Set Matrix Zero

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.
Examples:
Examples 1:

Input: matrix=[[1,1,1],[1,0,1],[1,1,1]]

Output: [[1,0,1],[0,0,0],[1,0,1]]

Explanation: Since matrix[2][2]=0.Therfore the 2nd column and 2nd row wil be set to 0.

Input: matrix=[[0,1,2,0],[3,4,5,2],[1,3,1,5]]

Output:[[0,0,0,0],[0,4,5,0],[0,3,1,0]]

Explanation:Since matrix[0][0]=0 and matrix[0][3]=0. Therefore 1st row, 1st column and 4th column will be set to 0

Solution
Disclaimer*: *Don't jump directly to the solution, try it out yourself first.
Solution 1:
Approach: Using brute force
Assuming all the elements in the matrix are non-negative. 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, which will be the answer.
Code:

```
#include<bits/stdc++.h>
using namespace std;
void setZeroes(vector < vector < int >> & matrix) {
 int rows = matrix.size(), cols = matrix[0].size();
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   if (matrix[i][j] == 0) {
    int ind = i - 1;
    while (ind >= 0) {
     if (matrix[ind][j] != 0) {
      matrix[ind][j] = -1;
     }
     ind--;
    }
    ind = i + 1;
    while (ind < rows) {
     if (matrix[ind][j] != 0) {
      matrix[ind][j] = -1;
     }
     ind++;
    }
    ind = j - 1;
```

```
      while (ind >= 0) {
        if (matrix[i][ind] != 0) {
          matrix[i][ind] = -1;

        }
        ind--;
      }
      ind = j + 1;
      while (ind < cols) {
        if (matrix[i][ind] != 0) {
          matrix[i][ind] = -1;

        }
        ind++;
      }
    }
  }
 }
 for (int i = 0; i < rows; i++) {
   for (int j = 0; j < cols; j++) {
     if (matrix[i][j] <= 0) {
       matrix[i][j] = 0;
     }
   }
 }

}

int main() {
  vector < vector < int >> arr;
  arr = {{0, 1, 2, 0}, {3, 4, 5, 2}, {1, 3, 1, 5}};
  setZeroes(arr);
  cout << "The Final Matrix is " << endl;
  for (int i = 0; i < arr.size(); i++) {
    for (int j = 0; j < arr[0].size(); j++) {
      cout << arr[i][j] << " ";
    }
    cout << "\n";
  }
}
```

Output:
The Final Matrix is
0 0 0 0
0 4 5 0
0 3 1 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)
Solution 2:Better approach
Intuition: Instead of traversing through each row and column, we can use dummy arrays to check if the particular row or column has an element 0 or not, which will improve the time complexity.

Approach: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.

Code:

```
#include<bits/stdc++.h>
using namespace std;
void setZeroes(vector < vector < int >> & matrix) {
 int rows = matrix.size(), cols = matrix[0].size();
 vector < int > dummy1(rows,-1), dummy2(cols,-1);
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   if (matrix[i][j] == 0) {
    dummy1[i] = 0;
    dummy2[j] = 0;
   }
  }

 }
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   if (dummy1[i] == 0 || dummy2[j]==0) {
    matrix[i][j] = 0;
   }
  }
 }

}

int main() {
 vector < vector < int >> arr;
 arr = {{0, 1, 2, 0}, {3, 4, 5, 2}, {1, 3, 1, 5}};
 setZeroes(arr);
 cout<<"The Final Matrix is "<<endl;
 for (int i = 0; i < arr.size(); i++) {
  for (int j = 0; j < arr[0].size(); j++) {
   cout << arr[i][j] << " ";
  }
  cout << "\n";
 }
}
```

Output:

```
The Final Matrix is
0 0 0 0
0 4 5 0
0 3 1 0
```

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

Space Complexity: O(N)

Solution 3:Optimizing the better approach.

Intuition: Instead of taking two dummy arrays we can use the first row and column of the matrix for the same work. This will help to reduce the space complexity of the problem. 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.
Approach: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.
Code:

```
#include<bits/stdc++.h>
using namespace std;
void setZeroes(vector < vector < int >> & matrix) {
  int col0 = 1, rows = matrix.size(), cols = matrix[0].size();
  for (int i = 0; i < rows; i++) {
    //checking if 0 is present in the 0th column or not
    if (matrix[i][0] == 0) col0 = 0;
    for (int j = 1; j < cols; j++) {
      if (matrix[i][j] == 0) {
        matrix[i][0] = 0;
        matrix[0][j] = 0;
      }
    }
  }
  //traversing in the reverse direction and
  //checking if the row or col has 0 or not
  //and setting values of matrix accordingly.
  for (int i = rows - 1; i >= 0; i--) {
    for (int j = cols - 1; j >= 1; j--) {
      if (matrix[i][0] == 0 || matrix[0][j] == 0) {
        matrix[i][j] = 0;
      }
    }
    if (col0 == 0) {
      matrix[i][0] = 0;
    }

  }

}

int main() {
  vector < vector < int >> arr;
  arr = {{0, 1, 2, 0}, {3, 4, 5, 2}, {1, 3, 1, 5}};
  setZeroes(arr);
  cout<<"The Final Matrix is "<<endl;
  for (int i = 0; i < arr.size(); i++) {
    for (int j = 0; j < arr[0].size(); j++) {
      cout << arr[i][j] << " ";
    }
    cout << "\n";
  }
}
```

Output:
The Final Matrix is
0 0 0 0
0 4 5 0
0 3 1 0
Time Complexity: O(2*(N*M)), as we are traversing two times in a matrix,
Space Complexity: O(1).

Program to generate Pascal's Triangle

Problem Statement: Given an integer N, return the first N rows of Pascal's triangle.
In Pascal's triangle, each number is the sum of the two numbers directly above it as shown in the figure below:
Example 1:
Input Format: N = 5

Result:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Explanation: There are 5 rows in the output matrix. Each row corresponds to each one of the rows in the image shown above.
Example 2:
Input Format: N = 1

Result:
1

Solution

Disclaimer*: *Don't jump directly to the solution, try it out yourself first.

Solution 1:
Intuition: When you see the image above, you get a pretty good idea of what you are supposed to do here. Think about the image as a matrix now where each line is basically a row in the matrix. So, first things first, if you are at the edge of the matrix, the value is 1, that's for sure. Now, what about the inner elements? Well, any inner element is obtained by doing the sum of the 2 values present in the row just above it, i.e., if the element is at index (i, j), then matrix[i][j] can be obtained by doing matrix[i – 1][j – 1] + matrix[i – 1][j].

Approach: To solve the problem, we need to first create an array of size N or numRows (input value). This array is used to store each of the rows expected in the output, so, for example, array[1] = [1,1]. In this array, the number of columns (say, numCols) is equal to the number of the i-th row + 1 (Since, 0-indexed), i.e., for 0-th row, numCols = 1. So, the number of columns is different for each row.
Next, we need to run a loop from i = 0 to numRows – 1 (inclusive) in order to store each row in our array. For each of iteration of this loop, we follow the below steps:

- **Create an array of size (i + 1) (*For some languages such as C++, you need to create a 2D array at the start of the program and resize array[i] to (i + 1)*).**
- **Set the first and last value of array[i] to 1.**
- **Run another loop from j = 1 to i – 1 (inclusive) and for each iteration put array[i][j] = array[i – 1][j – 1] + array[i – 1][j].**

After iterating numRows times, you return the array.
Dry Run: Let's do a dry run to understand it in a much better way.

Input: **numRows = 5**

- **Step – I: Initialized an array of size numRows, array = [[],[],[],[],[]]**
- **Step – II: Run a loop from i = 0 to numRows – 1**
    - **At i = 0:**
        - **We resize the first row of the array to (i + 1) = (0 + 1) = 1, so array = [[0],[],[],[],[]].**
        - **Set the first and last value of array[i] = 1, so, array[0][0] = 1 and array[0][i] = 1. Therefore, array = [[1],[],[],[],[]]**
        - **Since, i = 0, the nested for loop does not satisfy the running criteria and hence does not get executed.**
    - **At i = 1:**
        - **We resize the second row of the array to (i + 1) = (1 + 1) = 2, so array = [[1],[0,0],[],[],[]].**
        - **Set the first and last value of array[i] = 1, so, array[1][0] = 1 and array[1][i] = 1. Therefore, array = [[1],[1,1],[],[],[]]**
        - **Since, i = 1, the nested for loop does not satisfy the running criteria and hence does not get executed.**
    - **At i = 2:**
        - **We resize the third row of the array to (i + 1) = (2 + 1) = 3, so array = [[1],[1,1],[0,0,0],[],[]].**
        - **Set the first and last value of array[i] = 1, so, array[2][0] = 1 and array[2][i] = 1. Therefore, array = [[1],[1,1],[1,0,1],[],[]]**
        - **Run a loop from j = 1 to i – 1:**
            - **At j = 1:**
                - **array[i][j] = array[i – 1][j – 1] + array[i – 1][j]**

= array[1][0] + array[1][1] = 1 + 1 = 2

So, array = [[1],[1,1],[1,2,1],[],[]]

- **At i = 3 and i = 4, we follow the same approach to fill the two rows of the array and get array = [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]].**
- **Step – III: Return array.**

Code:

```
class Solution {
public:
  vector<vector<int>> generate(int numRows) {
    vector<vector<int>> r(numRows);

    for (int i = 0; i < numRows; i++) {
      r[i].resize(i + 1);
      r[i][0] = r[i][i] = 1;

      for (int j = 1; j < i; j++)
        r[i][j] = r[i - 1][j - 1] + r[i - 1][j];
    }

    return r;
  }
};
```

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 : find next lexicographically greater 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).
Example 1 :
Input format: Arr[] = {1,3,2}

Output: Arr[] = {2,1,3}

Explanation: All permutations of {1,2,3} are {{1,2,3} , {1,3,2}, {2,13} , {2,3,1} , {3,1,2} , {3,2,1}}. So, the next permutation just after {1,3,2} is {2,1,3}.
Example 2:
Input format: Arr[] = {3,2,1}

Output: Arr[] = {1,2,3}

Explanation: As we see all permutations of {1,2,3}, we find {3,2,1} at the last position. So, we have to return the topmost permutation.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Solution 1 Brute Force: Finding all possible permutations.
Approach :
Step 1: Find all possible permutations of elements present and store them.
Step 2: Search input from all possible permutations.
Step 3: Print the next permutation present right after it.

For reference of how to find all possible permutations, follow up https://www.youtube.com/watch?v=f2ic2Rsc9pU&t=32s. This video shows for distinct elements but code works for duplicates too.

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).

Solution 2 : Using C++ in-built function
C++ provides an in-built function called next_permutation() which directly returns the lexicographically next greater permutation of the input.

Code :

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

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

   next_permutation(arr,arr+3);//using in-built function of C++

   cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2];

   return 0;
}
```

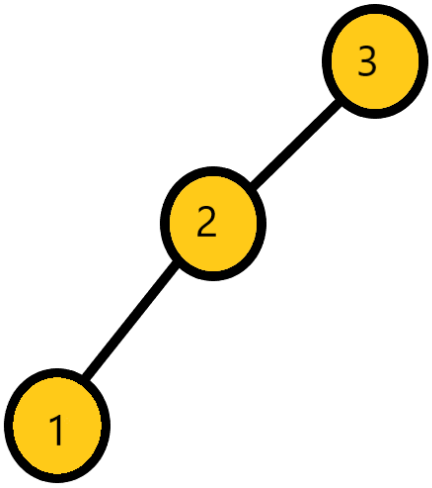
Solution 3 :

Intuition :

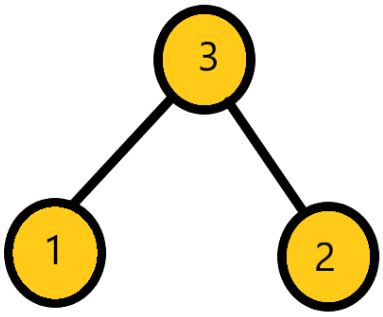
Intuition lies behind the lexicographical ordering of all possible permutations of a given array. There will always be an increasing sequence of all possible permutations when observed.

Let's check all sequences of permutations of {1,2,3}.

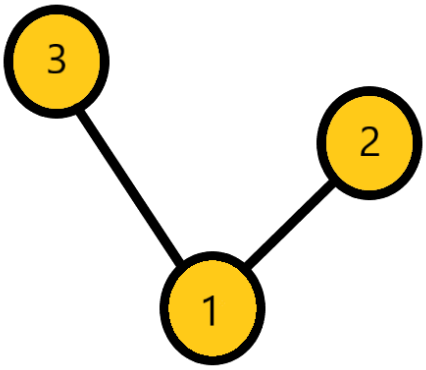
- **{1,2,3}**



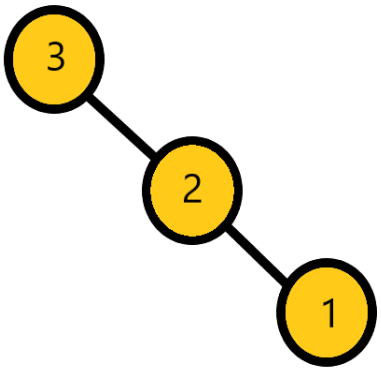
- **{1,3,2}**



- **{3,1,2}**



- **{3,2,1}**



Thus, we can see every sequence has increasing order. Hence, our approach aims to get a peak from where the increasing sequence starts. This is what we achieve from our first step of the approach.
Then, we need to get just a larger value than the point where the peak occurs. To make rank as few as possible but greater than input array, just perverse array from breakpoint achieved from the first step of the approach. We achieve these from all remaining steps of our approach.
Approach :
Step 1: Linearly traverse array from backward such that ith index value of the array is less than (i+1)th index value. Store that index in a variable.
Step 2: If the index value received from step 1 is less than 0. This means the given input array is the largest lexicographical permutation. Hence, we will reverse the input array to get the minimum or starting permutation. Linearly traverse array from backward. Find an index that has a value greater than the previously found index. Store index another variable.
Step 3: Swap values present in indices found in the above two steps.
Step 4: Reverse array from index+1 where the index is found at step 1 till the end of the array.
Code :

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int n = nums.size(), k, l;
            for (k = n - 2; k >= 0; k--) {
          if (nums[k] < nums[k + 1]) {
            break;
          }
        }
            if (k < 0) {
```

```
            reverse(nums.begin(), nums.end());
          } else {
            for (l = n - 1; l > k; l--) {
          if (nums[l] > nums[k]) {
            break;
          }
        }
            swap(nums[k], nums[l]);
            reverse(nums.begin() + k + 1, nums.end());
    }
  }
};
```

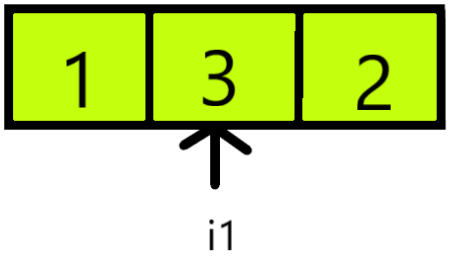
Time Complexity: For the first iteration backward, the second interaction backward and reversal at the end takes O(N) for each, where N is the number of elements in the input array. This sums up to 3*O(N) which is approximately O(N).

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

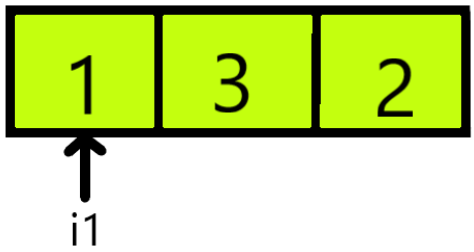
Dry Run :

We will take the input array {1,3,2}.

Step 1: First find an increasing sequence. We take i1 = 1. Starting traversing backward.

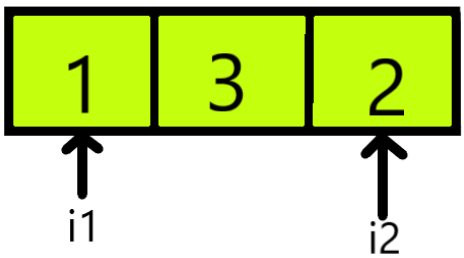


Step 2: Since 3 is not less than 2, we decrease i1 by 1.

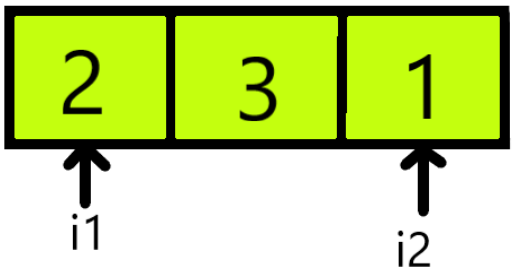


Step 3: Since 1 is less than 2, we achieved our start of the increasing sequence. Now, i1 = 0.

Step 4: i2 will be another index to find just greater than i1 indexed elements in the array. Point i2 to the last element.



Step 5: i2 indexed element is greater than i1 indexed element. So, i2 has a value of 2.
Step 6: Swapping values present in i1 and i2 indices.



Step 7: Reversing from i1+1 index to last of the array.

Thus, we achieved our final answer.

Kadane's Algorithm : Maximum Subarray Sum in an Array

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.
Examples:
Example 1:
Input: arr = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

Examples 2:
Input: arr = [1]
Output: 1
Explanation: Array has only one element and which is giving positive sum of 1.

Solution
Disclaimer*: *Don't jump directly to the solution, try it out yourself first.

Solution 1: Naive Approach
Approach: Using three for loops, we will get all possible subarrays in two loops and their sum in another loop, and then return the maximum of them.
Code:

```
#include<bits/stdc++.h>

using namespace std;
```

```
int maxSubArray(vector < int > & nums, vector < int > & subarray) {
 int max_sum = 0;
 int n = nums.size();
 if (n == 1) {
  return nums[0];
 }
 int i, j;
 for (i = 0; i <= n - 1; i++) {
  for (j = i; j <= n - 1; j++) {
   int sum = 0;
   for (int k = i; k <= j; k++)
    sum = sum + nums[k];
   if (sum > max_sum) {
    subarray.clear();
    max_sum = sum;
    subarray.push_back(i);
    subarray.push_back(j);

   }
  }
 }
 return max_sum;
}

int main() {
 vector<int> arr{-2,1,-3,4,-1,2,1,-5,4};
 vector < int > subarray;
 int lon = maxSubArray(arr, subarray);
 cout << "The longest subarray with maximum sum is " << lon << endl;
 cout << "The subarray is " << endl;
 for (int i = subarray[0]; i <= subarray[1]; i++) {
  cout << arr[i] << " ";
 }

}
```

Output:
The longest subarray with maximum sum is 6
The subarray is
4 -1 2 1
Time Complexity: O(N^3)
Space Complexity: O(1)
Solution 2: A better approach
Approach :
We can also do this problem using only two for loop, starting the function with (max_sum) variable which will have the final answer. We can get the sum of all possible subarrays in this way and then return the maximum so far.
Code:

```
#include<bits/stdc++.h>

using namespace std;
int maxSubArray(vector < int > & nums, vector < int > & subarray) {
 int max_sum = INT_MIN;
```

```
 for (int i = 0; i < nums.size(); i++) {
  int curr_sum = 0;
  for (int j = i; j < nums.size(); j++) {
   curr_sum += nums[j];
   if (curr_sum > max_sum) {
    subarray.clear();
    max_sum = curr_sum;
    subarray.push_back(i);
    subarray.push_back(j);
   }
  }
 }
 return max_sum;
}

int main() {
 vector<int> arr{-2,1,-3,4,-1,2,1,-5,4};
 vector < int > subarray;
 int lon = maxSubArray(arr, subarray);
 cout << "The longest subarray with maximum sum is " << lon << endl;
 cout << "The subarray is " << endl;
 for (int i = subarray[0]; i <= subarray[1]; i++) {
  cout << arr[i] << " ";
 }

}
```

Output:
The longest subarray with maximum sum is 6
The subarray is
4 -1 2 1
Time Complexity: O(N^2)
Space Complexity: O(1)

Solution : 3 Optimal Solution: Kadane's Algorithm
Intuition: Basically this problem can be done in linear time complexity with Kadane's algorithm along with that will also get the subarray which is giving the largest positive-sum.
Approach:
-> We will have the following variables in the beginning :
msf – max so far, meh – max end here, start – to get the starting index of ans's subarray, end – to get the ending index of ans's subarray.
-> Traverse the array starting with adding the ith element with max_end_here(meh) , now we will check that adding the element gives greater value than max_so_far(msf) , if yes then we will update our meh and also update the starting and ending index .

```
for(int i=0;i<nums.size();i++){
meh+=nums[i];
if(meh>msf){ msf=meh; start=s; end=i; }
if(meh<0){meh=0; s=i+1;}
}
```

->Now in this step, we will print the answer subarray using the start and end variables.
->Return the largest subarray sum that is:- msf.
Code:

```
#include<bits/stdc++.h>
```

```
using namespace std;
int maxSubArray(vector < int > & nums, vector < int > & subarray) {
  int msf = nums[0], meh = 0;
  int s = 0;
  for (int i = 0; i < nums.size(); i++) {
    meh += nums[i];
    if (meh > msf) {
      subarray.clear();
      msf = meh;
      subarray.push_back(s);
      subarray.push_back(i);

    }
    if (meh < 0) {
      meh = 0;
      s = i + 1;

    }
  }

  return msf;
}

int main() {
  vector<int> arr{-2,1,-3,4,-1,2,1,-5,4};
  vector < int > subarray;
  int lon = maxSubArray(arr, subarray);
  cout << "The longest subarray with maximum sum is " << lon << endl;
  cout << "The subarray is " << endl;
  for (int i = subarray[0]; i <= subarray[1]; i++) {
    cout << arr[i] << " ";
  }

}
```

Output:
The longest subarray with maximum sum is 6
The subarray is
4 -1 2 1
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)
Example 1:
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]

Input: nums = [2,0,1]
Output: [0,1,2]

Input: nums = [0]
Input: nums = [0]

Solution

***Disclaimer*: *Don't jump directly to the solution, try it out yourself first*.**

Solution 1: Sorting (even if it is not the expected solution here but it can be considered as one of the approach).
Since the array contains only 3 integers, 0, 1, and 2. Simply sorting the array would arrange the elements in increasing order.
Time Complexity: O(N*logN)
Space Complexity: O(1)

Solution 2: Keeping count of values
Intuition: 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.
Approach:

1. **Take 3 variables to maintain the count of 0, 1 and 2.**
2. **Travel the array once and increment the corresponding counting variables**

(let's consider count_0 = a, count_1 = b, count_2 = c)

3. **In 2nd traversal of array, we will now over write the first 'a' indices / positions in array with '0', the next 'b' with '1' and the remaining 'c' with '2'.**

Time Complexity: O(N) + O(N)
Space Complexity: O(1)

Solution 3: 3-Pointer approach
This problem is a variation of the *popular Dutch National flag algorithm*
Intuition: In this approach, we will be using 3 pointers named low, mid, and high. We will be using these 3 pointers to move around the values. 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.
Approach:

1. **Initialize the 3 pointers such that low and mid will point to 0th index and high pointer will point to last index**

int low = arr[0]
int mid = arr[0]
int high = arr[n – 1]

2. **Now there will 3 different operations / steps based on the value of arr[mid] and will be repeated until mid <= high.**

1. **arr[mid] == 0:**
   swap(arr[low], arr[mid])
   low++, mid++
2. **arr[mid] == 1:**
   mid++
3. **arr[mid] == 2:**
   swap(arr[mid], arr[high])
   high−;

The array formed after these steps will be a sorted array.

```
class Solution {
public:
   void sortColors(vector<int>& nums) {
      int lo = 0;
      int hi = nums.size() - 1;
      int mid = 0;

      while (mid <= hi) {
         switch (nums[mid]) {

         // If the element is 0
         case 0:
            swap(nums[lo++], nums[mid++]);
            break;

         // If the element is 1 .
         case 1:
            mid++;
            break;

         // If the element is 2
         case 2:
            swap(nums[mid], nums[hi--]);
            break;
         }
      }
   }};
```

Time Complexity: O(N) & Space Complexity: O(1)

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.
Examples:
Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note: That buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Solution:
***Disclaimer*: *Don't jump directly to the solution, try it out yourself first*.**
Solution 1: Brute Force
Intuition: We can simply use 2 loops and track every transaction and maintain a variable maxPro to contain the max value among all transactions.
Approach:

- **Use a for loop of 'i' from 0 to n.**
- **Use another for loop from 'i+1' to n.**
- **If arr[j] > arr[i] , take the difference and compare and store it in the maxPro variable.**
- **Return maxPro.**

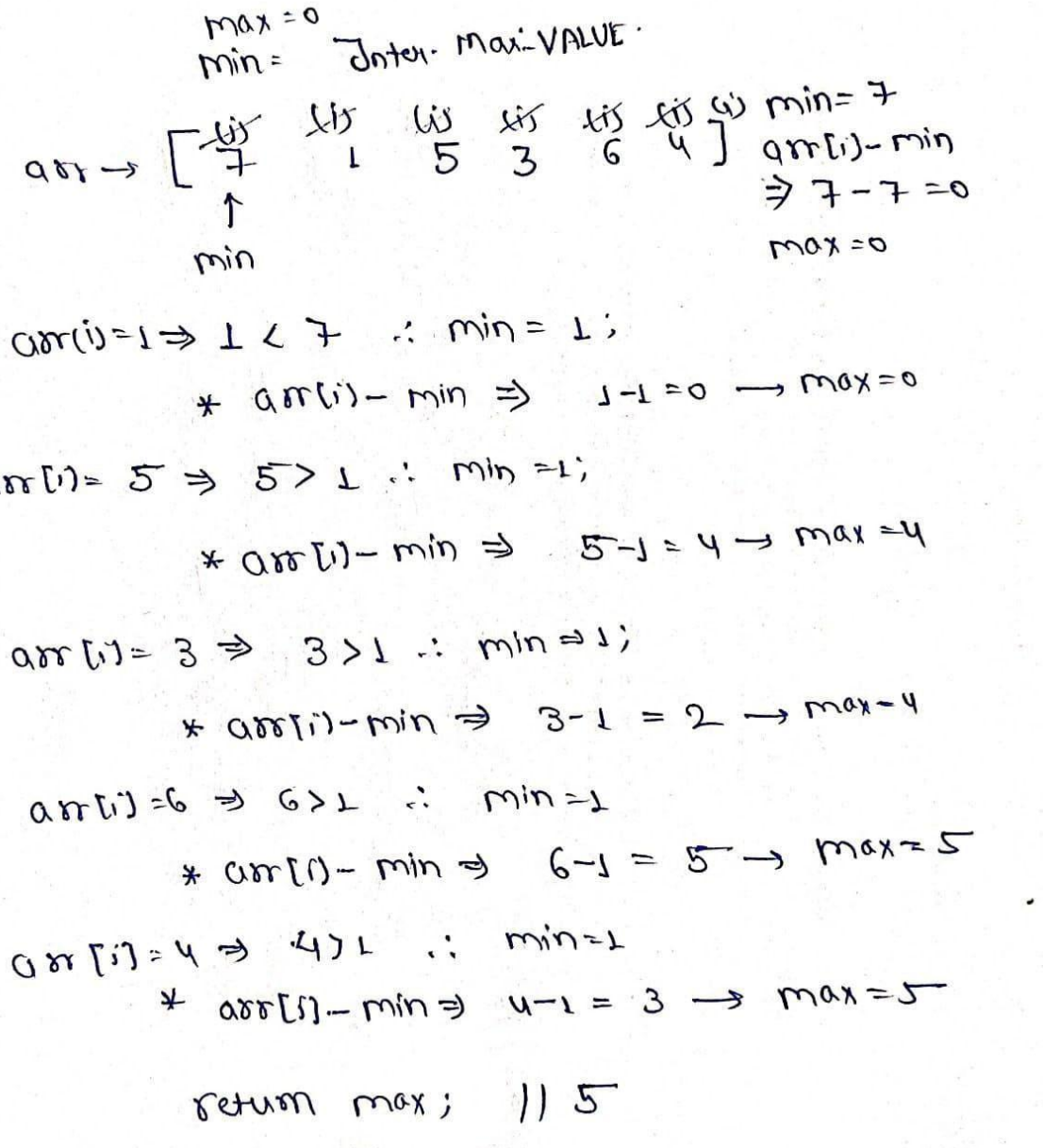
Code:

```
#include<bits/stdc++.h>

using namespace std;
int maxProfit(int arr[]) {
  int maxPro = 0;
  for (int i = 0; i < 6; i++) {
    for (int j = i + 1; j < 6; j++) {
      if (arr[j] > arr[i]) {
        maxPro = max(arr[j] - arr[i], maxPro);
      }
    }
  }
  return maxPro;
}

int main() {
  int arr[] = {7,1,5,3,6,4};
  int maxPro = maxProfit(arr);
  cout << "Max profit is: " << maxPro << endl;
}
```

Output:
Max profit is: 5
Time complexity: O(n^2)
Space Complexity: O(1)

Solution 2:Optimal 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 take the difference and maintain it in max, otherwise update the minimum.
Approach:

- **Create a variable maxPro and mark it as 0.**
- **Create a variable minPrice and mark it as max_value.**
- **Run a for loop from 0 to n.**
- **Update the minPrice at if it greater than current element of the array**
- **Take the difference of the minPrice with the current element of the array and compare and maintain it in maxPro.**
- **Return the maxPro.**



Code:

```
#include<bits/stdc++.h>

using namespace std;
int maxProfit(int arr[]) {
  int maxPro = 0;
```

```
 int minPrice = INT_MAX;
 for (int i = 0; i < 6; i++) {
  minPrice = min(minPrice, arr[i]);
  maxPro = max(maxPro, arr[i] - minPrice);
 }
 return maxPro;
}
int main() {
 int arr[] = {7,1,5,3,6,4};
 int maxPro = maxProfit(arr);
 cout << "Max profit is: " << maxPro << endl;
}
```

Output: Max profit is: 5
Time complexity: O(n)
Space Complexity: O(1)

Rotate Image by 90 degree

Problem Statement: Given a matrix, your task is to rotate the matrix by 90 degrees.
Examples:
Example 1:

Input: [[1,2,3],[4,5,6],[7,8,9]]

Output: [[7,4,1],[8,5,2],[9,6,3]]

Explanation: Rotate the matrix simply by 90 degree clockwise and return the matrix.

Example 2:

Input: [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

Output:[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

Explanation: Rotate the matrix simply by 90 degree clockwise and return the matrix

Solution

Disclaimer*: *Don't jump directly to the solution, try it out yourself first.
Solution 1:Brute force
Approach: Take another dummy matrix of n*n, and then take the first row of the matrix and put it in the last column of the dummy matrix, take the second row of the matrix, and put it in the second last column of the matrix and so.
Code:

```
#include<bits/stdc++.h>
using namespace std;
vector < vector < int >> rotate(vector < vector < int >> & matrix) {
 int n = matrix.size();
 vector < vector < int >> rotated(n, vector < int > (n, 0));
 for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
   rotated[j][n - i - 1] = matrix[i][j];
  }
 }
```

```
  return rotated;
}

int main() {
  vector < vector < int >> arr;
  arr =  {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  vector < vector < int >> rotated = rotate(arr);
  cout << "Rotated Image" << endl;
  for (int i = 0; i < rotated.size(); i++) {
    for (int j = 0; j < rotated[0].size(); j++) {
      cout << rotated[i][j] << " ";
    }
    cout << "\n";
  }

}
```

Output:

```
Rotated Image
7 4 1
8 5 2
9 6 3
```

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.
Solution 2: Optimized 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).
Approach:
Step1: Transpose of the matrix. (transposing means changing columns to rows and rows to columns)
Step2: Reverse each row of the matrix.
Code:

- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
void rotate(vector < vector < int >> & matrix) {
  int n = matrix.size();
  //transposing the matrix
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      swap(matrix[i][j], matrix[j][i]);
    }
  }
  //reversing each row of the matrix
  for (int i = 0; i < n; i++) {
    reverse(matrix[i].begin(), matrix[i].end());
  }
}
```

```
int main() {
  vector < vector < int >> arr;
  arr =  {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  rotate(arr);
  cout << "Rotated Image" << endl;
  for (int i = 0; i < arr.size(); i++) {
    for (int j = 0; j < arr[0].size(); j++) {
      cout << arr[i][j] << " ";
    }
    cout << "\n";
  }

}
```

Output:
Rotated Image
7 4 1
8 5 2
9 6 3
Time Complexity: O(N*N) + O(N*N).One O(N*N) for transposing the matrix and the other for reversing the matrix.
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.
Examples
Example 1:

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] are overlapping we can merge them to form [1,6] intervals.

Example 2:

Input: [[1,4],[4,5]]

Output: [[1,5]]

Explanation: Since intervals [1,3] and [2,6] are overlapping we can merge them to form [1,6] intervals.

Solution

Disclaimer*: *Don't jump directly to the solution, try it out yourself first.
Solution 1: Brute force
Approach: First check whether the array is sorted or not.If not sort the array. Now linearly iterate over the array and then check for all of its next intervals whether they are overlapping with the interval at the current index. Take a new data structure and insert the overlapped interval. If while iterating if the interval lies in the interval present in the data structure simply continue and move to the next interval.
Code:

- C++ Code
- Java Coding

```
#include<bits/stdc++.h>

using namespace std;
vector < pair < int, int >> merge(vector < pair < int, int >> & arr) {

 int n = arr.size();
 sort(arr.begin(), arr.end());
 vector < pair < int, int >> ans;

 for (int i = 0; i < n; i++) {
  int start = arr[i].first, end = arr[i].second;

  //since the intervals already lies
  //in the data structure present we continue
  if (!ans.empty()) {
   if (start <= ans.back().second) {
    continue;
   }
  }

  for (int j = i + 1; j < n; j++) {
   if (arr[j].first <= end) {
    end = arr[j].second;
   }
  }

  ans.push_back({
   start,
   end
  });
 }

 return ans;
}

int main() {
 vector < pair < int, int >> arr;
 arr = {{1,3},{2,4},{2,6},{8,9},{8,10},{9,11},{15,18},{16,17}};
 vector < pair < int, int >> ans = merge(arr);

 cout << "Merged Overlapping Intervals are " << endl;

 for (auto it: ans) {
  cout << it.first << " " << it.second << "\n";
 }
}
```

Output:

Merged Overlapping Intervals are
1 6
8 11
15 17
Time Complexity: O(NlogN)+O(N*N). O(NlogN) for sorting the array, and O(N*N) because we are checking to the right for each index which is a nested loop.
Space Complexity: O(N), as we are using a separate data structure.
Solution 2: Optimal approach
Approach: Linearly iterate over the array if the data structure is empty insert the interval in the data structure. If the last element in the data structure overlaps with the current interval we merge the intervals by updating the last element in the data structure, and if the current interval does not overlap with the last element in the data structure simply insert it into the data structure.
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.
Code:

- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
vector < vector < int >> merge(vector < vector < int >> & intervals) {

 sort(intervals.begin(), intervals.end());
 vector < vector < int >> merged;

 for (int i = 0; i < intervals.size(); i++) {
  if (merged.empty() || merged.back()[1] < intervals[i][0]) {
   vector < int > v = {
    intervals[i][0],
    intervals[i][1]
   };

   merged.push_back(v);
  } else {
   merged.back()[1] = max(merged.back()[1], intervals[i][1]);
  }
 }

 return merged;
}

int main() {
 vector < vector < int >> arr;
 arr = {{1, 3}, {2, 4}, {2, 6}, {8, 9}, {8, 10}, {9, 11}, {15, 18}, {16, 17}};
 vector < vector < int >> ans = merge(arr);

 cout << "Merged Overlapping Intervals are " << endl;
```

```
  for (auto it: ans) {
    cout << it[0] << " " << it[1] << "\n";
  }
}
```

Output:
Merged Overlapping Intervals are
1 6
8 11
15 18
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.

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.
Examples:
Example 1:

Input:
n = 4, arr1[] = [1 4 8 10]
m = 5, arr2[] = [2 3 9]

Output:
arr1[] = [1 2 3 4]
arr2[] = [8 9 10]

Explanation:
After merging the two
non-decreasing arrays, we get,
1,2,3,4,8,9,10.

Example2:

Input:
n = 4, arr1[] = [1 3 5 7]
m = 5, arr2[] = [0 2 6 8 9]

Output:
arr1[] = [0 1 2 3]
arr2[] = [5 6 7 8 9]

Explanation:
After merging the two
non-decreasing arrays, we get,
0 1 2 3 5 6 7 8 9.

Solution:

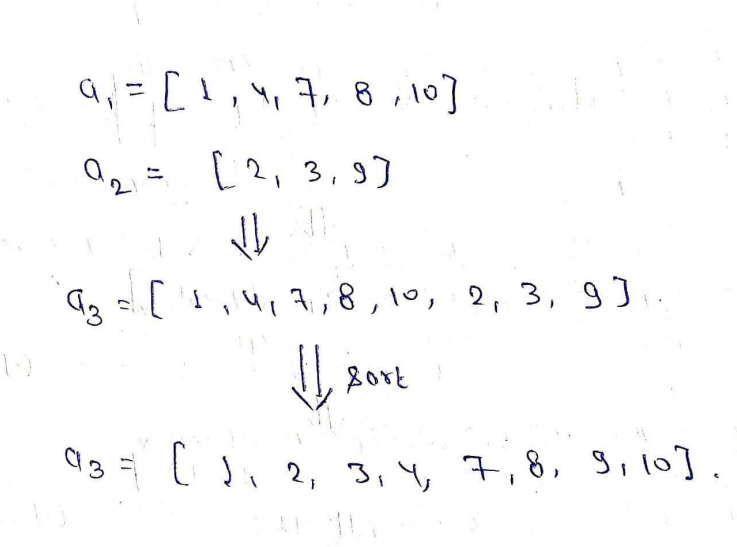
Disclaimer*: *Don't jump directly to the solution, try it out yourself first.

Solution1: Brute Force-

Intuition: 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, but all the elements in arr1 and arr2.

Approach:

- **Make an arr3 of size n+m.**
- **Put elements arr1 and arr2 in arr3.**
- **Sort the arr3.**
- **Now first fill the arr1 and then fill remaining elements in arr2.**



Code:

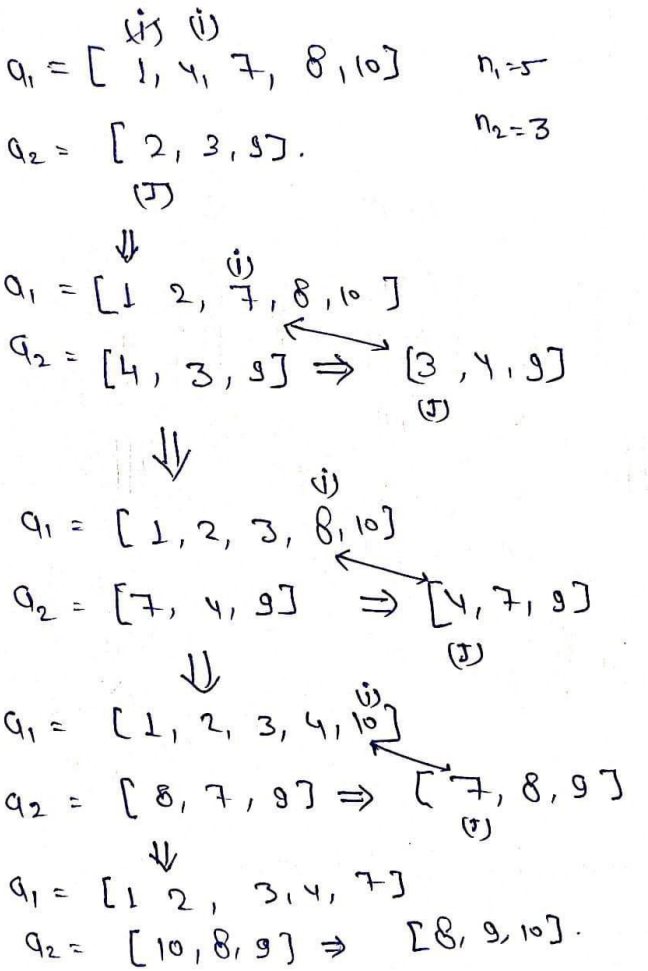
- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>
using namespace std;
void merge(int arr1[], int arr2[], int n, int m) {
   int arr3[n+m];
   int k = 0;
   for (int i = 0; i < n; i++) {
     arr3[k++] = arr1[i];
   }
   for (int i = 0; i < m; i++) {
     arr3[k++] = arr2[i];
   }
   sort(arr3,arr3+m+n);
   k = 0;
   for (int i = 0; i < n; i++) {
     arr1[i] = arr3[k++];
   }
   for (int i = 0; i < m; i++) {
     arr2[i] = arr3[k++];
   }
```

```
 }
int main() {
   int arr1[] = {1,4,7,8,10};
            int arr2[] = {2,3,9};
   cout<<"Before merge:"<<endl;
   for (int i = 0; i < 5; i++) {
     cout<<arr1[i]<<" ";
   }
   cout<<endl;
   for (int i = 0; i < 3; i++) {
     cout<<arr2[i]<<" ";
   }
   cout<<endl;
   merge(arr1, arr2, 5, 3);
   cout<<"After merge:"<<endl;
   for (int i = 0; i <5; i++) {
     cout<<arr1[i]<<" ";
   }
   cout<<endl;
   for (int i = 0; i < 3; i++) {
     cout<<arr2[i]<<" ";
   }

 }
```

Output:
Before merge:
1 4 7 8 10
2 3 9
After merge:
1 2 3 4 7
8 9 10
Time complexity: O(n*log(n))+O(n)+O(n)
Space Complexity: O(n)
Solution 2: Without using space-
Intuition: We can think of Iterating in arr1 and whenever we encounter an element that is greater than the first element of arr2, just swap it. Now rearrange the arr2 in a sorted manner, after completion of the loop we will get elements of both the arrays in non-decreasing order.
Approach:

- **Use a for loop in arr1.**
- **Whenever we get any element in arr1 which is greater than the first element of arr2,swap it.**
- **Rearrange arr2.**
- **Repeat the process.**



Code:

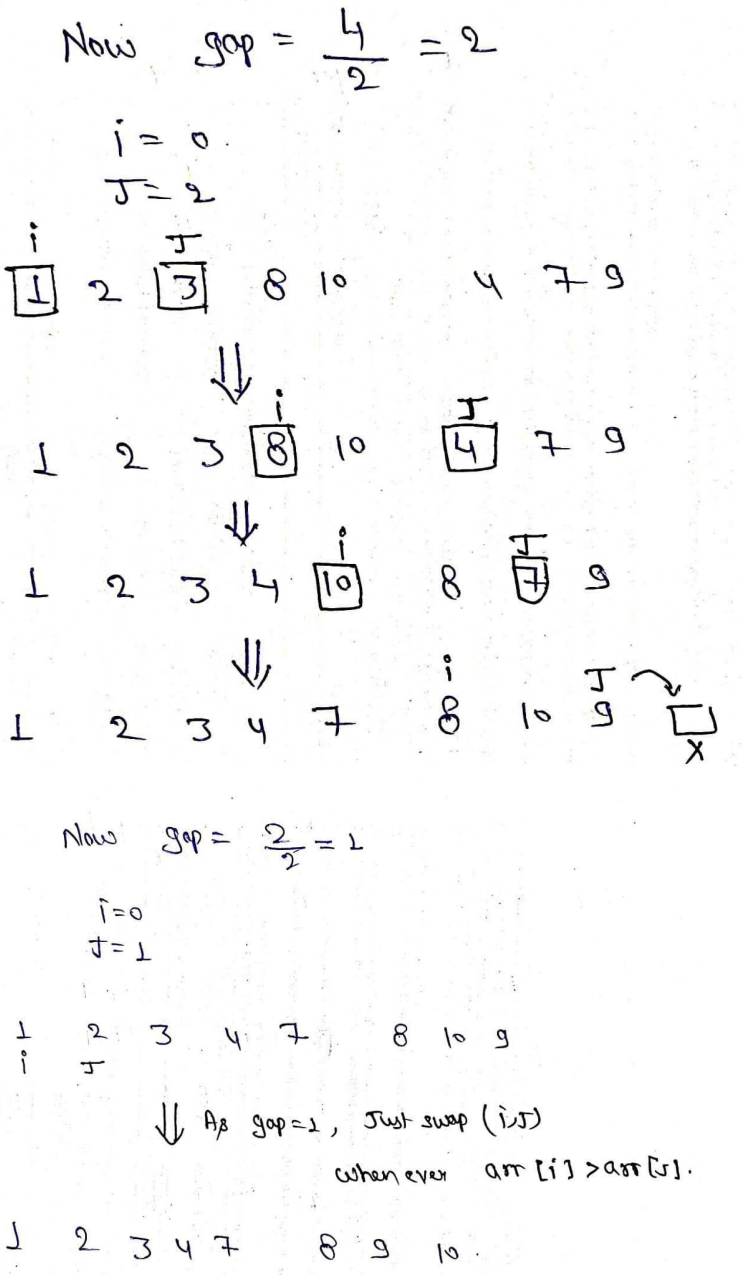
- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
void merge(int arr1[], int arr2[], int n, int m) {
  // code here
  int i, k;
  for (i = 0; i < n; i++) {
    // take first element from arr1
    // compare it with first element of second array
    // if condition match, then swap
    if (arr1[i] > arr2[0]) {
      int temp = arr1[i];
      arr1[i] = arr2[0];
      arr2[0] = temp;
    }
```

```
    // then sort the second array
    // put the element in its correct position
    // so that next cycle can swap elements correctly
    int first = arr2[0];
    // insertion sort is used here
    for (k = 1; k < m && arr2[k] < first; k++) {
      arr2[k - 1] = arr2[k];
    }
    arr2[k - 1] = first;
  }
}
int main() {
  int arr1[] = {1,4,7,8,10};
  int arr2[] = {2,3,9};
  cout << "Before merge:" << endl;
  for (int i = 0; i < 5; i++) {
    cout << arr1[i] << " ";
  }
  cout << endl;
  for (int i = 0; i < 3; i++) {
    cout << arr2[i] << " ";
  }
  cout << endl;
  merge(arr1, arr2, 5, 3);
  cout << "After merge:" << endl;
  for (int i = 0; i < 5; i++) {
    cout << arr1[i] << " ";
  }
  cout << endl;
  for (int i = 0; i < 3; i++) {
    cout << arr2[i] << " ";
  }

}
```

Output:
Before merge:
1 4 7 8 10
2 3 9
After merge:
1 2 3 4 7
8 9 10
Time complexity: O(n*m)
Space Complexity: O(1)
Solution 3: Gap method-
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.**



Code:

- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>
using namespace std;
void merge(int ar1[], int ar2[], int n, int m) {
```

```
  // code here
  int gap = ceil((float)(n + m) / 2);
  while (gap > 0) {
    int i = 0;
    int j = gap;
    while (j < (n + m)) {
      if (j < n && ar1[i] > ar1[j]) {
        swap(ar1[i], ar1[j]);
      } else if (j >= n && i < n && ar1[i] > ar2[j - n]) {
        swap(ar1[i], ar2[j - n]);
      } else if (j >= n && i >= n && ar2[i - n] > ar2[j - n]) {
        swap(ar2[i - n], ar2[j - n]);
      }
      j++;
      i++;
    }
    if (gap == 1) {
      gap = 0;
    } else {
      gap = ceil((float) gap / 2);
    }
  }
}
int main() {
  int arr1[] = {1,4,7,8,10};
  int arr2[] = {2,3,9};
  cout << "Before merge:" << endl;
  for (int i = 0; i < 5; i++) {
    cout << arr1[i] << " ";
  }
  cout << endl;
  for (int i = 0; i < 3; i++) {
    cout << arr2[i] << " ";
  }
  cout << endl;
  merge(arr1, arr2, 5, 3);
  cout << "After merge:" << endl;
  for (int i = 0; i < 5; i++) {
    cout << arr1[i] << " ";
  }
  cout << endl;
  for (int i = 0; i < 3; i++) {
    cout << arr2[i] << " ";
  }

}
Output:
Before merge:
1 4 7 8 10
2 3 9
After merge:
1 2 3 4 7
```

8 9 10
Time complexity: O(logn)
Space Complexity: O(1)

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.
Examples:
Example 1:

Input: arr=[1,3,4,2,2]

Output: 2

Explanation: Since 2 is the duplicate number the answer will be 2.

Example 2:

Input: [3,1,3,4,2]

Output:3

Explanation: Since 3 is the duplicate number the answer will be 3.

Solution

***Disclaimer*: *Don't jump directly to the solution, try it out yourself first*.**
Solution 1:Using sorting
Approach: 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.
Code:

- **C++Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
int findDuplicate(vector < int > & arr) {
  int n = arr.size();
  sort(arr.begin(), arr.end());
  for (int i = 0; i < n - 1; i++) {
    if (arr[i] == arr[i + 1]) {
      return arr[i];
    }
  }
}
int main() {
  vector < int > arr;
```

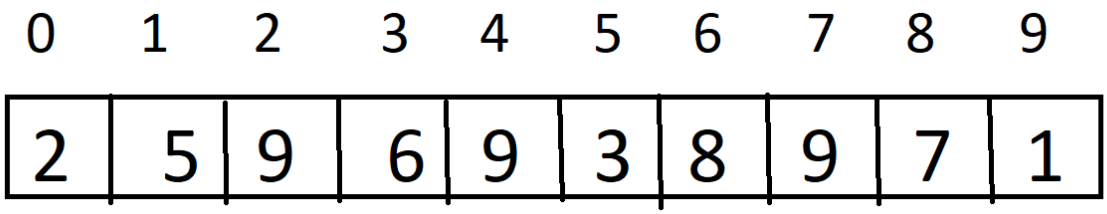
```
 arr = {1,3,4,2,2};
 cout << "The duplicate element is " << findDuplicate(arr) << endl;
}
```

Output: The duplicate element is 2
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)
Solution 2:Using frequency array
Approach: 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.
Code:

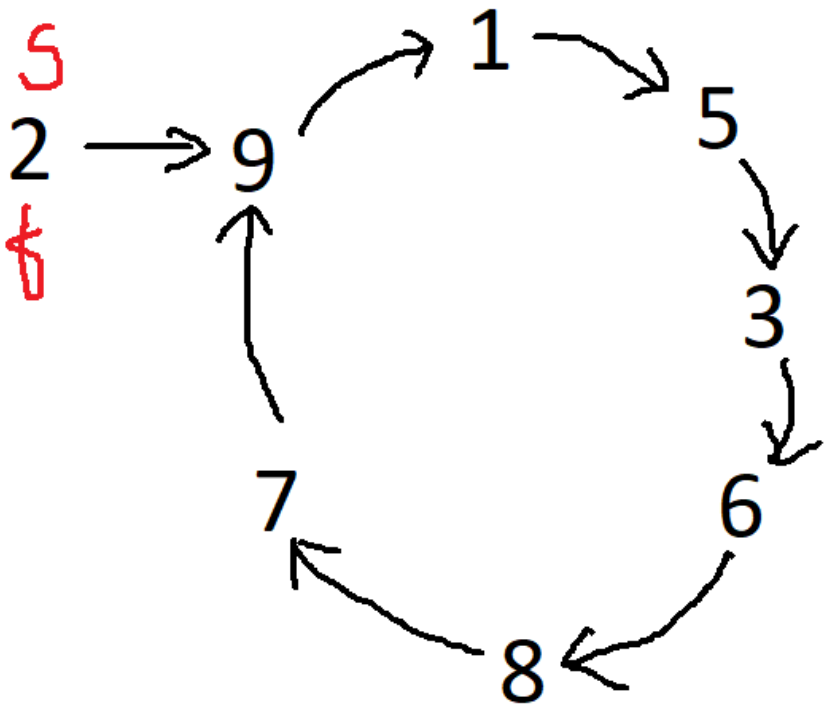
- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
int findDuplicate(vector < int > & arr) {
 int n = arr.size();
 int freq[n + 1] = {
  0
 };
 for (int i = 0; i < n; i++) {
  if (freq[arr[i]] == 0) {
   freq[arr[i]] += 1;
  } else {
   return arr[i];
  }
 }
 return 0;
}
int main() {
 vector < int > arr;
 arr = {1,3,4,2,3};
 cout << "The duplicate element is " << findDuplicate(arr) << endl;
}
```

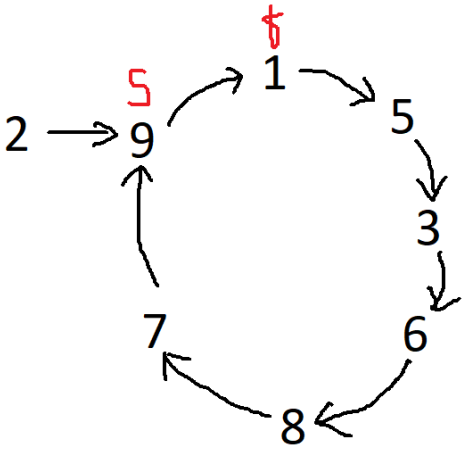
Output: The duplicate element is 3
Time Complexity: O(N), as we are traversing through the array only once.
Space Complexity: O(N), as we are using extra space for frequency array.
Solution 3: Linked List cycle method
Approach: Let's take an example and dry run on it to understand.

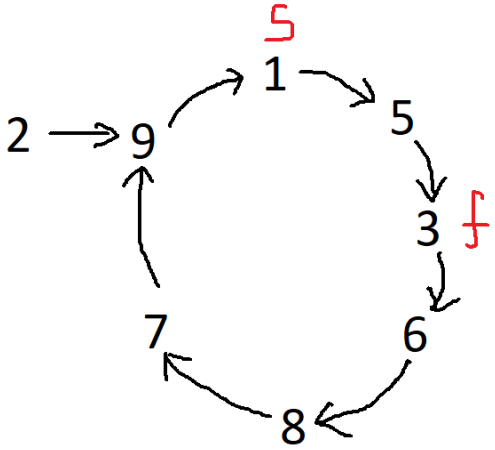


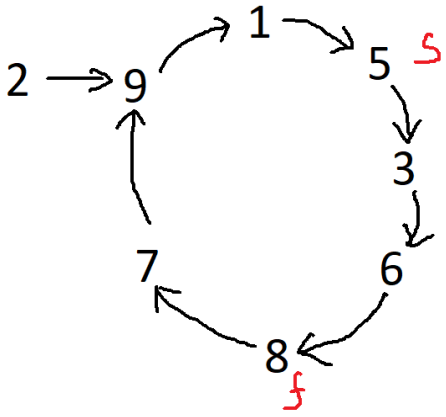
Initially, we have 2, then we got to the second index, we have 9, then we go to the 9th index, we have 1, then we go to the 1st index, we again have 5, then we go to the 5th index, we have 3, then we go to the 3rd index, we get 6, then we go to the 6th index, we get 8, then we go to the 8th index, we get 7, then we go to the 7th index and we get 9 and here cycle is formed.

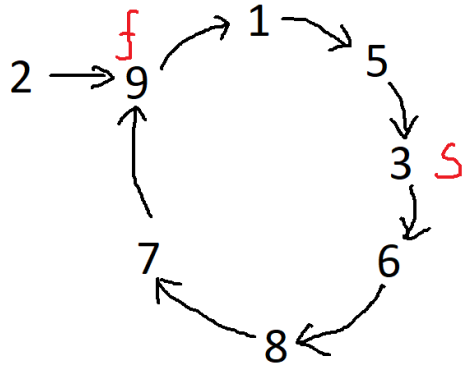


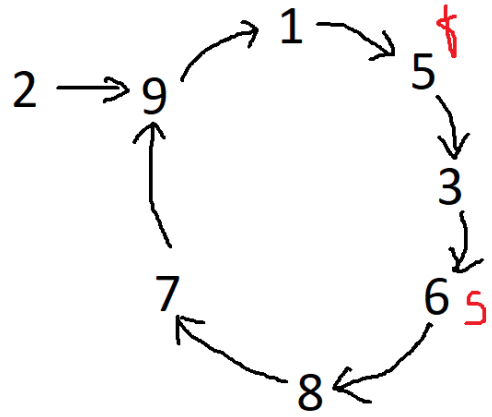
Now initially, the slow and fast pointer is at the start, the slow pointer moves by one step, and the fast pointer moves by 2 steps.

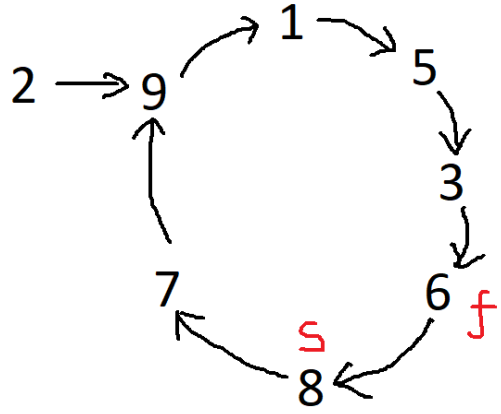

f
s
1
2
9
5
3
6
7
8

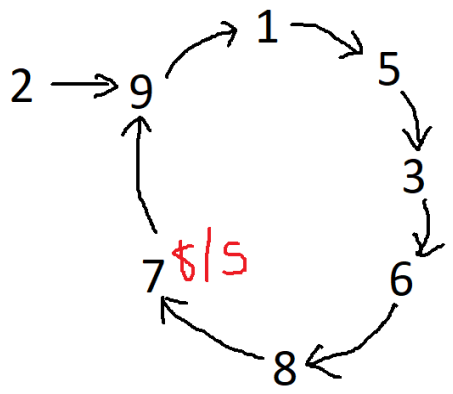

s
1
2
9
5
3
f
6
7
8


1
2
9
5
s
3
6
7
8
f

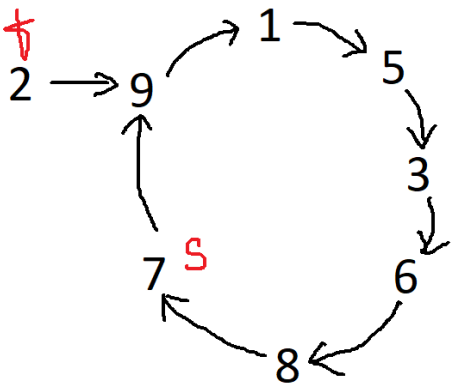

f
1
2
9
5
3
s
6
7
8

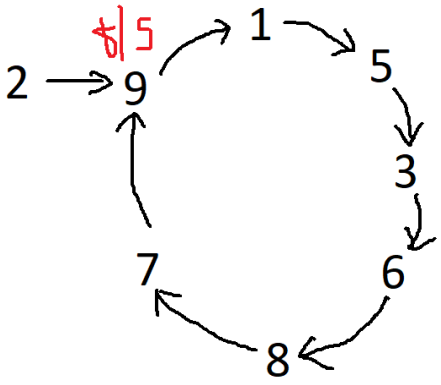

1
f
2
9
5
3
6
s
7
8


1
2
9
5
3
6
f
7
s
8

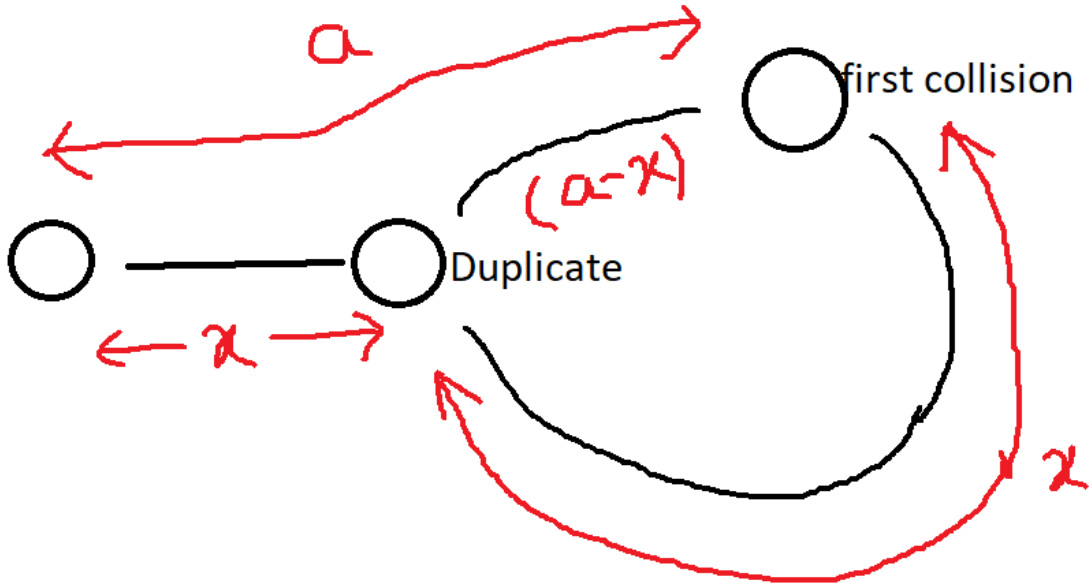


The slow and fast pointers meet at 7. Now take the fast pointer and place it at the first element i.e 2 and move the fast and slow pointer both by 1 step. The point where they collide will be the duplicate number.





So 9 is the duplicate number.
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 be x, as seen from the diagram, and so now fast and slow pointer both should move by one step.

Code:

- **C++ Code**
- **Java Code**

```
#include<bits/stdc++.h>

using namespace std;
int findDuplicate(vector < int > & nums) {
  int slow = nums[0];
  int fast = nums[0];
  do {
    slow = nums[slow];
    fast = nums[nums[fast]];
  } while (slow != fast);
  fast = nums[0];
  while (slow != fast) {
    slow = nums[slow];
    fast = nums[fast];
  }
  return slow;
}
int main() {
  vector < int > arr;
  arr = {1,3,4,2,3};
  cout << "The duplicate element is " << findDuplicate(arr) << endl;
```

}
Output:
The duplicate element is 3
Time complexity: O(N). Since we traversed through the array only once.
Space complexity: O(1).

Find the repeating and missing numbers

Problem Statement: You are given a read-only array of N integers with values also in the range [1,N] both inclusive. Each integer appears exactly once except A which appears twice and B which is missing. The task is to find the repeating and missing numbers A and B where A repeats twice and B is missing.
Example 1:
Input Format: array[] = {3,1,2,5,3}

Result: {3,4)

Explanation: A = 3 , B = 4
Since 3 is appearing twice and 4 is missing
Example 2:
Input Format: array[] = {3,1,2,5,4,6,7,5}

Result: {5,8)

Explanation: A = 5 , B = 8
Since 5 is appearing twice and 8 is missing

Solution

Disclaimer*: *Don't jump directly to the solution, try it out yourself first.
Solution 1: Using Count Sort
Intuition + Approach:
Since the numbers are from 1 to N in the array arr[]

1. **Take a substitute array of size N+1 and initalize it with 0.**
2. **Traverse the given array and increase the value of substitute[arr[i]] by one .**
3. **Then again traverse the substitute 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.
Dry Run: lets take the example of array[] = {3,1,2,5,3}
The size of the array is 5
We initialize a substitute array of size 6 with elements 0.
Now traversing through the array

1. **Found 3 at 0 index, increase the value of substitue array at index 3 by 1.**
2. **Found 1 at 1 index, increase the value of substitue array at index 1 by 1.**
3. **Found 2 at 2 index, increase the value of substitue array at index 2 by 1.**
4. **Found 5 at 3 index, increase the value of substitue array at index 5 by 1.**
5. **Found 3 at 4 index, increase the value of substitue array at index 3 by 1.**

Now Traversing through the substitute array
At index 3, the value is greater than 1 i.e 2. So A = 3.
At index 4, the value is 0 so B = 4.

Code:

```
vector<int> find_missing_repeating(vector<int> array)
{
   int n = array.size() + 1;
```

```
    vector<int> substitute(n, 0); // initializing the substitute array with 0 of size n+1.

    vector<int> ans;            // initializing the answer  array .

    for (int i = 0; i < array.size(); i++)
    {
        substitute[array[i]]++;
    }

    for (int i = 1; i <= array.size(); i++)
    {
        if (substitute[i] == 0 || substitute[i] > 1)
        {
            ans.push_back(i);
        }
    }

    return ans;
}
```

Time Complexity: O(N) + O(N) (as we are traversing 2 times)
Space Complexity: O(N) As we are making new substitute array
Solution 2: Using Maths
Intuition + Approach:
The idea is to convert the given problem into mathematical equations. Since we have two variables where one is missing and one is repeating, can we form two linear equations and then solve for the values of these two variables using the equations?
Let's see how.
Assume the missing number to be X and the repeating one to be Y.
Now since the numbers are from 1 to N in the array arr[]. Let's calculate sum of all integers from 1 to N and sum of squares of all integers from 1 to N. These can easily be done using mathematical formulae.
Therefore,
Sum of all elements from 1 to N:
S = N*(N+1)/2 ---- equation 1
And, Sum of squares of all elements from 1 to N:
P = N(N+1)(2N+1)/6. ----- equation 2
Similarly, find the sum of all elements of the array and sum of squares of all elements of the array respectively.

- **s1 = Sum of all elements of the array. —– equation 3**
- **P1 = Sum of squares of all elements of the array. ——– equation 4**

Now, if we subtract the sum of all elements of array from sum of all elements from 1 to N, that should give us the value for (X – Y).
Therefore,
(X-Y) = s – s1 = s'
Similarily,
X^2 – Y^2 = P – P1 = P'
or, (X + Y)(X – Y) = P'
or, (X + Y)*s' = P'
or, X + Y = P'/s'
Great,
we have the two equations we needed:
(X – Y) = s'

(X + Y) = P'/s'
We can use the two equations to solve and find values for X and Y respectively.
Note: here s and P can be large so take long long int data type.
Code:

- **C++ Code**

```
vector<int>missing_repeated_number(const vector<int> &A) {
  long long int len = A.size();

  long long int S = (len * (len+1) ) /2;
  long long int P = (len * (len +1) *(2*len +1) )/6;
  long long int missingNumber=0, repeating=0;

  for(int i=0;i<A.size(); i++){
    S -= (long long int)A[i];
    P -= (long long int)A[i]*(long long int)A[i];
  }

  missingNumber = (S + P/S)/2;

  repeating = missingNumber - S;

  vector <int> ans;

  ans.push_back(repeating);
  ans.push_back(missingNumber);

  return ans;
}
```

Time Complexity: O(N)
Space Complexity: O(1) As we are NOT USING EXTRA SPACE
Solution 3: XOR
Intuition + Approach:
Let x and y be the desired output elements.
Calculate the XOR of all the array elements.
xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
XOR the result with all numbers from 1 to n
xor1 = xor1^1^2^.....^n
xor1 will have the result as (x^y), as others would get canceled. Since we are doing XOR, we know xor of 1 and 0, is only 1, so all the set bits in xor1, means that the index bit is only set at x or y.

So we can take any set bit, in code we have taken the rightmost set bit, and iterate over, and divide the numbers into two hypothetical buckets.

If we check for numbers with that particular index bit set, we will get a set of numbers that belongs to the first bucket, also we will get another set of numbers belonging to the second bucket. The first bucket will be containing either x or y, similarly, the second bucket will also be containing either of x and y. XOR of all elements in the first bucket will give X or Y, and XOR of all elements of the second bucket will give either X or Y, since there will be double instances of every number in each bucket except the X or Y.

We just need to iterate again to check which one is X, and which one is y. Can be simply checked by linear iterations. For better understanding, you can check the video explanation.
Code:

- **C++ Code**
- **Java Code**

```
vector < int >Solution::repeatedNumber (const vector < int >&arr) {
  /* Will hold xor of all elements and numbers from 1 to n */
  int xor1;

  /* Will have only single set bit of xor1 */
  int set_bit_no;

  int i;
  int x = 0; // missing
  int y = 0; // repeated
  int n = arr.size();

  xor1 = arr[0];

  /* Get the xor of all array elements */
  for (i = 1; i < n; i++)
    xor1 = xor1 ^ arr[i];

  /* XOR the previous result with numbers from 1 to n */
  for (i = 1; i <= n; i++)
    xor1 = xor1 ^ i;

  /* Get the rightmost set bit in set_bit_no */
  set_bit_no = xor1 & ~(xor1 - 1);

  /* Now divide elements into two sets by comparing a rightmost set bit
    of xor1 with the bit at the same position in each element.
    Also, get XORs of two sets. The two XORs are the output elements.
    The following two for loops serve the purpose */

  for (i = 0; i < n; i++) {
    if (arr[i] & set_bit_no)
      /* arr[i] belongs to first set */
      x = x ^ arr[i];

    else
      /* arr[i] belongs to second set */
      y = y ^ arr[i];
  }

  for (i = 1; i <= n; i++) {
    if (i & set_bit_no)
      /* i belongs to first set */
      x = x ^ i;
```

```
      else
         /* i belongs to second set */
         y = y ^ i;
   }

   // NB! numbers can be swapped, maybe do a check ?
   int x_count = 0;
   for (int i=0; i<n; i++) {
      if (arr[i]==x)
         x_count++;
   }

   if (x_count==0)
      return {y, x};

   return {x, y};
}
```

Note: This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing. We can add one more step that checks which one is missing and which one is repeating by iterating over the array. This can be easily done in O(N) time.
Time Complexity: O(N)
Space Complexity: O(1) As we are NOT USING EXTRA SPACE

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].
Example 1:
Input Format: N = 5, array[] = {1,2,3,4,5}

Result: 0

Explanation: we have a sorted array and the sorted array has 0 inversions as for i < j you will never find a pair such that A[j] < A[i]. More clear example: 2 has index 1 and 5 has index 4 now 1 < 5 but 2 < 5 so this is not an inversion.
Example 2:
Input Format: N = 5, array[] = {5,4,3,2,1}

Result: 10

Explanation: we have a reverse sorted array and we will get the maximum inversions as for i < j we will always find a pair such that A[j] < A[i].
Example: 5 has index 0 and 3 has index 2 now (5,3) pair is inversion as 0 < 2 and 5 > 3 which will satisfy out conditions and for reverse sorted array we will get maximum inversions and that is (n)*(n-1) / 2.

For above given array there is 4 + 3 + 2 + 1 = 10 inversions.
Example 3:
Input Format: N = 5, array[] = {5,3,2,1,4}

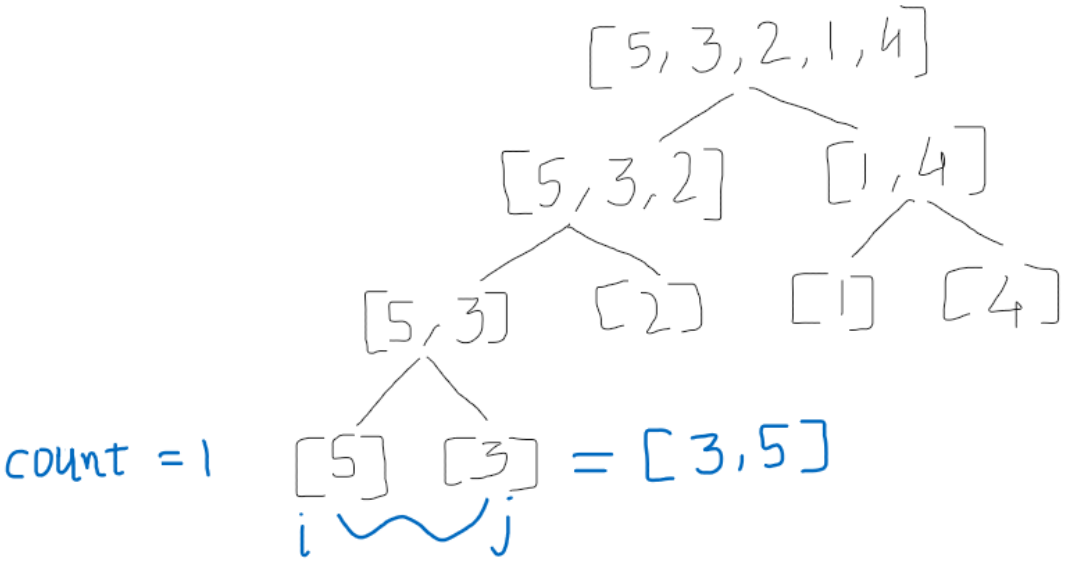
Result: 8

Explanation: There are 7 pairs (5,1), (5,3), (5,2), (5,4), (3,2), (3,1), (2,1) and we have left 2 pairs (2,4) and (1,4) as both are not satisfy our condition.

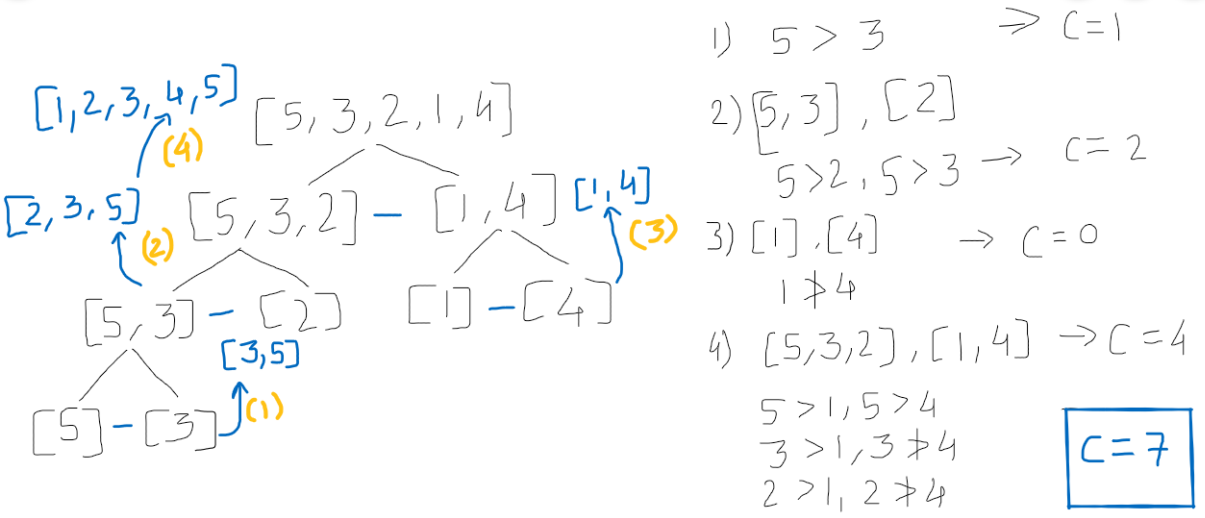
Solution

***Disclaimer*: *Don't jump directly to the solution, try it out yourself first*.**
Solution 1:
Intuition: Let's understand the Question more deeply we are required to give the total number of inversions and the inversions are: For i & j < size of an array if i < j then you have to find pair (A[i], A[j]) such that A[j] < A[i]. Let's take an example of array [5,3,2,1,4] and to satisfy the first condition which is i < j for i, j < N, we fix an element and traverse the next array elements then we are good to satisfy the first requirement.
The second condition which is for i < j we can only take pairs (A[i], A[j]) such that A[j] < A[i] and to satisfy this condition we will iterate through the array and check the condition A[j] < A[i] and if this is true then we will add 1 to answer.
But if we do it by brute force, it will cost O(n^2) time complexity because we have to add two nested loops to check the 2nd condition, but if we have the sorted array, then it could be easier to get the answer. How?
If an array is sorted, the array inversions are 0, and if reverse sorted, the array inversions are maximum.
But what if we have an unsorted array?
To sort the array, we will use mergeSort. In mergeSort, we will deep in one element, compare two elements, and put it in a new array in sorted order. By doing this for log(N) time, we will get the sorted array, and while comparing the two array elements, we will add some more lines of code such that it will count the inversion of the smaller array and slowly it will count for larger and larger array.

Approach:
Let's understand the algorithm by example. We slice the array in the middle and further slice it in merge sort, as shown in the figure.



The single element is always sorted after slicing to the bottom and getting them on an element as an array. Before returning the merged array with sorted numbers, we will count the inversion from there. How?
1st condition i < j above in the image, you can see that the right element's index is always greater, so while computing the inversion, we should take care only 2nd condition, which is if i < j then A[j] < A[i] to make a pair and add one to the count.
In the above example i < j as i is the 5's index and j is 3's index and (A[i] == 5) > (A[j] == 3) so we got our first inversion pair (5,3) after that merge then into one array [3,5] and return it for further computations now lets take another example:
[2,3,5] and [1,4] and count = 3. How to calculate it further?
Compare elements in 1st array with the 2nd array's all elements if 1's array's element is greater than 2's array then we will count it as inversion pair as 1st condition for inversion will always satisfy with right arrays. 2 > [1], 3 > [1], 5 > [1,4] so we will get 4 inversion pairs from this. and total inversion pair from [5,3,2,1,4] is 7.
Dry Run: I have explained the main dry run case above, and for a full dry run, I have added this image:



Code:

```
#include<bits/stdc++.h>
using namespace std;
int merge(int arr[],int temp[],int left,int mid,int right)
{
   int inv_count=0;
   int i = left;
   int j = mid;
   int k = left;
   while((i <= mid-1) && (j <= right)){
      if(arr[i] <= arr[j]){
         temp[k++] = arr[i++];
      }
      else
      {
         temp[k++] = arr[j++];
         inv_count = inv_count + (mid - i);
      }
   }

   while(i <= mid - 1)
      temp[k++] = arr[i++];
   while(j <= right)
      temp[k++] = arr[j++];

   for(i = left ; i <= right ; i++)
      arr[i] = temp[i];

   return inv_count;
}

int merge_Sort(int arr[],int temp[],int left,int right)
{
   int mid,inv_count = 0;
   if(right > left)
   {
      mid = (left + right)/2;

      inv_count += merge_Sort(arr,temp,left,mid);
      inv_count += merge_Sort(arr,temp,mid+1,right);

      inv_count += merge(arr,temp,left,mid+1,right);
   }
   return inv_count;
}

int main()
{
   int arr[]={5,3,2,1,4};
   int n=5;
   int temp[n];
   int ans = merge_Sort(arr,temp,0,n-1);
```

```
    cout<<"The total inversions are "<<ans<<endl;
    return 0;
}
```