# 

## 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).