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

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:

```
#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:

```
#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:

```
#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:

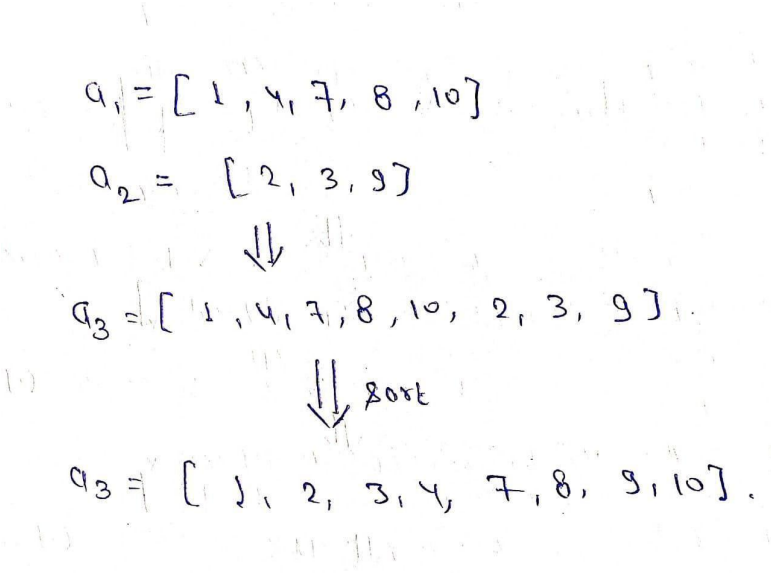
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

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:

```
#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:

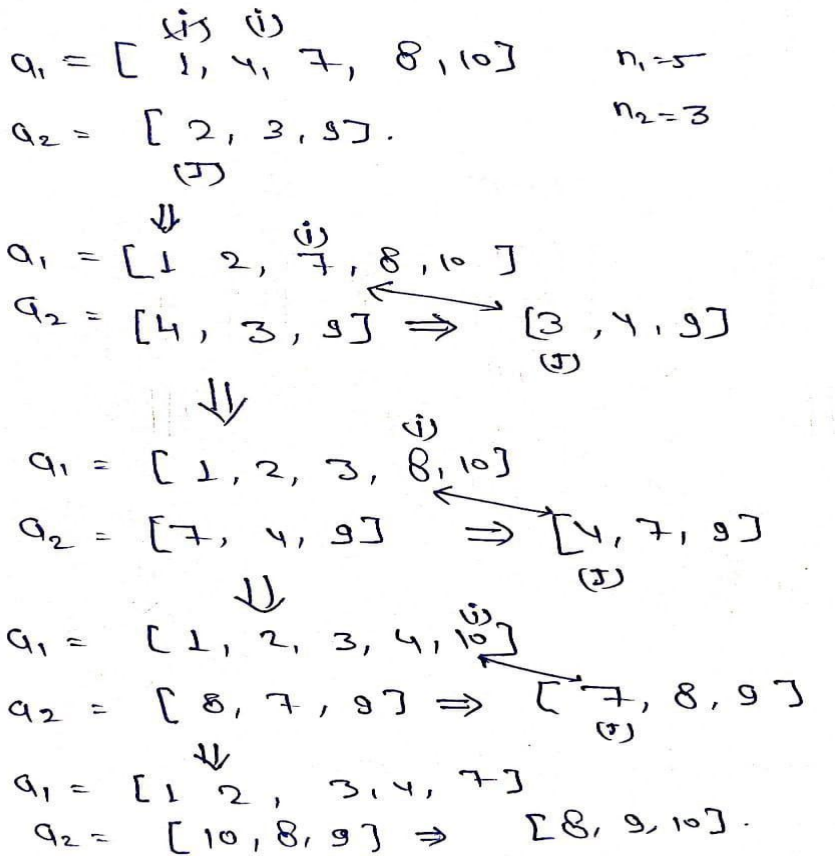
Before merge:
1 4 7 8 10
2 3 9
After merge:
1 2 3 4 7
8 9 10

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:

```
#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
```

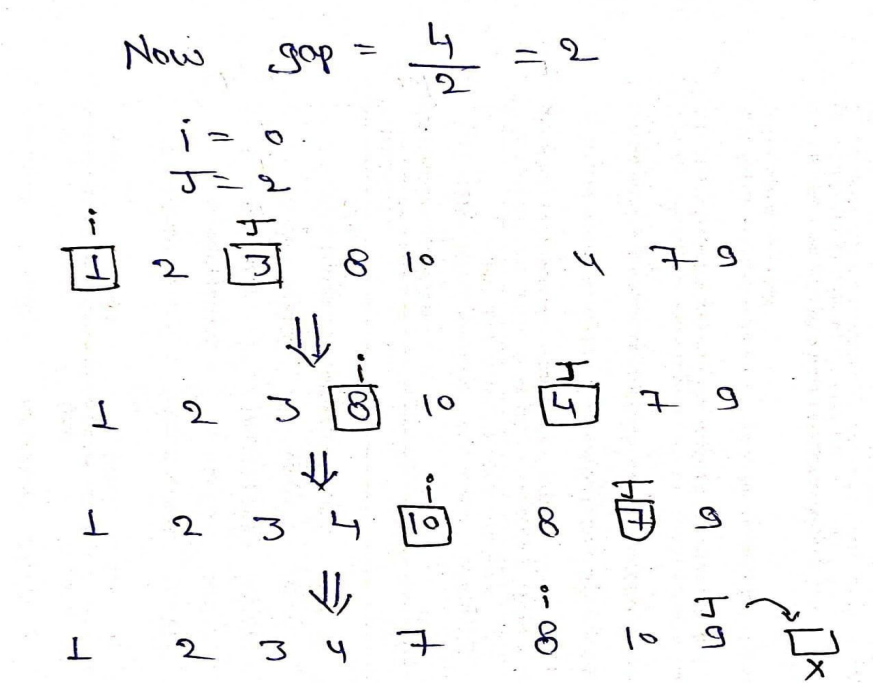
```
    for (k = 1; k < m && arr2[k] < first; k++) {
      arr2[k - 1] = arr2[k];
    }
    arr2[k - 1] = first;
  }
}
```

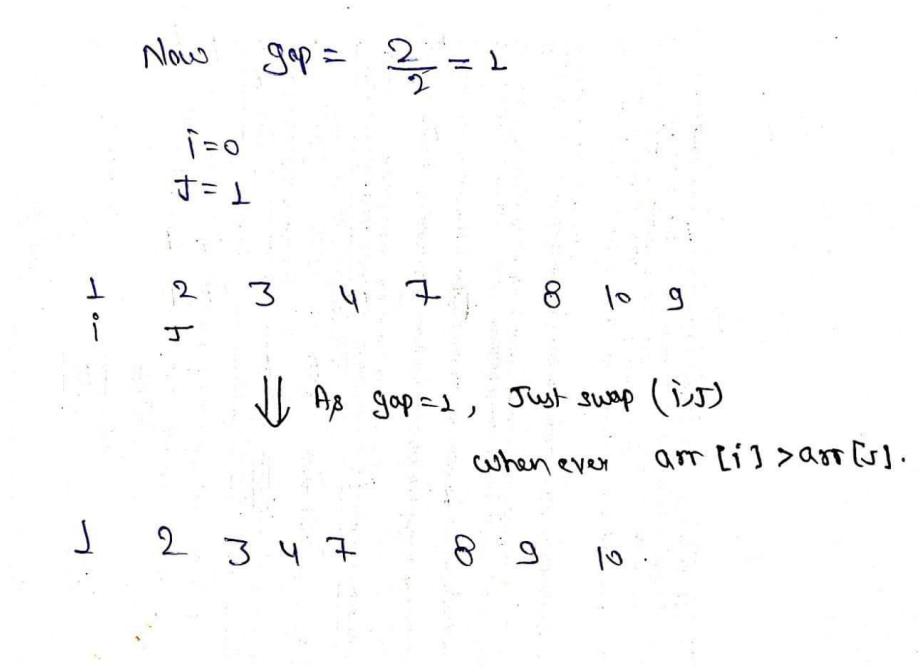
Output:
Before merge:
1 4 7 8 10
2 3 9
After merge:
1 2 3 4 7
8 9 10

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 loop from pointer 1 & pointer 2 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:

```
#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);
    }
  }
```

```
}
```

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:
```
#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)

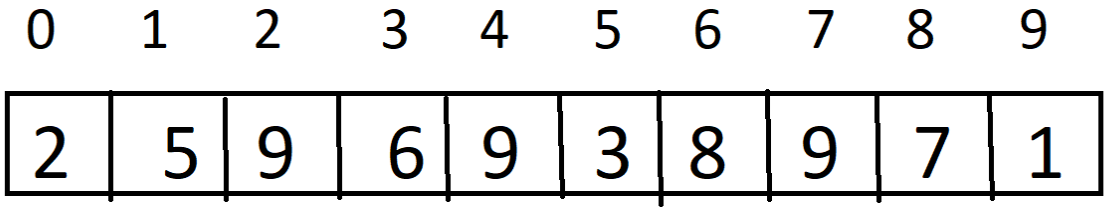
Solution 2:Using frequency array
Approach: Take a frequency array of size N+1 and initialize it to 0. Now traverse through the array and if the frequency of the element is 0 increase it by 1, else if the frequency is not 0 then that element is the required answer.
Code:
```
#include<bits/stdc++.h>
using 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;
}
```

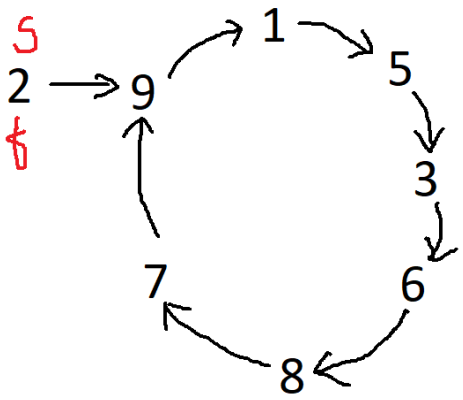
Output: The duplicate element is 3

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.

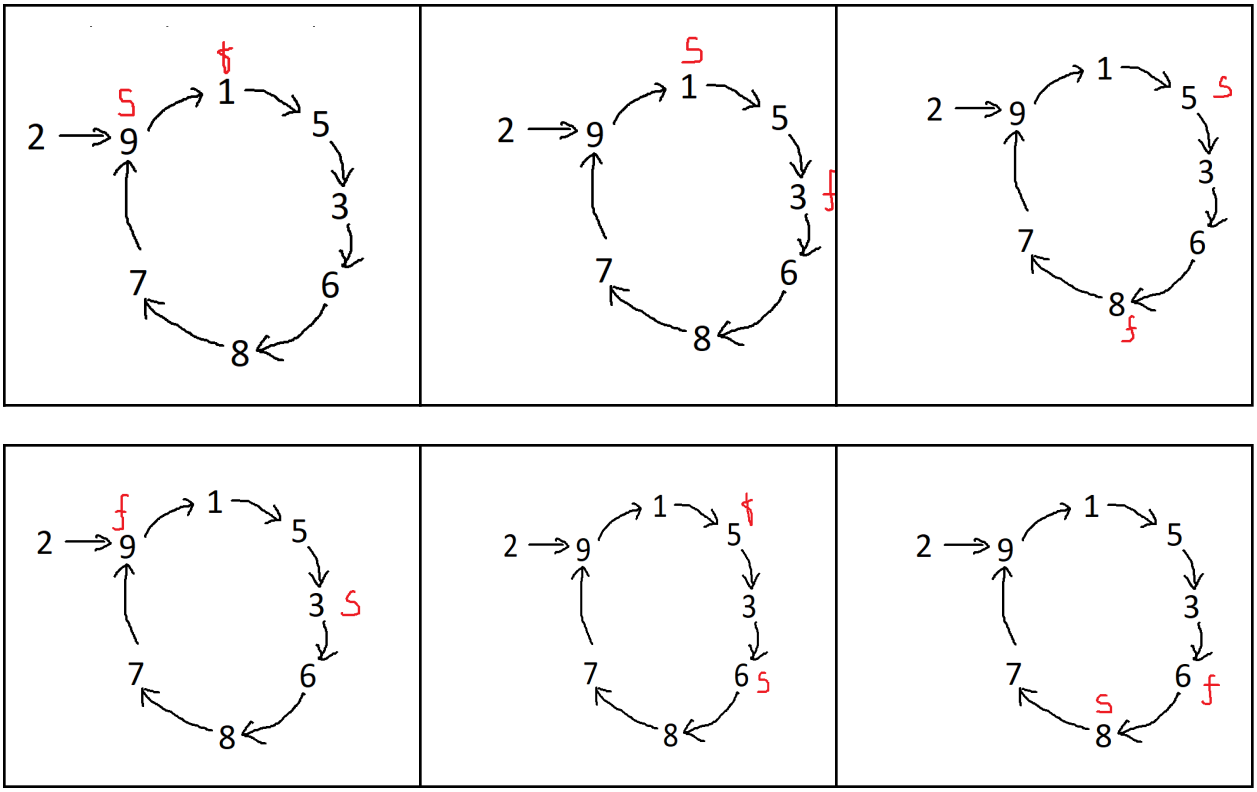
Solution 3: Linked List cycle method
Approach: Let's take an example and dry run on it to understand.

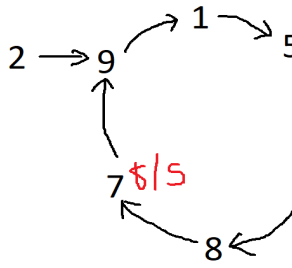


Initially, we have 2, then we got to the second index, we have 9, then we go to the 9th index, we have 1, then we go to the 1st index, we again have 5, then we go to the 5th index, we have 3, then we go to the 3rd index, we get 6, then we go to the 6th index, we get 8, then we go to the 8th index, we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

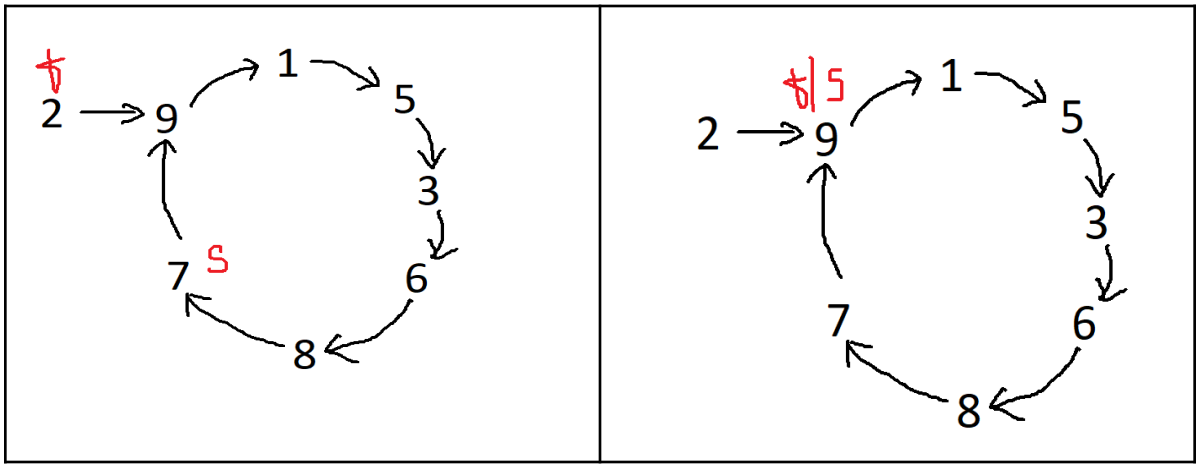


Now initially, the slow and fast pointer is at the start, the slow pointer moves by one step, and the fast pointer moves by 2 steps.

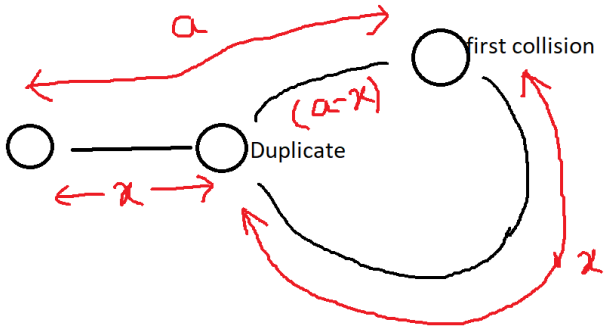




The slow and fast pointers meet at 7. Now take the fast pointer and place it at the first element i.e 2 and move the fast and slow pointer both by 1 step. The point where they collide will be the duplicate number.



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:

```
#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:

```
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:

```
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

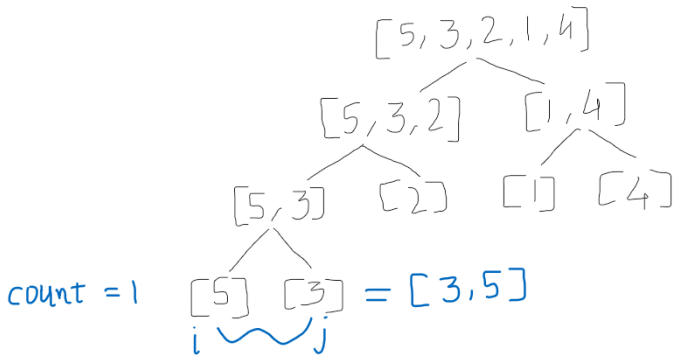
Explanation: There are 7 pairs (5,1), (5,3), (5,2), (5,4), (3,2), (3,1), (2,1) and we have left 2 pairs (2,4) and (1,4) as both are not satisfy our condition.

Solution

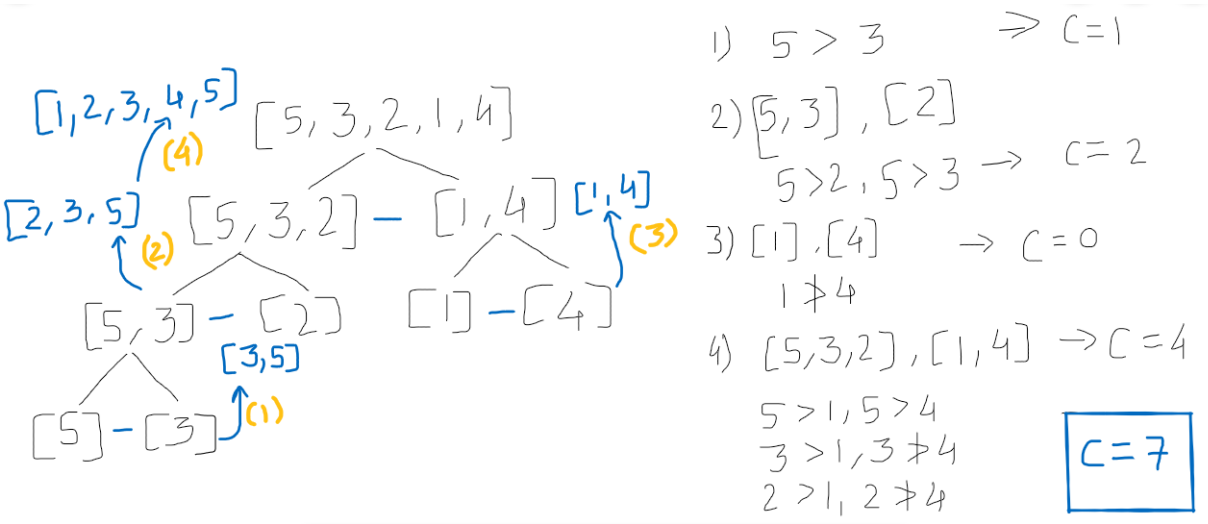
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Solution 1:

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;
}
```