;

}

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

return dp[n][num];

long long res = 0;

for(int i=0;i<arr[num].size();i++)

res += fun(n-1, arr[num][i]);

return dp[n][num] = res;

}

long long getCount(int N)

{

// Your code goes here

arr.clear();

arr.push\_back({0, 8});

arr.push\_back({1, 2, 4});

arr.push\_back({2, 1, 3, 5});

arr.push\_back({3, 2, 6});

arr.push\_back({4, 1, 5, 7});

arr.push\_back({5, 2, 4, 6, 8});

arr.push\_back({6, 3, 5, 9});

arr.push\_back({7, 4, 8});

arr.push\_back({8, 5, 7, 9, 0});

arr.push\_back({9, 6, 8});

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

count = 0;

for(int i=0;i<=9;i++){

count += fun(N, i);

}

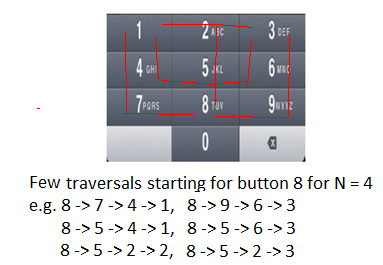
return count;

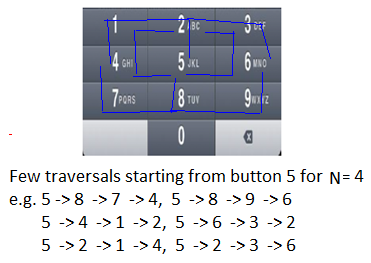
}

**Time Complexity:** O(**N**)

**Space Complexity:** O(**N**)

**Dynamic Programming**   
There are many repeated traversal on smaller paths (traversal for smaller N) to find all possible longer paths (traversal for bigger N). See following two diagrams for example. In this traversal, for N = 4 from two starting positions (buttons ‘4’ and ‘8’), we can see there are few repeated traversals for N = 2 (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc). 





Since the problem has both properties: [Optimal Substructure](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/) and [Overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/), it can be efficiently solved using dynamic programming.

Following is the program for dynamic programming implementation.

// A Dynamic Programming based C program to count number of

// possible numbers of given length

#include <stdio.h>

// Return count of all possible numbers of length n

// in a given numeric keyboard

int getCount(char keypad[][3], int n)

{

if(keypad == NULL || n <= 0)

return 0;

if(n == 1)

return 10;

// left, up, right, down move from current location

int row[] = {0, 0, -1, 0, 1};

int col[] = {0, -1, 0, 1, 0};

// taking n+1 for simplicity - count[i][j] will store

// number count starting with digit i and length j

int count[10][n+1];

int i=0, j=0, k=0, move=0, ro=0, co=0, num = 0;

int nextNum=0, totalCount = 0;

// count numbers starting with digit i and of lengths 0 and 1

for (i=0; i<=9; i++)

{

count[i][0] = 0;

count[i][1] = 1;

}

// Bottom up - Get number count of length 2, 3, 4, ... , n

for (k=2; k<=n; k++)

{

for (i=0; i<4; i++) // Loop on keypad row

{

for (j=0; j<3; j++) // Loop on keypad column

{

// Process for 0 to 9 digits

if (keypad[i][j] != '\*' && keypad[i][j] != '#')

{

// Here we are counting the numbers starting with

// digit keypad[i][j] and of length k keypad[i][j]

// will become 1st digit, and we need to look for

// (k-1) more digits

num = keypad[i][j] - '0';

count[num][k] = 0;

// move left, up, right, down from current location

// and if new location is valid, then get number

// count of length (k-1) from that new digit and

// add in count we found so far

for (move=0; move<5; move++)

{

ro = i + row[move];

co = j + col[move];

if (ro >= 0 && ro <= 3 && co >=0 && co <= 2 &&

keypad[ro][co] != '\*' && keypad[ro][co] != '#')

{

nextNum = keypad[ro][co] - '0';

count[num][k] += count[nextNum][k-1];

}

}

}

}

}

}

// Get count of all possible numbers of length "n" starting

// with digit 0, 1, 2, ..., 9

totalCount = 0;

for (i=0; i<=9; i++)

totalCount += count[i][n];

return totalCount;

}

// Driver program to test above function

int main(int argc, char \*argv[])

{

char keypad[4][3] = {{'1','2','3'},

{'4','5','6'},

{'7','8','9'},

{'\*','0','#'}};

printf("Count for numbers of length %d: %dn", 1, getCount(keypad, 1));

printf("Count for numbers of length %d: %dn", 2, getCount(keypad, 2));

printf("Count for numbers of length %d: %dn", 3, getCount(keypad, 3));

printf("Count for numbers of length %d: %dn", 4, getCount(keypad, 4));

printf("Count for numbers of length %d: %dn", 5, getCount(keypad, 5));

return 0;

}

**Output:**

Count for numbers of length 1: 10

Count for numbers of length 2: 36

Count for numbers of length 3: 138

Count for numbers of length 4: 532

Count for numbers of length 5: 2062

**A Space Optimized Solution:**   
The above dynamic programming approach also runs in O(n) time and requires O(n) auxiliary space, as only one for loop runs n times, other for loops runs for constant time. We can see that nth iteration needs data from (n-1)th iteration only, so we need not keep the data from older iterations. We can have a space efficient dynamic programming approach with just two arrays of size 10.

// A Space Optimized C++ program to count number of possible numbers

// of given length

#include <bits/stdc++.h>

using namespace std;

// Return count of all possible numbers of length n

// in a given numeric keyboard

int getCount(char keypad[][3], int n)

{

if (keypad == NULL || n <= 0)

return 0;

if (n == 1)

return 10;

// odd[i], even[i] arrays represent count of numbers starting

// with digit i for any length j

int odd[10], even[10];

int i = 0, j = 0, useOdd = 0, totalCount = 0;

for (i = 0; i <= 9; i++)

odd[i] = 1; // for j = 1

for (j = 2; j <= n; j++) // Bottom Up calculation from j = 2 to n

{

useOdd = 1 - useOdd;

// Here we are explicitly writing lines for each number 0

// to 9. But it can always be written as DFS on 4X3 grid

// using row, column array valid moves

if (useOdd == 1)

{

even[0] = odd[0] + odd[8];

even[1] = odd[1] + odd[2] + odd[4];

even[2] = odd[2] + odd[1] + odd[3] + odd[5];

even[3] = odd[3] + odd[2] + odd[6];

even[4] = odd[4] + odd[1] + odd[5] + odd[7];

even[5] = odd[5] + odd[2] + odd[4] + odd[8] + odd[6];

even[6] = odd[6] + odd[3] + odd[5] + odd[9];

even[7] = odd[7] + odd[4] + odd[8];

even[8] = odd[8] + odd[0] + odd[5] + odd[7] + odd[9];

even[9] = odd[9] + odd[6] + odd[8];

}

else

{

odd[0] = even[0] + even[8];

odd[1] = even[1] + even[2] + even[4];

odd[2] = even[2] + even[1] + even[3] + even[5];

odd[3] = even[3] + even[2] + even[6];

odd[4] = even[4] + even[1] + even[5] + even[7];

odd[5] = even[5] + even[2] + even[4] + even[8] + even[6];

odd[6] = even[6] + even[3] + even[5] + even[9];

odd[7] = even[7] + even[4] + even[8];

odd[8] = even[8] + even[0] + even[5] + even[7] + even[9];

odd[9] = even[9] + even[6] + even[8];

}

}

// Get count of all possible numbers of length "n" starting

// with digit 0, 1, 2, ..., 9

totalCount = 0;

if (useOdd == 1)

{

for (i = 0; i <= 9; i++)

totalCount += even[i];

}

else

{

for (i = 0; i <= 9; i++)

totalCount += odd[i];

}

return totalCount;

}

// Driver program to test above function

int main()

{

char keypad[4][3] = {{'1', '2', '3'},

{'4', '5', '6'},

{'7', '8', '9'},

{'\*', '0', '#'}};

cout << "Count for numbers of length 1: " << getCount(keypad, 1) << endl;

cout << "Count for numbers of length 2: " << getCount(keypad, 2) << endl;

cout << "Count for numbers of length 3: " << getCount(keypad, 3) << endl;

cout << "Count for numbers of length 4: " << getCount(keypad, 4) << endl;

cout << "Count for numbers of length 5: " << getCount(keypad, 5) << endl;

return 0;

}

**Output:**

Count for numbers of length 1: 10

Count for numbers of length 2: 36

Count for numbers of length 3: 138

Count for numbers of length 4: 532

Count for numbers of length 5: 2062

# 432. Boolean Parenthesization Problem

Given a boolean expression **S** of length **N** with following symbols.  
Symbols  
    'T' ---> true  
    'F' ---> false  
and following operators filled between symbols  
Operators  
    &   ---> boolean AND  
    |   ---> boolean OR  
    ^   ---> boolean XOR  
Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

**Example 1:**

**Input:** N = 7

S = T|T&F^T

**Output:** 4

**Explaination:** The expression evaluates

to true in 4 ways ((T|T)&(F^T)),

(T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T)).

**Example 2:**

**Input:** N = 5

S = T^F|F

**Output:** 2

**Explaination:** ((T^F)|F) and (T^(F|F)) are the

only ways.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **countWays()** which takes N and S as input parameters and returns number of possible ways modulo 1003.

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

**Constraints:**  
1 ≤ N ≤ 200

## Solution:

class Solution{

public:

pair<int, int> fun(vector<vector<pair<int, int>>> &dp, string &str, int i, int j){

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

return dp[i][j];

if(i==j){

if(str[i]=='T')

return dp[i][j] = {1, 0};

else

return dp[i][j] = {0, 1};

}

int res0 = 0, res1 = 0;

for(int ind = i+1; ind <= j-1; ind += 2){

pair<int, int> p1 = fun(dp, str, i, ind-1);

int a1 = p1.first, a2 = p1.second;

pair<int, int> p2 = fun(dp, str, ind+1, j);

int b1 = p2.first, b2 = p2.second;

if(str[ind]=='&'){

res1 += (a1\*b1);

res0 += (a1\*b2 + a2\*b1 + a2\*b2);

}

else if(str[ind]=='|'){

res1 += (a1\*b2 + b1\*a2 + a1\*b1);

res0 += (a2\*b2);

}

else{

res1 += (a1\*b2 + b1\*a2);

res0 += (a1\*b1 + a2\*b2);

}

}

return dp[i][j] = {res1%1003, res0%1003};

}

int countWays(int N, string S){

vector<vector<pair<int,int>>> dp(N, vector<pair<int, int>> (N, {-1, -1}));

pair<int, int> res = fun(dp, S, 0, N-1);

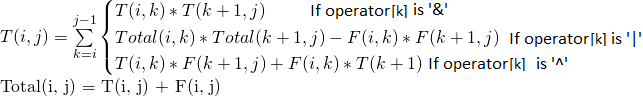
return (res.first);

}

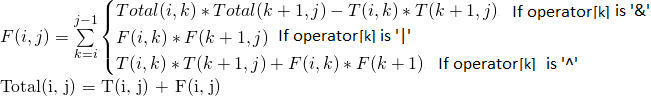
};

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

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



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



Base Cases:

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

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

F(i, i) = 1 if symbol[i] = 'F'

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

If we draw the recursion tree of the above recursive solution, we can observe that it many overlapping subproblems. Like other [dynamic programming problems](https://www.geeksforgeeks.org/tag/dynamic-programming/), it can be solved by filling a table in bottom-up manner. Following is the implementation of a dynamic programming solution.

#include <cstring>

#include <iostream>

using namespace std;

// Returns count of all possible

// parenthesizations that lead

// to result true for a boolean

// expression with symbols like

// true and false and operators

// like &, | and ^ filled

// between symbols

int countParenth(char symb[], char oper[], int n)

{

int F[n][n], T[n][n];

// Fill diagonal entries first

// All diagonal entries in

// T[i][i] are 1 if symbol[i]

// is T (true). Similarly,

// all F[i][i] entries are 1 if

// symbol[i] is F (False)

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

F[i][i] = (symb[i] == 'F') ? 1 : 0;

T[i][i] = (symb[i] == 'T') ? 1 : 0;

}

// Now fill T[i][i+1],

// T[i][i+2], T[i][i+3]... in order

// And F[i][i+1], F[i][i+2],

// F[i][i+3]... in order

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

{

for (int i = 0, j = gap;

j < n; ++i, ++j)

{

T[i][j] = F[i][j] = 0;

for (int g = 0;

g < gap; g++)

{

// Find place of parenthesization using

// current value of gap

int k = i + g;

// Store Total[i][k]

// and Total[k+1][j]

int tik = T[i][k] + F[i][k];

int tkj = T[k + 1][j]

+ F[k + 1][j];

// Follow the recursive formulas

// according

// to the current operator

if (oper[k] == '&') {

T[i][j] += T[i][k]

\* T[k + 1][j];

F[i][j] += (tik \* tkj

- T[i][k]

\* T[k + 1][j]);

}

if (oper[k] == '|') {

F[i][j] += F[i][k]

\* F[k + 1][j];

T[i][j] += (tik \* tkj

- F[i][k]

\* F[k + 1][j]);

}

if (oper[k] == '^') {

T[i][j] += F[i][k]

\* T[k + 1][j]

+ T[i][k]

\* F[k + 1][j];

F[i][j] += T[i][k]

\* T[k + 1][j]

+ F[i][k] \* F[k + 1][j];

}

}

}

}

return T[0][n - 1];

}

// Driver code

int main()

{

char symbols[] = "TTFT";

char operators[] = "|&^";

int n = strlen(symbols);

// There are 4 ways

// ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and

// (T|((T&F)^T))

cout << countParenth(symbols, operators, n);

return 0;

}

**Output:**

4

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

# 433. Largest rectangular sub-matrix whose sum is 0

Given a matrix **mat**[][] of size **N** x **M.**The task is to find the largest rectangular sub-matrix whose sum is 0.

**Example 1:**

**Input:** N = 3, M = 3

mat[][] = 1, 2, 3

-3,-2,-1

1, 7, 5

**Output:** 1, 2, 3

-3,-2,-1

**Explanation:** This is the largest sub-matrix,

whose sum is 0.

**Example 2:**

**Input:** N = 4, M = 4

mat[][] = 9, 7, 16, 5

1,-6,-7, 3

1, 8, 7, 9

7, -2, 0, 10

**Output:** -6,-7

8, 7

-2, 0

**Your Task:**  
You don't need to read input or print anything. You just have to complete the function **sumZeroMatrix()** which takes a 2D matrix **mat**[][], its dimensions **N** and **M** as inputs and returns a largest sub-matrix, whose sum is 0.

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

**Constraints**:  
1 <= N, M <= 100  
-1000 <= mat[][] <= 1000

## Solution:

The naive solution for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be O(n^4).  
The solution is based on[Maximum sum rectangle in a 2D matrix](https://www.geeksforgeeks.org/dynamic-programming-set-27-max-sum-rectangle-in-a-2d-matrix/). The idea is to reduce the problem to 1 D array. We can use Hashing to find maximum length of sub-array in 1-D array in O(n) time. We fix the left and right columns one by one and find the largest sub-array with 0 sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have sum is zero) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say temp[]. So temp[i] indicates sum of elements from left to right in row i. If we find largest subarray with 0 sum on temp, and no. of elements is greater than previous no. of elements then update the values of final row\_up, final row\_down, final col\_left, final col\_right.

// A C++ program to find Largest rectangular

// sub-matrix whose sum is 0

#include <bits/stdc++.h>

using namespace std;

const int MAX = 100;

// This function basically finds largest 0

// sum subarray in temp[0..n-1]. If 0 sum

// does't exist, then it returns false. Else

// it returns true and sets starting and

// ending indexes as starti and endj.

bool sumZero(int temp[], int\* starti,

int\* endj, int n)

{

// Map to store the previous sums

map<int, int> presum;

int sum = 0; // Initialize sum of elements

// Initialize length of sub-array with sum 0

int max\_length = 0;

// Traverse through the given array

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

{

// Add current element to sum

sum += temp[i];

if (temp[i] == 0 && max\_length == 0)

{

\*starti = i;

\*endj = i;

max\_length = 1;

}

if (sum == 0)

{

if (max\_length < i + 1)

{

\*starti = 0;

\*endj = i;

}

max\_length = i + 1;

}

// Look for this sum in Hash table

if (presum.find(sum) != presum.end())

{

// store previous max\_length so

// that we can check max\_length

// is updated or not

int old = max\_length;

// If this sum is seen before,

// then update max\_len

max\_length = max(max\_length, i - presum[sum]);

if (old < max\_length)

{

// If max\_length is updated then

// enter and update start and end

// point of array

\*endj = i;

\*starti = presum[sum] + 1;

}

}

else

// Else insert this sum with

// index in hash table

presum[sum] = i;

}

// Return true if max\_length is non-zero

return (max\_length != 0);

}

// The main function that finds Largest rectangle

// sub-matrix in a[][] whose sum is 0.

void sumZeroMatrix(int a[][MAX], int row, int col)

{

int temp[row];

// Variables to store the final output

int fup = 0, fdown = 0, fleft = 0, fright = 0;

int sum;

int up, down;

int maxl = INT\_MIN;

// Set the left column

for (int left = 0; left < col; left++)

{

// Initialize all elements of temp as 0

memset(temp, 0, sizeof(temp));

// Set the right column for the left column

// set by outer loop

for (int right = left; right < col; right++)

{

// Calculate sum between current left

// and right for every row 'i'

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

temp[i] += a[i][right];

// Find largest subarray with 0 sum in

// temp[]. The sumZero() function also

// sets values of start and finish. So

// 'sum' is sum of rectangle between (start,

// left) and (finish, right) which is

// boundary columns strictly as left and right.

bool sum = sumZero(temp, &up, &down, row);

int ele = (down - up + 1) \* (right - left + 1);

// Compare no. of elements with previous

// no. of elements in sub-Matrix.

// If new sub-matrix has more elements

// then update maxl and final boundaries

// like fup, fdown, fleft, fright

if (sum && ele > maxl)

{

fup = up;

fdown = down;

fleft = left;

fright = right;

maxl = ele;

}

}

}

// If there is no change in boundaries

// than check if a[0][0] is 0

// If it not zero then print

// that no such zero-sum sub-matrix exists

if (fup == 0 && fdown == 0 && fleft == 0 &&

fright == 0 && a[0][0] != 0) {

cout << "No zero-sum sub-matrix exists";

return;

}

// Print final values

for (int j = fup; j <= fdown; j++)

{

for (int i = fleft; i <= fright; i++)

cout << a[j][i] << " ";

cout << endl;

}

}

// Driver program to test above functions

int main()

{

int a[][MAX] = { { 9, 7, 16, 5 }, { 1, -6, -7, 3 },

{ 1, 8, 7, 9 }, { 7, -2, 0, 10 } };

int row = 4, col = 4;

sumZeroMatrix(a, row, col);

return 0;

}

Output: 

-6, -7

8, 7

-2, 0

Output: 

-6, -7

8, 7

-2, 0

**My Solution:**

class Solution{

public:

pair<int, int> fun(vector<int> vec, int n){

unordered\_map<int, int> mp;

mp[0] = -1;

int len=0, end=-1, sum=0;

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

sum += vec[i];

//cout<<sum<<endl;

if(mp.find(sum)==mp.end())

mp[sum] = i;

else{

//cout<<"here"<<endl;

int l = i-mp[sum];

if(l>len){

len = l;

end = i;

}

}

}

return {len, end};

}

vector<vector<int>> sumZeroMatrix(vector<vector<int>> a){

//Code Here

int area=0, row1, col1, row2, col2;

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

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

vector<int> vec(n, 0);

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

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

vec[k] += a[j][k];

//cout<<vec[k]<<" ";

}

//cout<<endl;

pair<int, int> p = fun(vec, n);

//cout<<p.first<<" "<<p.second<<endl;

if((p.first\*(j-i+1))>=area){

area = p.first\*(j-i+1);

row1 = i; row2 = j;

col1 = p.second-p.first+1; col2 = p.second;

}

}

}

//cout<<area<<" "<<row1<<" "<<col1<<" "<<row2<<" "<<col2<<endl;

vector<vector<int>> res;

if(area>0){

for(int i=row1;i<=row2;i++){

vector<int> temp;

for(int j=col1;j<=col2;j++){

temp.push\_back(a[i][j]);

}

res.push\_back(temp);

}

}

return res;

}

};

# 434. Largest area rectangular sub-matrix with equal number of 1’s and 0’s [ IMP ]

Given a matrix **mat**[][] of size **N** x **M.**The task is to find the largest rectangular sub-matrix whose sum is 0.

**Example 1:**

**Input:** N = 3, M = 3

mat[][] = 1, 2, 3

-3,-2,-1

1, 7, 5

**Output:** 1, 2, 3

-3,-2,-1

**Explanation:** This is the largest sub-matrix,

whose sum is 0.

**Example 2:**

**Input:** N = 4, M = 4

mat[][] = 9, 7, 16, 5

1,-6,-7, 3

1, 8, 7, 9

7, -2, 0, 10

**Output:** -6,-7

8, 7

-2, 0

**Your Task:**  
You don't need to read input or print anything. You just have to complete the function **sumZeroMatrix()** which takes a 2D matrix **mat**[][], its dimensions **N** and **M** as inputs and returns a largest sub-matrix, whose sum is 0.

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

**Constraints**:  
1 <= N, M <= 100  
-1000 <= mat[][] <= 1000

## Solution:

The **naive solution** for this problem is to check every possible rectangle in given 2D array by counting the total number of 1’s and 0’s in that rectangle. This solution requires 4 nested loops and time complexity of this solution would be O(n^4). An **efficient solution** is based on [Largest rectangular sub-matrix whose sum is 0](https://www.geeksforgeeks.org/largest-rectangular-sub-matrix-whose-sum-0/) which reduces the time complexity to O(n^3). First of all consider every ‘0’ in the matrix as ‘-1’. Now, the idea is to reduce the problem to 1-D array. We fix the left and right columns one by one and find the largest sub-array with 0 sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have sum zero) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say temp[]. So temp[i] indicates sum of elements from left to right in row i. If we find largest subarray with 0 sum in temp[], we can get the index positions of rectangular sub-matrix with sum equal to 0 (i.e. having equal number of 1’s and 0’s). With this process we can find the largest area rectangular sub-matrix with sum equal to 0 (i.e. having equal number of 1’s and 0’s). We can use Hashing technique to find maximum length sub-array with sum equal to 0 in 1-D array in O(n) time.

// C++ implementation to find largest area rectangular

// submatrix with equal number of 1's and 0's

#include <bits/stdc++.h>

using namespace std;

#define MAX\_ROW 10

#define MAX\_COL 10

// This function basically finds largest 0

// sum subarray in arr[0..n-1]. If 0 sum

// does't exist, then it returns false. Else

// it returns true and sets starting and

// ending indexes as start and end.

bool subArrWithSumZero(int arr[], int &start,

int &end, int n)

{

// to store cumulative sum

int sum[n];

// Initialize all elements of sum[] to 0

memset(sum, 0, sizeof(sum));

// map to store the indexes of sum

unordered\_map<int, int> um;

// build up the cumulative sum[] array

sum[0] = arr[0];

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

sum[i] = sum[i-1] + arr[i];

// to store the maximum length subarray

// with sum equal to 0

int maxLen = 0;

// traverse to the sum[] array

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

{

// if true, then there is a subarray

// with sum equal to 0 from the

// beginning up to index 'i'

if (sum[i] == 0)

{

// update the required variables

start = 0;

end = i;

maxLen = (i+1);

}

// else if true, then sum[i] has not

// seen before in 'um'

else if (um.find(sum[i]) == um.end())

um[sum[i]] = i;

// sum[i] has been seen before in the

// unordered\_map 'um'

else

{

// if previous subarray length is smaller

// than the current subarray length, then

// update the required variables

if (maxLen < (i-um[sum[i]]))

{

maxLen = (i-um[sum[i]]);

start = um[sum[i]] + 1;

end = i;

}

}

}

// if true, then there is no

// subarray with sum equal to 0

if (maxLen == 0)

return false;

// else return true

return true;

}

// function to find largest area rectangular

// submatrix with equal number of 1's and 0's

void maxAreaRectWithSumZero(int mat[MAX\_ROW][MAX\_COL],

int row, int col)

{

// to store intermediate values

int temp[row], startRow, endRow;

// to store the final outputs

int finalLeft, finalRight, finalTop, finalBottom;

finalLeft = finalRight = finalTop = finalBottom = -1;

int maxArea = 0;

// Set the left column

for (int left = 0; left < col; left++)

{

// Initialize all elements of temp as 0

memset(temp, 0, sizeof(temp));

// Set the right column for the left column

// set by outer loop

for (int right = left; right < col; right++)

{

// Calculate sum between current left

// and right for every row 'i'

// consider value '1' as '1' and

// value '0' as '-1'

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

temp[i] += mat[i][right] ? 1 : -1;

// Find largest subarray with 0 sum in

// temp[]. The subArrWithSumZero() function

// also sets values of finalTop, finalBottom,

// finalLeft and finalRight if there exists

// a subarray with sum 0 in temp

if (subArrWithSumZero(temp, startRow, endRow, row))

{

int area = (right - left + 1) \*

(endRow - startRow + 1);

// Compare current 'area' with previous area

// and accordingly update final values

if (maxArea < area)

{

finalTop = startRow;

finalBottom = endRow;

finalLeft = left;

finalRight = right;

maxArea = area;

}

}

}

}

// if true then there is no rectangular submatrix

// with equal number of 1's and 0's

if (maxArea == 0)

cout << "No such rectangular submatrix exists:";

// displaying the top left and bottom right boundaries

// with the area of the rectangular submatrix

else

{

cout << "(Top, Left): "

<< "(" << finalTop << ", " << finalLeft

<< ")" << endl;

cout << "(Bottom, Right): "

<< "(" << finalBottom << ", " << finalRight

<< ")" << endl;

cout << "Area: " << maxArea << " sq.units";

}

}

// Driver program to test above

int main()

{

int mat[MAX\_ROW][MAX\_COL] = { {0, 0, 1, 1},

{0, 1, 1, 0},

{1, 1, 1, 0},

{1, 0, 0, 1} };

int row = 4, col = 4;

maxAreaRectWithSumZero(mat, row, col);

return 0;

}

Output:

(Top, Left): (0, 0)

(Bottom, Right): (3, 1)

Area: 8 sq.units

Time Complexity: O(n3)

Auxiliary Space: O(n)

# 435. Maximum sum rectangle in a 2D matrix

Given a 2D matrix M of dimensions RxC. Find the maximum sum submatrix in it.

**Example 1:**

**Input:**

R=4

C=5

M=[[1,2,-1,-4,-20],

[-8,-3,4,2,1],

[3,8,10,1,3],

[-4,-1,1,7,-6]]

**Output:**

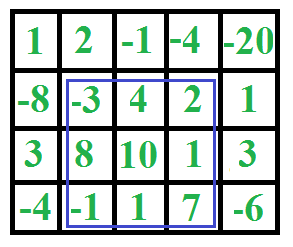
29

**Explanation:**

The matrix is as follows and the

blue rectangle denotes the maximum sum

rectangle.



**Example 2:**

**Input:**

R=2

C=2

M=[[-1,-2],[-3,-4]]

**Output:**

-1

**Explanation:**

Taking only the first cell is the

optimal choice.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **maximumSumRectangle()** which takes the number R, C, and the 2D matrix M as input parameters and returns the maximum sum submatrix.

**Expected Time Complexity:**O(R\*R\*C)  
**Expected Auxillary Space:**O(R\*C)

**Constraints:**  
1<=R,C<=500  
-1000<=M[i][j]<=1000

## Solution:

class Solution {

public:

//Kadane's algorithm

int fun(vector<int> temp, int n){

int sum=0, res = INT\_MIN;

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

sum += temp[i];

res = max(sum, res);

if(sum<0)

sum = 0;

}

return res;

}

int maximumSumRectangle(int R, int C, vector<vector<int>> M) {

// code

int max = INT\_MIN;

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

vector<int> temp(C, 0);

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

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

temp[k] += M[j][k];

int val = fun(temp, C);

if(val>max)

max = val;

}

}

return max;

}

};

**Time Complexity:**O(R\*R\*C)

# 436. Maximum profit by buying and selling a share at most k times

In the stock market, a person buys a stock and sells it on some future date. Given the stock prices of **N**days in an array **A[ ]** and a positive integer **K**, find out the maximum profit a person can make in at-most **K**transactions. A transaction is equivalent to (buying + selling) of a stock and new transaction can start only when the previous transaction has been completed.

**Example 1:**

**Input:** K = 2, N = 6

A = {10, 22, 5, 75, 65, 80}

**Output:** 87

**Explaination:**

1st transaction: buy at 10 and sell at 22.

2nd transaction : buy at 5 and sell at 80.

**Example 2:**

**Input:** K = 3, N = 4

A = {20, 580, 420, 900}

**Output:** 1040

**Explaination:** The trader can make at most 2

transactions and giving him a profit of 1040.

**Example 3:**

**Input:** K = 1, N = 5

A = {100, 90, 80, 50, 25}

**Output:** 0

**Explaination:** Selling price is decreasing

daily. So seller cannot have profit.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **maxProfit()**which takes the values K, N and the array A[] as input parameters and returns the maximum profit.

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

**Constraints:**  
1 ≤ N ≤ 500  
1 ≤ K ≤ 200  
1 ≤ A[ i ] ≤ 1000

## Solution:

In this post, we are only allowed to make at max k transactions. The problem can be solved by using dynamic programming.   
Let **profit[t][i]** represent maximum profit using at most t transactions up to day i (including day i). Then the relation is:  
profit[t][i] = max(profit[t][i-1], max(price[i] – price[j] + profit[t-1][j]))   
          for all j in range [0, i-1]   
profit[t][i] will be maximum of –

1. profit[t][i-1] which represents not doing any transaction on the ith day.
2. Maximum profit gained by selling on ith day. In order to sell shares on ith day, we need to purchase it on any one of [0, i – 1] days. If we buy shares on jth day and sell it on ith day, max profit will be price[i] – price[j] + profit[t-1][j] where j varies from 0 to i-1. Here profit[t-1][j] is best we could have done with one less transaction till jth day.

Below is Dynamic Programming based implementation.

// C++ program to find out maximum profit by

// buying and selling a share atmost k times

// given stock price of n days

#include <climits>

#include <iostream>

using namespace std;

// Function to find out maximum profit by buying

// & selling a share atmost k times given stock

// price of n days

int maxProfit(int price[], int n, int k)

{

// table to store results of subproblems

// profit[t][i] stores maximum profit using

// atmost t transactions up to day i (including

// day i)

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

// For day 0, you can't earn money

// irrespective of how many times you trade

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

profit[i][0] = 0;

// profit is 0 if we don't do any transaction

// (i.e. k =0)

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

profit[0][j] = 0;

// fill the table in bottom-up fashion

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

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

int max\_so\_far = INT\_MIN;

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

max\_so\_far = max(max\_so\_far,

price[j] - price[m] + profit[i - 1][m]);

profit[i][j] = max(profit[i][j - 1], max\_so\_far);

}

}

return profit[k][n - 1];

}

// Driver code

int main()

{

int k = 2;

int price[] = { 10, 22, 5, 75, 65, 80 };

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

cout << "Maximum profit is: "

<< maxProfit(price, n, k);

return 0;

}

**Output :**

Maximum profit is: 87

**Optimized Solution:**   
The above solution has time complexity of O(k.n2). It can be reduced if we are able to calculate the maximum profit gained by selling shares on the ith day in constant time.  
profit[t][i] = max(profit [t][i-1], max(price[i] – price[j] + profit[t-1][j]))   
                            for all j in range [0, i-1]  
If we carefully notice,   
max(price[i] – price[j] + profit[t-1][j])   
for all j in range [0, i-1]  
can be rewritten as,   
= price[i] + max(profit[t-1][j] – price[j])   
for all j in range [0, i-1]   
= price[i] + max(prevDiff, profit[t-1][i-1] – price[i-1])   
where prevDiff is max(profit[t-1][j] – price[j])   
for all j in range [0, i-2]  
So, if we have already calculated max(profit[t-1][j] – price[j]) for all j in range [0, i-2], we can calculate it for j = i – 1 in constant time. In other words, we don’t have to look back in the range [0, i-1] anymore to find out best day to buy. We can determine that in constant time using below revised relation.  
profit[t][i] = max(profit[t][i-1], price[i] + max(prevDiff, profit [t-1][i-1] – price[i-1])   
where prevDiff is max(profit[t-1][j] – price[j]) for all j in range [0, i-2]  
Below is its optimized implementation –

// C++ program to find out maximum profit by buying

// and/ selling a share atmost k times given stock

// price of n days

#include <climits>

#include <iostream>

using namespace std;

// Function to find out maximum profit by buying &

// selling/ a share atmost k times given stock price

// of n days

int maxProfit(int price[], int n, int k)

{

// table to store results of subproblems

// profit[t][i] stores maximum profit using atmost

// t transactions up to day i (including day i)

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

// For day 0, you can't earn money

// irrespective of how many times you trade

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

profit[i][0] = 0;

// profit is 0 if we don't do any transaction

// (i.e. k =0)

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

profit[0][j] = 0;

// fill the table in bottom-up fashion

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

int prevDiff = INT\_MIN;

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

prevDiff = max(prevDiff,

profit[i - 1][j - 1] - price[j - 1]);

profit[i][j] = max(profit[i][j - 1],

price[j] + prevDiff);

}

}

return profit[k][n - 1];

}

// Driver Code

int main()

{

int k = 3;

int price[] = { 12, 14, 17, 10, 14, 13, 12, 15 };

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

cout << "Maximum profit is: "

<< maxProfit(price, n, k);

return 0;

}

Output :

Maximum profit is: 12

# 437. Find if a string is interleaved of two other strings

Given strings **A**, **B**, and **C**, find whether **C** is formed by an interleaving of **A** and **B**.

An interleaving of two strings S and T is a configuration such that it creates a new string Y from the concatenation substrings of A and B and |Y| = |A + B| = |C|

For example:

A = "XYZ"

B = "ABC"

so we can make multiple interleaving string Y like, XYZABC, XAYBCZ, AXBYZC, XYAZBC and many more so here your task is to check whether you can create a string Y which can be equal to C.

Specifically, you just need to create substrings of string A and create substrings B and concatenate them and check whether it is equal to C or not.

Note: **a + b** is the concatenation of strings a and b.

Return **true** if **C** is formed by an interleaving of **A** and **B**, else return **false.**

**Example 1:**

**Input:**

A = YX, B = X, C = XXY

**Output:** 0

**Explanation:** XXY is not interleaving

of YX and X

**Example 2:**

**Input:**

A = XY, B = X, C = XXY

**Output:** 1

**Explanation:** XXY is interleaving of

XY and X.

**Your Task:**  
Complete the function **isInterleave()**which takes three strings A, B and C as input and returns true if C is an interleaving of A and B else returns false. (1 is printed by the driver code if the returned value is true, otherwise 0.)

**Expected Time Complexity:** O(N \* M)  
**Expected Auxiliary Space:** O(N \* M)  
here, N = length of A, and M = length of B

**Constraints:**  
1 ≤ length of A, B ≤ 100  
1 ≤ length of C ≤ 200

## Solution:

**Method 1:** Recursion.   
**Approach:** A simple solution is discussed here: [Check whether a given string is an interleaving of two other given string](https://www.geeksforgeeks.org/check-whether-a-given-string-is-an-interleaving-of-two-other-given-strings/).   
The simple solution doesn’t work if the strings A and B have some common characters. For example, let the given string be A and the other strings be B and C. Let A = “XXY”, string B = “XXZ” and string C = “XXZXXXY”. Create a recursive function that takes parameters A, B, and C. To handle all cases, two possibilities need to be considered.

1. If the first character of C matches the first character of A, we move one character ahead in A and C and recursively check.
2. If the first character of C matches the first character of B, we move one character ahead in B and C and recursively check.

If any of the above function returns true or A, B and C are empty then return true else return false.

// A simple recursive function to check

// whether C is an interleaving of A and B

bool isInterleaved(

char\* A, char\* B, char\* C)

{

// Base Case: If all strings are empty

if (!(\*A || \*B || \*C))

return true;

// If C is empty and any of the

// two strings is not empty

if (\*C == '\0')

return false;

// If any of the above mentioned

// two possibilities is true,

// then return true, otherwise false

return ((\*C == \*A) && isInterleaved(

A + 1, B, C + 1))

|| ((\*C == \*B) && isInterleaved(

A, B + 1, C + 1));

}

**Complexity Analysis:**

* **Time Complexity:**O(2^n), where n is the length of the given string.   
  We need to iterate the whole string only once hence this is possible in O(n).
* **Space Complexity:**O(1).   
  The space complexity is constant.

**Method 2:**Dynamic Programming.   
**Approach:** The above recursive solution certainly has many overlapping sub-problems. For example, if we consider A = “XXX”, B = “XXX” and C = “XXXXXX” and draw a recursion tree, there will be many overlapping subproblems. Therefore, like any other typical [Dynamic Programming problems](https://www.geeksforgeeks.org/tag/dynamic-programming/), we can solve it by creating a table and store results of sub-problems in a **bottom-up manner**. The top-down approach of the above solution can be modified by adding a Hash Map.

**Algorithm:**

1. Create a DP array (matrix) of size M\*N, where m is the size of the first string and n is the size of the second string. Initialize the matrix to false.
2. If the sum of sizes of smaller strings is not equal to the size of the larger string then return false and break the array as they cant be the interleaved to form the larger string.
3. Run a nested loop the outer loop from 0 to m and the inner loop from 0 to n. Loop counters are i and j.
4. If the values of i and j are both zeroes then mark dp[i][j] as true. If the value of i is zero and j is non zero and the j-1 character of B is equal to j-1 character of C the assign dp[i][j] as dp[i][j-1] and similarly if j is 0 then match i-1 th character of C and A and if it matches then assign dp[i][j] as dp[i-1][j].
5. Take three characters x, y, z as (i-1)th character of A and (j-1)th character of B and (i + j – 1)th character of C.
6. if x matches with z and y does not match with z then assign dp[i][j] as dp[i-1][j] similarly if x is not equal to z and y is equal to z then assign dp[i][j] as dp[i][j-1]
7. if x is equal to y and y is equal to z then assign dp[i][j] as bitwise OR of dp[i][j-1] and dp[i-1][j].
8. return value of dp[m][n].

// A Dynamic Programming based program

// to check whether a string C is

// an interleaving of two other

// strings A and B.

#include <iostream>

#include <string.h>

using namespace std;

// The main function that

// returns true if C is

// an interleaving of A

// and B, otherwise false.

bool isInterleaved(

char\* A, char\* B, char\* C)

{

// Find lengths of the two strings

int M = strlen(A), N = strlen(B);

// Let us create a 2D table

// to store solutions of

// subproblems. C[i][j] will

// be true if C[0..i+j-1]

// is an interleaving of

// A[0..i-1] and B[0..j-1].

bool IL[M + 1][N + 1];

// Initialize all values as false.

memset(IL, 0, sizeof(IL));

// C can be an interleaving of

// A and B only of the sum

// of lengths of A & B is equal

// to the length of C.

if ((M + N) != strlen(C))

return false;

// Process all characters of A and B

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

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

// two empty strings have an

// empty string as interleaving

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

IL[i][j] = true;

// A is empty

else if (i == 0) {

if (B[j - 1] == C[j - 1])

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

}

// B is empty

else if (j == 0) {

if (A[i - 1] == C[i - 1])

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

}

// Current character of C matches

// with current character of A,

// but doesn't match with current

// character of B

else if (

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

&& B[j - 1] != C[i + j - 1])

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

// Current character of C matches

// with current character of B,

// but doesn't match with current

// character of A

else if (

A[i - 1] != C[i + j - 1]

&& B[j - 1] == C[i + j - 1])

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

// Current character of C matches

// with that of both A and B

else if (

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

&& B[j - 1] == C[i + j - 1])

IL[i][j]

= (IL[i - 1][j]

|| IL[i][j - 1]);

}

}

return IL[M][N];

}

// A function to run test cases

void test(char\* A, char\* B, char\* C)

{

if (isInterleaved(A, B, C))

cout << C << " is interleaved of "

<< A << " and " << B << endl;

else

cout << C << " is not interleaved of "

<< A << " and " << B << endl;

}

// Driver program to test above functions

int main()

{

test("XXY", "XXZ", "XXZXXXY");

test("XY", "WZ", "WZXY");

test("XY", "X", "XXY");

test("YX", "X", "XXY");

test("XXY", "XXZ", "XXXXZY");

return 0;

}

**Output:**

XXZXXXY is not interleaved of XXY and XXZ

WZXY is interleaved of XY and WZ

XXY is interleaved of XY and X

XXY is not interleaved of YX and X

XXXXZY is interleaved of XXY and XXZ

**Complexity Analysis:**

* **Time Complexity:**O(M\*N).   
  Since a traversal of DP array is needed, so the time complexity is O(M\*N).
* **Space Complexity:**O(M\*N).   
  This is the space required to store the DP array.

https://youtu.be/WBXy-sztEwI   
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Method 3:** Dynamic Programming(Memoization)

**Approach:**We can make a matrix where rows and columns represent the characters of the string A and B. If C is the interleaved string of A and B then there exist a path from top left of the Matrix to bottom right. That is if we can go from index 0,0 to n,m while matching characters of all A and B with C then C is interleaved of A and B.

Let A be “XXY”, B be “XXZ” and C be “XXZXXY” then the path would look something like this:

|  | X | X | Y |  |
| --- | --- | --- | --- | --- |
| X | 1 | 0 | 0 | 0 |
| X | 1 | 0 | 0 | 0 |
| Z | 1 | 0 | 0 | 0 |
|  | 1 | 1 | 1 | R |

let us consider one more example, let A be “ABC”, B be “DEF” and C be “ADBECF”, then path would look something like this:

|  | D | E | F |  |
| --- | --- | --- | --- | --- |
| A | 1 | 0 | 0 | 0 |
| B | 1 | 1 | 0 | 0 |
| C | 0 | 1 | 1 | 0 |
|  | 0 | 0 | 1 | R |

If there exist a path through which we can reach R, then C is the interleaved strings of A and B.

**Algorithm:**

1. We will first create a matrix dp to store the path since one path can be explore multiple time, the Matrix index dp[i][j] will store if there exist a path from this index or not.

2. If we are at i’th index of A and j’th index of B and C[i+j] matches both A[i] and B[j] then we explore both the paths that is we will go right and down i.e. we will explore index i+1,j and j+1,i.

3. If C[i+j] only matches with A[i] or B[j] then we will go just down or right respectively that is i+1,j or i,j+1.

// A Memoization program

// to check whether a string C is

// an interleaving of two other

// strings A and B.

#include <iostream>

#include <string.h>

using namespace std;

// Declare n,m for storing the size of the strings.

int n, m;

// Declare dp array

int dp[101][101];

// declaration of dfs function.

bool dfs(int i, int j, string &A, string &B, string &C);

// The main function that

// returns true if C is

// an interleaving of A

// and B, otherwise false.

bool isInterleave(string A, string B, string C)

{

// Strong the length in n,m

n = A.length(), m = B.length();

// C can be an interleaving of

// A and B only of the sum

// of lengths of A & B is equal

// to the length of C.

if ((n + m) != C.length())

return 0;

// initializing dp array with -1

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

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

dp[i][j] = -1;

// calling and returning the answer

return dfs(0, 0, A, B, C);

}

// This function checks if there exist a valid path from 0,0 to n,m

bool dfs(int i, int j, string &A, string &B, string &C)

{

// If path has already been calculated from this index

// then return calculated value.

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

return dp[i][j];

// If we reach the destination return 1

if (i == n && j == m)

return 1;

// If C[i+j] matches with both A[i] and B[j]

// we explore both the paths

if (i < n && A[i] == C[i + j] && j < m && B[j] == C[i + j])

{

// go down and store the calculated value in x

// and go right and store the calculated value in y.

int x = dfs(i + 1, j, A, B, C), y = dfs(i, j + 1, A, B, C);

// return the best of both.

return dp[i][j] = x | y;

}

// If C[i+j] matches with A[i].

if (i < n && A[i] == C[i + j])

{

// go down

int x = dfs(i + 1, j, A, B, C);

// Return the calculated value.

return dp[i][j] = x;

}

// If C[i+j] matches with B[j].

if (j < m && B[j] == C[i + j])

{

int y = dfs(i, j + 1, A, B, C);

// Return the calculated value.

return dp[i][j] = y;

}

// if nothing matches we return 0

return dp[i][j] = 0;

}

// A function to run test cases

void test(string A, string B, string C)

{

if (isInterleave(A, B, C))

cout << C << " is interleaved of "

<< A << " and " << B << endl;

else

cout << C << " is not interleaved of "

<< A << " and " << B << endl;

}

// Driver program to test above functions

int main()

{

test("XXY", "XXZ", "XXZXXXY");

test("XY", "WZ", "WZXY");

test("XY", "X", "XXY");

test("YX", "X", "XXY");

test("XXY", "XXZ", "XXXXZY");

test("ACA", "DAS", "DAACSA");

return 0;

}

**Output**

XXZXXXY is not interleaved of XXY and XXZ

WZXY is interleaved of XY and WZ

XXY is interleaved of XY and X

XXY is not interleaved of YX and X

XXXXZY is interleaved of XXY and XXZ

DAACSA is interleaved of ACA and DAS

**Complexity Analysis:**

Time Complexity:  **O(m\*n).**

This is the worst case time complexity, if the given strings contain no common character matching with C then time complexity will be O(n+m).  
Space Complexity: **O(m\*n).**

This is the space required to store the DP array.

# 438. Maximum Length of Pair Chain

You are given an array of n pairs pairs where pairs[i] = [lefti, righti] and lefti < righti.

A pair p2 = [c, d] **follows** a pair p1 = [a, b] if b < c. A **chain** of pairs can be formed in this fashion.

Return *the length longest chain which can be formed*.

You do not need to use up all the given intervals. You can select pairs in any order.

**Example 1:**

**Input:** pairs = [[1,2],[2,3],[3,4]]

**Output:** 2

**Explanation:** The longest chain is [1,2] -> [3,4].

**Example 2:**

**Input:** pairs = [[1,2],[7,8],[4,5]]

**Output:** 3

**Explanation:** The longest chain is [1,2] -> [4,5] -> [7,8].

**Constraints:**

* n == pairs.length
* 1 <= n <= 1000
* -1000 <= lefti < righti <= 1000

## Solution:

#### Approach #1: Dynamic Programming [Accepted]

**Intuition**

If a chain of length k ends at some pairs[i], and pairs[i][1] < pairs[j][0], we can extend this chain to a chain of length k+1.

**Algorithm**

Sort the pairs by first coordinate, and let dp[i] be the length of the longest chain ending at pairs[i]. When i < j and pairs[i][1] < pairs[j][0], we can extend the chain, and so we have the candidate answer dp[j] = max(dp[j], dp[i] + 1).

class Solution {

public int findLongestChain(int[][] pairs) {

Arrays.sort(pairs, (a, b) -> a[0] - b[0]);

int N = pairs.length;

int[] dp = new int[N];

Arrays.fill(dp, 1);

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

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

if (pairs[i][1] < pairs[j][0])

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

}

}

int ans = 0;

for (int x: dp) if (x > ans) ans = x;

return ans;

}

}

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2) where N*N* is the length of pairs. There are two for loops, and N^2*N*2 dominates the sorting step.
* Space Complexity: O(N)*O*(*N*) for sorting and to store dp.

#### Approach #2: Greedy [Accepted]

**Intuition**

We can greedily add to our chain. Choosing the next addition to be the one with the lowest second coordinate is at least better than a choice with a larger second coordinate.

**Algorithm**

Consider the pairs in increasing order of their second coordinate. We'll try to add them to our chain. If we can, by the above argument we know that it is correct to do so.

class Solution {

public int findLongestChain(int[][] pairs) {

Arrays.sort(pairs, (a, b) -> a[1] - b[1]);

int cur = Integer.MIN\_VALUE, ans = 0;

for (int[] pair: pairs) if (cur < pair[0]) {

cur = pair[1];

ans++;

}

return ans;

}

}

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*) where N*N* is the length of S. The complexity comes from the sorting step, but the rest of the solution does linear work.
* Space Complexity: O(N)*O*(*N*). The additional space complexity of storing cur and ans, but sorting uses O(N)*O*(*N*) space. Depending on the implementation of the language used, sorting can sometimes use less space.