Striver SDE Sheet

[1 Arrays I 2](#_Toc125152353)

[1.1 Set Matrix Zeroes 2](#_Toc125152354)

[1.2 Pascal's Triangle 2](#_Toc125152355)

[1.3 Next Permutation 3](#_Toc125152356)

[1.4 Kadane’s Algorithm 4](#_Toc125152357)

[1.5 Sort an array of 0s, 1s and 2s 5](#_Toc125152358)

[1.6 Stock Buy And Sell 6](#_Toc125152359)

[2 Arrays II 6](#_Toc125152360)

[2.1 Rotate Image by 90 degrees 6](#_Toc125152361)

[2.2 Merge Overlapping Sub-intervals 7](#_Toc125152362)

[2.3 Find the duplicate in an array of N+1 integers 7](#_Toc125152363)

[2.4 Merge two Sorted Arrays Without Extra Space 9](#_Toc125152364)

[2.5 Repeating and Missing numbers 10](#_Toc125152365)

[2.6 Count inversions in an array 11](#_Toc125152366)

[3 Arrays III 12](#_Toc125152367)

[3.1 Search in a sorted 2D matrix 12](#_Toc125152368)

[3.2 Implement Pow(x,n) | X raised to the power N 14](#_Toc125152369)

[3.3 Find the Majority Element that occurs more than N/2 times 14](#_Toc125152370)

[3.4 Majority Elements(>N/3 times) 14](#_Toc125152371)

[3.5 Grid Unique Paths 15](#_Toc125152372)

[3.6 Count Reverse Pairs 16](#_Toc125152373)

[4 Arrays IV 16](#_Toc125152374)

[4.1 Two Sum 16](#_Toc125152375)

## Arrays I

### 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)

## Arrays II

### Rotate Image by 90 degrees

**Problem Statement:** Given a matrix, your task is to rotate the matrix 90 degrees clockwise.

**Naive approach** : Taking the help of dummy auxillary matrix.

Time Complexity: O(N\*N) to linearly iterate and put it into some other matrix.

Space Complexity: O(N\*N) to copy it into some other matrix.

**Best approach:**

Intuition: By observation, we see that the first column of the original matrix is the reverse of the first row of the rotated matrix, so that’s why we transpose the matrix and then reverse each row, and since we are making changes in the matrix itself space complexity gets reduced to O(1).

Time Complexity: O(N\*N) + O(N\*N).One O(N\*N) for transposing the matrix and the other for reversing.

Space Complexity: O(1).

### Merge Overlapping Sub-intervals

**Problem Statement:** Given an array of intervals, merge all the overlapping intervals and return an array of non-overlapping intervals.

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

**Best solution:** (brute force was also very close to this approach)

First ask interviewer if the intervals are already sorted on starting time, if not we need to sort them. we can use comparator function as an argument to sort if needed.

Intuition: Since we have sorted the intervals, the intervals which will be merging are bound to be adjacent. We kept on merging simultaneously as we were traversing through the array and when the element was non-overlapping, we simply inserted the element in our data structure.

Take two variables *merge\_start, merge\_end* initially representing first interval, we need to traverse the remaining list of intervals from the first index of interval list.

The overlapping condition will be *current\_start <= merge\_end* in which case we need to check for which is greater between *merge\_end, current\_end,* because cases like { [1,10] , [2 , 6 ] , [3 ,8] } also possible.

So, *merge\_end = max (merge\_end, current\_end).*

So the variable merge*\_end* alone keeps changing while we traverse.

In case of non overlapping condition , we simply insert the sofar formed merged interval { *merge\_start, merge\_end* } to our answer and reset their values to the current non overlapping interval.

At the end we just need to insert the pair {*merge\_start, merge\_end* } to our answer as the loop finishes,this pair will still remain.

**Time Complexity**: O(NlogN) + O(N). O(NlogN) for sorting and O(N) for traversing through the array.

**Space Complexity:** O(N) to return the answer of the merged intervals.

### Find the duplicate in an array of N+1 integers

Problem Statement: Given an array of N + 1 size, where each element is between 1 and N. Assuming there is only one duplicate number, your task is to find the duplicate number.

**Brute Force:**

Sort the array. After that, if there is any duplicate number they will be adjacent. So we simply have to check if arr[i]==arr[i+1] and if it is true,arr[i] is the duplicate number.

Time Complexity:O(Nlogn + N)

Reason: NlogN for sorting the array and O(N) for traversing through the array and checking if adjacent elements are equal or not. But this will distort the array.

Space Complexity:O(1)

**Better Appraoch:**

**i)Using frequency array** : Here since we know that the elements are within the range of 1 to N we can make use of that fact . Take a frequency array of size N+1 and initialize it to 0. Now traverse through the array and if the frequency of the element is 0 increase it by 1, else if the frequency is not 0 then that element is the required answer.

Time Complexity: O(N), as we are traversing through the array only once.

Space Complexity: O(N), as we are using extra space for frequency array.

**ii)Use map or set:** We can traverse through the array if we find that an element does not exist in the map, just push that to it, but if it does it is our duplicate.

Time Complexity: O(N), as we are traversing through the array only once.

Space Complexity: O(N), as we are using extra space for map.

**Best Approach:**

**i)Using xor method**: since we know elements lie in range 1 to N, first we can take xor of all 1 to N elements and then we traverse the array of N+1 elemets and keep adding the elements to the xor result therefore, all the same pairs will get cancelled and the duplicate element will remain as xor result.

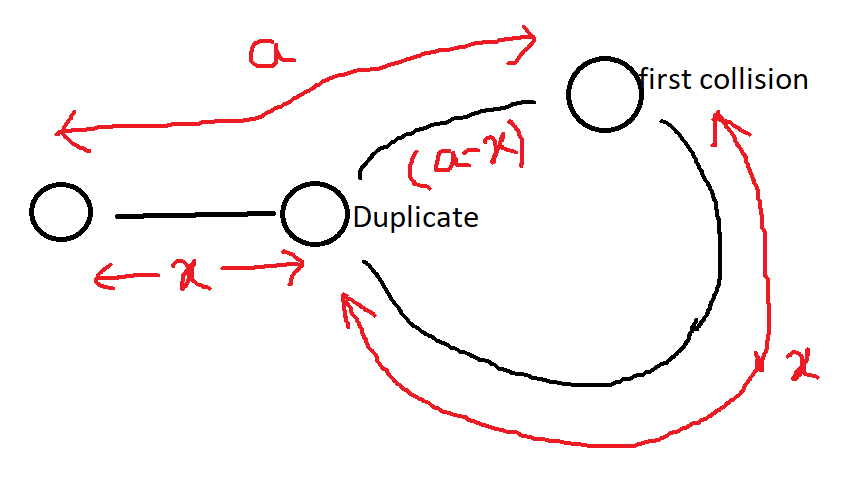
Time Complexity: O(N) + O (N), as we are traversing through the array once and for finding xor of 1 to N second time.

Space Complexity: O (1), as we are using constant space.

**ii)Using cycle detection method:**

As the size of the array is N+1 and the elements will be from [1, N] the indices should be from [0, N] , so the elements can be indexible.

**Intuition**: Since there is a duplicate number, we can always say that cycle will be formed.



The slow pointer moves by one step and the fast pointer moves by 2 steps and there exists a cycle so the first collision is bound to happen.

Let’s assume the distance between the first element and the first collision is ‘a’. So slow pointer has traveled a distance while fast(since moving 2 steps at a time) has traveled ‘2a’ distance. For slow and a fast pointer to collide 2a-a=a should be multiple of the length of cycle, Now we place a fast pointer to start.

Assume the distance between the start and duplicate to be x. So now the distance between slow and duplicate shows also to be x, as seen from the diagram, and so now fast and slow pointer both should move by one step.

**Inshort:**

1)Find the first collision.

2)Begin from start and slow\_ptr from the collision and travel until they collide which is out duplicate.

### Merge two Sorted Arrays Without Extra Space

Problem statement: Given two sorted arrays arr1[] and arr2[] of sizes n and m in non-decreasing order. Merge them in sorted order. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.

**Brute Force :**

We can use a new array of size n+m and put all elements of arr1 and arr2 in it, and sort it. After sorting it, put back all the

Time complexity: O((m+n)\*log(m+n))+O(m)+O(n)

Space Complexity: O(m+n)

**Better Approach:**

Use the technique followed in the merge sort, while merging two sorted arrays. Use variables i, j and k.

Use a temp array to store the merge of the elements and put back elements in arr1 and arr2.after merging.

Time complexity: O(m+n)

Space Complexity: O(m+n)

**Best Approach:** Without using space

**i)Gap Method or Shell sorting method**

Just assume that we are traversing the single array of length (m+n).

**Approach:**

* Initially take the gap as (m+n)/2;
* Take as a pointer1 = 0 and pointer2 = gap.
* Run a oop from pointer1 & pointer2 to m+n and whenever arr[pointer2]<arr[pointer1], just swap those.
* After completion of the loop reduce the gap as gap=gap/2.
* Repeat the process until gap>0.

Time complexity: O(m+n) (log(m+n)) as we are dividing the array of size (m+n) at every stage by 2 and iterating the whole.

Space complexity: O (1)

**ii)Compare from the back**

Intuition: To compare two sorted arrays [7, 9, 0, 0, 0] and [1, 2, 8] If we compare from end, we would be able to fill up the last 0's spaces with greater elements and at last what ever left in arr2 can be copied to the front of arr1.

### Repeating and Missing numbers

You are given an array of size ‘N’. The elements of the array are in the range from 1 to ‘N’. Ideally, the array should contain elements from 1 to ‘N’. But due to some miscalculations, there is a number R in the range [1, N] which appears in the array twice and another number M in the range [1, N] which is missing from the array.

Your task is to find the missing number (M) and the repeating number (R).

**Naive approach:** Using frequency array.

Since the numbers are from 1 to N in the array arr []

* Take a frequency array of size N+1 and initalize it with 0.
* Traverse the given array and increase the value of frequency[arr[i]] by one.
* Then again traverse the frequency array starting from index 1 to N.

If you find any index value greater than 1 that is repeating element A. If you find any index value = 0 then that is the missing element B.

Time Complexity: O(N)

Space Complexity: O(N) As we are making new frequency array

**Best approach:**

Here the cyclic method does not work in finding the duplicate number because, it is possible that

N = 6 and at sametime 6 also present in the array, if we had to look for nums [6] would throw array out of bound

Also, we can not find the duplicate number by using the xor method either because here unlike the case of N+1 elements where each number from 1 to N appear atleast once, we can not guarantee that here in case of N elements from [1, N-1] only exist as it is possible that N itself can exist.

**Approach:**

Let us take N=3 and arr = [1, 3 ,3], if we take xor between sets [1,2,3] and [1,3,3], it will come out to be 2^3, which is nothing but xor of M ^ R.

Let us say findXOR = M ^ R.

All the set bits in findXOR will be in either the missing number or the repeating number (because only 0 ^ 1 or 1 ^ 0 will make it 1 which is set bit). Using any set bit in the findXOR, let's say the rightmost set bit, we can divide the elements of the given array in two sets.

* In the first set, we have the elements with the same bit set.
* In the second set, we have all the elements with the same bit not set.

How to find the rightmost set bit number, let us say if findXOR = 12 => 1 1 0 0, take ~findXor => 0 0 1 1

-findXor is (take 2’s complement) 0 0 1 1 + 1 => 0 1 0 0

now findXOR & (-findXor) => ((1 1 0 0) & (0 1 0 0)) => 0 1 0 0 which is the right most set bit number.

Initialize two variables, ‘bitSet’ and ‘bitNotSet’ with 0.

Traverse the given array and if the element belongs to the first set, take its XOR with bitSet else take its XOR with bitNotSet.

Traverse from 1 to n, and if the number belongs to the first set, take its XOR with bitSet else take its XOR with bitNotSet.

Now, we have our missing and repeating numbers stored in bitSet and bitNotSet (since there will be double instances of every number in each bucket except them), but we do not know which one is missing and which one is repeating.

To find which one is repeating, we will again traverse the given array and if ‘bitSet’ is present in the array, it means ‘bitSet’ is our repeating number and ‘bitNotSet’ is our missing number. We will store ‘bitSet’ in the variable ‘R’ and ‘bitNotSet’ in the variable ‘M’. If ‘bitNotSet’ is present in the array, it means ‘bitNotSet’ is our repeating number and ‘bitSet’ is our missing number. We will store ‘bitSet’ in the variable ‘M’ and ‘bitNotSet’ in the variable ‘R’.

Example:

Take [1 1 2 4]

findXOR =[1 1 2 4] xor [1 ,2 ,3 ,4 ] => 1 ^ 3 => 0 0 1 0 whose first set bit number is also 0 0 1 0.

First let us iterate through array and divide it.

bitSet = 2, bitNotSet = 1^1^4

Next, let us iterate through [1,2,3,4] and divide it.

bitSet = (2) ^ (2 ^ 3 ) bitNotSet = (1^1^4) ^ (1^4)

bitSet =3 , bitNotSet = 1 which are our answers.

### Count inversions in an array

**Problem Statement:** Given an array of N integers, count the inversion of the array (using merge-sort).

What is an inversion of an array? Definition: for all i & j < size of array, if i < j then you have to find pair (A[i],A[j]) such that A[j] < A[i].

**Brute Force :**

Use the nested for loop and maintain the variables i,j.

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

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

if( arr[i] > arr[j] )

Time Complexity: O(N^2)

Space Complexity: O (1)

**Best Approach:** Using merge sort

In the merge sort, we divide the array down into halves and build again up into the one single array.

When we go deep down into one single element, we compare two elements (calculate inversion counts)

and put it in a new array in sorted order.

**Note:** Here we should not worry about change in the order of the array.

Another point is when we are comparing one element in left sorted array with another element in right sorted array all the indices in left array is less than right array indices i.e always ( left\_index < right\_index )

so the condition that we need to check is ( left\_array[left\_index] > right\_array[right\_index] )

left\_array right\_array

[10 15 18 ] [9 20 25]

let us say left\_index =0 , right\_index =0 and 10 > 9 , here this implicitly mean that remaining all the elements in left\_array also greater than 9.

so **, inversion\_count += left\_array\_size - left\_index ;**

**Time Complexity:** O (n logn) for merge sort algo

**Space Complexity:** O(n) for the auxillary space required for two arrays.

## Arrays III

### Search in a sorted 2D matrix

**Problem Statement:** Given an m\*n 2D matrix and an integer, write a program to find if the given integer exists in the matrix.

Given matrix has the following properties:

Integers in each row are sorted from left to right. The first integer of each row is greater than the last integer of the previous row.

**Naive Approach:**

We can traverse through every element that is present in the matrix and return true if we found any element in the matrix is equal to the target integer. If the traversal is finished, we can directly return false as we did not find any element in the matrix to be equal to the target integer.

Time complexity: O(m\*n)

Space complexity: O (1)

**Best Approach:** Using binary search

Here we need to consider the 2D array as the single 1D array

Initially have a low index as the first index of the considered 1D matrix(i.e: 0)

and high index as the last index of the considered 1D matrix(i.e: (R\*C)-1).

Get the middle index as (low+high)/2.We can get the element at middle index

using matrix[middle/C][middle%C].

Example :

[0,0] 0

[0,1] 1

[0,2] 2

[0,3] 3

[1,0] 4

[1,1] 5

[1,2] 6

[1,3] 7

[2,0] 8

[2,1] 9

[2,2] 10

[2,3] 11

**Time complexity:** O(log(m\*n))

**Space complexity:** O(1)

### Implement Pow(x,n) | X raised to the power N

**Problem Statement:** Given a double x and integer n, calculate x raised to power n. Basically Implement pow(x, n).

**Brute force :** Looping from 1 to n and keeping a ans(double) variable. Now every time your loop runs, multiply x with ans. At last, we will return the ans.

Now if n is negative we must check if n is negative, if it is negative divide 1 by the end.

Time Complexity: O(N)

Space Complexity: O(1)

**Binary exponentiation :**

While calculating (n^k), binary exponentiation relies on whether k is even or odd.

If k is even (n^k) can be written as (n^2)^k/2. As we can see that computation steps were reduced from k to k/2 in just one step.If k is odd (n^k) can be written as n.(n)^k-1, so now (k-1) is even.

Time Complexity: O(logn).

Space Complexity: O(1)

### Find the Majority Element that occurs more than N/2 times

**Problem Statement:** Given an array of N integers, write a program to return an element that occurs more than N/2 times in the given array. You may consider that such an element always exists in the array.

**Brute force**: Find the frequency of each element in the array check if the value is greater than the floor of N/2.

**Time Complexity**: O(N)-> Frequency array or O(N log N) -> HashMap

**Space Complexity:** O(N)

**Best Approach**: Moore’s Voting Algorithm

The question clearly states that the nums array has a majority element. Since it has a majority element, we can say definitely the count is more than N/2.

Majority element count = N/2 + x;

Minority/Other elements = N-( N/2 + x) = N/2 – x;

Where x is the number of times it occurs after reaching the minimum value N/2.

We can use this fact to keep track a Count , Element variable

* If Count is 0 then initialize the current traversing integer of array as Element
* If the traversing integer of array and Element are same increase Count by 1
* If they are different decrease Count by 1

The integer present in Element is the result we are expecting , as we have for the count after it gets cancelled with the minority elements , remaining are all majority only.

### Majority Elements(>N/3 times)

**Problem Statement:** Given an array of N integers. Find the elements that appear more than N/3 times in the array. If no such element exists, return an empty vector.

**Brute-Force :**

Simply count the no. of appearance for each element using nested loops and whenever you find the count of an element greater than N/3 times, that element will be our answer.

Time Complexity: O(n)

Space Complexity: O(n)

**Best Approach** : Extended Moore’s Voting Algorithm

We are sure that there will be a max of 2 elements(at most) which occurs > N/3 times,extend the algorithm for 2 values.

Time Complexity: O(n)

Space Complexity: O(1)

### Grid Unique Paths

**Problem Statement**: Given a matrix m X n, count paths from left-top to the right bottom of a matrix with the constraints that from each cell you can either only move to the rightward direction or the downward direction.

**Brute force :**

The problem states that we can either move rightward or downward direction. So, we recursively try out all the possible combinations.

Time complexity : The time comp[lexity of this recursive solution is exponential. O(2^(m+n))

Space complexity : it is determined by the maximum depth of the recursive call stack. Since each call to uniquePaths(m, n) results in two additional calls, the maximum depth of the stack is equal to m+n-2. Therefore, the space complexity of this solution is O(m+n)

**Better Approach:** Use DP

Time complexity: O(n\*m)

Space complexity: O(n\*m)

**Best Approach**: Combinatorics Solution

Each time we are taking an exactly m+n-2 number of steps to reach the end.

Out of these m+n-2 steps, n-1 rightward direction steps and m-1 downward direction steps, so if we calculate here the combinations ( ie: m+n-2Cn-1 or m+n-2Cm-1) we’ll get the total number of paths.

Time Complexity: The time complexity of this solution will be O(n-1) or O(m-1) depending on the formula we are using.

Space Complexity: As we are not using any extra space the space complexity of the solution will be O(1).

(Thought process: Assume the straight line of m+n-2 bricks, the is can be bent in any such ways so that each bent structure should have n-1 right units and m-1 down units).

### Count Reverse Pairs

**Problem Statement:** Given an array of numbers, you need to return the count of reverse pairs. Reverse Pairs are those pairs where i<j and arr[i]>2\*arr[j].

**Brute Force:** As we can see from the given question that i < j, so we can just use 2 nested loops and check for the given condition which is arr [i] > 2\* arr[j]

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

**Space Complexity**: O (1)

**Best Approach**: Use merge sort technique

We will need to find the count before the merging (unlike while merging as suited for count inversions problem earlier)

// calculating the inversion count O(n) operation

**for(**left\_index **=**0 **;** left\_index **<** left\_array\_size **;** left\_index **++** **){**

**while((**right\_index **<** right\_array\_size**)** **and** **(**left\_array**[**left\_index**]** **>** 2LL **\*** right\_array**[**right\_index**])** **)**

right\_index **++** **;**

inversion\_count **+=** **(**right\_index**)** **;**

**}**

The inner loop increments right\_index as long as left\_array[left\_index] > 2LL \* right\_array[right\_index], meaning as long as the current element from the left subarray is greater than twice the current element from the right subarray. When this condition is no longer true, it means that right\_index represents the count of reverse pairs for the current element in the left subarray, which is added to the overall count.

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

Reason : O(N) – Merge operation , O(N) – counting operation ( at each iteration of i , j doesn’t start from 0 . Both of them move linearly )

Space Complexity : O(N)

Reason : O(N) – Temporary vector

## Arrays IV

### Two Sum

**Problem Statement:** Given an array of integers nums [] and an integer target, return indices of the two numbers such that their sum is equal to the target.

Note: Assume that there is exactly one solution, and you are not allowed to use the same element twice. Example: If target is equal to 6 and num [1] = 3, then nums [1] + nums [1] = target is not a solution.

**Brute Force:** Use nested for loop with variables i, j.

Time Complexity: O(N2)

Space Complexity: O (1)

**Better Approach: Two-Pointer Approach**

Given array, nums = [2,1,3,4], target = 4

* First, we sort the array. So nums after sorting is [1,2,3,4]
* We take two pointers, i and j. i points to index 0 and j points to index 3.
* Now we check if nums[i] + nums[j] == target. In this case, they don’t sum up, and nums[i] + nums[j] > target, so we will reduce j by 1.
* Now, i = 0, j=2
* Here, nums[i] + nums[j] == 1 + 3 == 4, which is the required target, so we store both the elements and break the loop.
* Now, we run another loop to find the indices of the elements in the actual array, i.e [2,1,3,4]
* Then, return the actual indices, [1,2].

Time Complexity: O(NlogN)

Space Complexity: O(N)

**Best Approach:** Hashing (Most efficient)

We can solve this problem efficiently by using hashing. We’ll use a hash-map to see if there exists a value target – x for each value x. If target – x is found in the map, can return current element x’s index and (target-x)’s index

Time Complexity: O(N)

Space Complexity: O(N)