Striver SDE Sheet

[1 Arrays 2](#_Toc124107508)

[1.1 Set Matrix Zeroes 2](#_Toc124107509)

[1.2 Pascal's Triangle 2](#_Toc124107510)

[1.3 Next Permutation 3](#_Toc124107511)

[1.4 Kadane’s Algorithm 4](#_Toc124107512)

[1.5 Sort an array of 0s, 1s and 2s 5](#_Toc124107513)

## Arrays

### Set Matrix Zeroes

**Problem Statement :** Given a matrix if an element in the matrix is 0 then you will have to set its entire column and row to 0 and then return the matrix.

**Brute Force :**

Traverse through the matrix and if you find an element with value 0, then change all the elements in its row and column to -1, except when an element is 0. The reason for not changing other elements to 0, but -1, is because that might affect other columns and rows. Now traverse through the matrix again and if an element is -1 change it to 0.

**Time Complexity:**O((N\*M)\*(N + M)). O(N\*M) for traversing through each element and (N+M)for traversing to row and column of elements having value 0.

**Space Complexity:** O(1)

**Better :**

Take two dummy array one of size of row and other of size of column.Now traverse through the array.If matrix[i][j]==0 then set dummy1[i]=0(for row) and dummy2[j]=0(for column).Now traverse through the array again and if dummy1[i]==0 || dummy2[j]==0 then arr[i][j]=0,else continue.

**Time Complexity**: O(N\*M + N\*M) for two times traversing the matrix .

**Space Complexity**: O(N) for the two dummy arrays .

**Optimal :**

Instead of taking two separate dummy array,take first row and column of the matrix as the array for checking whether the particular column or row has the value 0 or not.Since matrix[0][0] are overlapping.Therefore take separate variable col0(say) to check if the 0th column has 0 or not and use matrix[0][0] to check if the 0th row has 0 or not.Now traverse from last element to the first element and check if matrix[i][0]==0 || matrix[0][j]==0 and if true set matrix[i][j]=0,else continue.

**Note :** While traversing for the second time the first row and column will be computed first, which will affect the values of further elements that’s why we traversing in the reverse direction.

Take below example to understand reverse case

1 1 1

0 1 1

1 1 1

**Time Complexity:** O(2\*(N\*M)), as we are traversing two times in a matrix,

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

### Pascal's Triangle

**Approach :** Straight foreward

vector**<**vector**<**int**>>** generate**(**int numRows**)** **{**

vector**<**vector**<**int**>>** pascalMat **;**

**for(**int r**=**0 **;** r**<**numRows **;** r**++** **){**

vector**<**int**>** currentRow**(**r**+**1**,**0**)** **;**

currentRow**[**0**]** **=** currentRow**[**r**]** **=** 1 **;**

**for(**int c**=**1 **;** c**<=**r**-**1 **;** c**++** **){**

currentRow**[**c**]** **=** pascalMat**[**r**-**1**][**c**-**1**]** **+** pascalMat**[**r**-**1**][**c**]** **;**

**}**

pascalMat**.**push\_back**(**currentRow**)** **;**

**}**

**return** pascalMat **;**

**}**

**Time Complexity:** We are creating a 2D array of size (numRows \* numCols) (where 1 <= numCols <= numRows), and we are traversing through each of the cells to update it with its correct value, so Time Complexity = O(numRows2).

**Space Complexity:** Since we are creating a 2D array, space complexity = O(numRows2).

### Next Permutation

**Problem Statement:** Given an array Arr[] of integers, rearrange the numbers of the given array into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

**Brute Force :**

Find all possible permutations of elements present and store them.Search input from all possible permutations.Print the next permutation present right after it.

**Time Complexity :**

For finding, all possible permutations, it is taking N!xN. N represents the number of elements present in the input array. Also for searching input arrays from all possible permutations will take N!. Therefore, it has a Time complexity of O(N!xN).

**Space Complexity :**

Since we are not using any extra spaces except stack spaces for recursion calls. So, it has a space complexity of O(1).

**Optimal :**

For any dictionary order , it will be always increasing from the back (atleast 1 element should follow this pattern Ex in 1 2 3 , ‘3’ is such pattern ) and

For 1 3 5 4 2 , ‘5 4 2' is such a pattern , this indicates that pattern part is in it’s max possible can not derive the next greater number from this pattern . To do so we need to start involving before digit where the pattern breaks first , here it is at ‘3’ .

i.e bp = A[i] < A[i+1] , now we have ‘1’ and ‘3 5 4 2’ , now second part can be used to generate the next greater number . we know that ’5 4 2’ is in increasing order from back , we just need to find out digit which is just greater than ‘3’ in this pattern so that we can swap with that .

A[sw] > A[bp] , ‘4’ is such number ( we are taking this just greater for generating the immediate next lexographical number ) .

After swapping it will look like ‘4 5 3 2’ , but this is not actually the immediate next lexographical number as ‘5 3 2’ stil in increasing order from back . so here we should reverse this .

We’ll get ‘4 2 3 5’ and finally ‘1 4 2 3 5’ is our answer .

Edge case : If the whole number is already greater like ‘3 2 1’ , there is no break point simply reverse whole number .

**Time Complexity:** This sums up to 3\*O(N) which is approximately O(N).

**Space Complexity:** Since no extra storage is required. Thus, its complexity is O(1).

### Kadane’s Algorithm

**Problem Statement**: Given an integer array arr, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum and print the subarray.

(Need to ask the interviewer that if we have to return 0 incase all the elements are negative)

**Brute Force :**

We can do this problem using nested for loop maintaining two variables i and j to find the sum of every possible slice of the array and then find the maximum sum.

**Time Complexity:** O(N^2)

**Space Complexity:** Since no extra storage is required. Thus, its complexity is O(1).

**Best Approach:** Using Kadanes Algorithm.

Take two variables **max\_so\_far** and **curr\_max** (or max\_ending\_here).

Here **max\_so\_far** will return the final answer. We can initialize both **max\_so\_far , curr\_max** to the first element of the array.

**max\_so\_far = curr\_max = arr[0] ,** but incase we need to return the 0 for all the -ve elements in the array we can initialize **max\_so\_far =0**

For every incoming element in the array starting from the index 1, we need to see if adding **arr[i],** to the

**curr\_max** will be better than **arr[i]** itself if so, we will continue the streak, otherwise we need to break the streak and start considering the **curr\_max** from **arr[i].**

*when to break streak?*

If the **curr\_max** is already negative no matter what incoming element **arr[i],** we should break the streak.

As long as **curr\_max** is positive we will continue the streak even if incoming element **arr[i]** is negative (we hope for the better sum in the coming sequence)

At every step we need to recaulate **max\_so\_far,** by looking at the **curr\_max** if that better our answer.

int maxSubArray**(**vector**<**int**>&** nums**)** **{**

int max\_so\_far **=** nums**[**0**];**

int curr\_max **=** nums**[**0**];**

**for** **(**int i **=** 1**;** i **<** nums**.**size**();** i**++)**

**{**

curr\_max **=** max**(**nums**[**i**],** curr\_max**+**nums**[**i**]);**

max\_so\_far **=** max**(**max\_so\_far**,** curr\_max**);**

**}**

**return** max\_so\_far**;**

**}**

**Time Complexity:** O(N)

**Space Complexity**:O(1)

### Sort an array of 0s, 1s and 2s

**Problem Statement**: Given an array consisting of only 0s, 1s and 2s. Write a program to in-place sort the array without using inbuilt sort functions. (Expected: Single pass-O(N) and constant space)

**Brute Force:**

Sorting (even if it is not the expected solution here but it can be considered as one of the approaches).

**Time Complexity:** O(N x Log N) or O(N^2)

**Space Complexity**:O(1)

**Better Approach:** Keeping count of values

Since in this case there are only 3 distinct values in the array so it’s easy to maintain the count of all, Like the count of 0, 1, and 2. This can be followed by overwriting the array based on the frequency(count) of the values.

Time Complexity: O(N) + O(N)

Space Complexity: O(1)

**Best Approach: Dutch National flag algorithm (**3 pointer approach**)**

Here we will maintain three pointers

next\_zero -> represents the index where the next zero should be placed

next\_two -> represents the index where the next two should be placed

curr\_pos-> this is an iterator for the array and will traverse from start.

The primary goal here is to move 0s to the left and 2s to the right of the array and at the same time all the 1s shall be in the middle region of the array and hence the array will be sorted.

Whenever we encounter 0 at curr\_pos, we swap it with next\_zero index and increment next\_zero by 1.

No need to decrement curr\_pos as we are traversing from the begining only we would have

known that already for sure that it is 1.

Whenever we encounter 2 at curr\_pos, we swap it with next\_two index and decrement next\_two by 1.

We need to decrement curr\_pos since we dont know what is already there in next\_two position that's why we need to re visit that curr\_pos again.

We will continue this process till curr\_pos cross over the next\_two.

### Stock Buy And Sell

**Problem Statement:** You are given an array of prices where prices[i] is the price of a given stock on an ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

**Brute Force:**

We can simply use 2 loops and track of tranaction difference (If arr[j] > arr[i]) and maintain a variable maxPro to contain the max value among all transactions.

Time complexity: O(n^2)

Space Complexity: O (1)

**Optimal/Best solution:**

Intuition: We will linearly travel the array. We can maintain a minimum from the starting of the array and compare it with every element of the array, if it is greater than the minimum then takes the difference and maintain it in max, otherwise update the minimum.

Here min\_val should always preceed the curr\_pos element, means should be always likely to be subtracted, as that makes the difference positive.

Time complexity: O(n)

Space Complexity: O (1)