1. Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do linear search.The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.
We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

```
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
```

```
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array")
                   : printf("Element is present at index %d",
                            result);
    return 0;
}
```

Output :

```
Element is present at index 3
```

Iterative implementation of Binary Search

```
// C program to implement iterative Binary Search
#include <stdio.h>

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }
```

```
    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                            " in array")
                   : printf("Element is present at "
                            "index %d",
                            result);
    return 0;
}
```

Output :

```
Element is present at index 3
```

Time Complexity:
The time complexity of Binary Search can be written as

```
T(n) = T(n/2) + c
```

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is $Theta(Logn)$.

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

2. Given an array A[] and a number x, check for pair in A[] with sum as x

Write a program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

METHOD 1 (Use Sorting)
Algorithm :

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
      (a) Initialize first to the leftmost index: l = 0
      (b) Initialize second  the rightmost index:  r = ar_size-1
3) Loop while l < r.
      (a) If (A[l] + A[r] == sum)  then return 1
      (b) Else if( A[l] + A[r] <  sum )  then l++
      (c) Else r--
4) No candidates in whole array - return 0
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then (-)(nlogn) in worst case. If we use Quick Sort then O(n^2) in worst case.
Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and O(1) for Heap Sort.

Example :
Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16
Sort the array
A = {-8, 1, 4, 6, 10, 45}

We will increment'l' when sum of pair is less than required sum and decrement 'r' when sum of pair is more than required sum.
This is because when sum is less than required then we need to get the number which could increase our sum of pair so we move from left to right(also array is sorted)thus "l++" and vice versa.

Initialize l = 0, r = 5
A[l] + A[r] (-8 + 45) > 16 => decrement r. Now r = 4
A[l] + A[r] (-8 + 10) increment l. Now l = 1
A[l] + A[r] (1 + 10) increment l. Now l = 2
A[l] + A[r] (4 + 10) increment l. Now l = 3
A[l] + A[r] (6 + 10) == 16 => Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Below is the implementation of the above approach.

```cpp
// C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value

#include <bits/stdc++.h>
using namespace std;

// Function to check if array has 2 elements
// whose sum is equal to the given value
bool hasArrayTwoCandidates(int A[], int arr_size,
                           int sum)
{
    int l, r;

    /* Sort the elements */
    sort(A, A + arr_size);

    /* Now look for the two candidates in
       the sorted array*/
    l = 0;
    r = arr_size - 1;
    while (l < r) {
        if (A[l] + A[r] == sum)
            return 1;
        else if (A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return 0;
}

/* Driver program to test above function */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, -8 };
    int n = 16;
    int arr_size = sizeof(A) / sizeof(A[0]);

    // Function calling
    if (hasArrayTwoCandidates(A, arr_size, n))
        cout << "Array has two elements with given sum";
    else
        cout << "Array doesn't have two elements with given sum";

    return 0;
}
```

Output :

```
Array has two elements with the given sum
```

METHOD 2 (Use Hashing)
This method works in O(n) time.

```
1) Initialize an empty hash table s.
2) Do following for each element A[i] in A[]
   (a)    If s[x - A[i]] is set then print the pair (A[i], x - A[i])
   (b)    Insert A[i] into s.
```

Below is the implementation of the above approach :

```
// C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value
#include <bits/stdc++.h>

using namespace std;

void printPairs(int arr[], int arr_size, int sum)
{
    unordered_set<int> s;
    for (int i = 0; i < arr_size; i++) {
        int temp = sum - arr[i];

        if (s.find(temp) != s.end())
            cout << "Pair with given sum " << sum << " is (" <<
arr[i] << ", " << temp << ")" << endl;

        s.insert(arr[i]);
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int n = 16;
    int arr_size = sizeof(A) / sizeof(A[0]);

    // Function calling
    printPairs(A, arr_size, n);

    return 0;
}
```

Output:

```
Pair with given sum 16 is (10, 6)
```

Time Complexity: O(n)
Auxiliary Space: O(n) where n is size of array.
If range of numbers include negative numbers then also it works

3. Majority Element

Write a function which takes an array and prints the majority element (if it exists), otherwise prints “No Majority Element”. A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).
Examples :

```
Input : {3, 3, 4, 2, 4, 4, 2, 4, 4}

Output : 4


Input : {3, 3, 4, 2, 4, 4, 2, 4}

Output : No Majority Element
```

Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

METHOD 1 (Basic)
The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than n/2 then break the loops and return the element having maximum count. If maximum count doesn’t become more than n/2 then majority element doesn’t exist.
Below is the implementation of the above approach :

```
// C++ program to find Majority
// element in an array
#include <bits/stdc++.h>
using namespace std;

// Function to find Majority element
// in an array
void findMajority(int arr[], int n)
{
    int maxCount = 0;
    int index = -1; // sentinels
    for(int i = 0; i < n; i++)
    {
        int count = 0;
        for(int j = 0; j < n; j++)
        {
            if(arr[i] == arr[j])
            count++;
        }

        // update maxCount if count of
        // current element is greater
        if(count > maxCount)
```

```
        {
            maxCount = count;
            index = i;
        }
    }

    // if maxCount is greater than n/2
    // return the corresponding element
    if (maxCount > n/2)
       cout << arr[index] << endl;

    else
        cout << "No Majority Element" << endl;
}

// Driver code
int main()
{
    int arr[] = {1, 1, 2, 1, 3, 5, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function calling
    findMajority(arr, n);

    return 0;
}
```

Output :

```
1
```

Time Complexity : O(n*n).
Auxiliary Space : O(1).

METHOD 2 (Using Binary Search Tree)
Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than n/2 then return.
The method works well for the cases where n/2+1 occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 1, 2, 3, 4}.

Time Complexity : If a Binary Search Tree is used then time complexity will be O(n^2). If a self-balancing-binary-search tree is used then O(nlogn)
Auxiliary Space : O(n)

METHOD 3 (Using Moore’s Voting Algorithm)
This is a two step process.
NOTE : This Method only works when we are given that majority element do exist in the array , otherwise this method won’t work , as in the problem definition we said that majority element may or may not exist but for applying

this approach you can assume that majority element do exist in the given input array

1. The first step gives the element that may be majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise it will return candidate for majority element.
2. Check if the element obtained from above step is majority element.This step is necessary as we are not always sure that element return by first step is majority element.

1. Finding a Candidate :
The algorithm for first phase that works in O(n) is known as Moore's Voting Algorithm. Basic idea of the algorithm is that if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element.

```
findCandidate(a[], size)

1.  Initialize index and count of majority element

    maj_index = 0, count = 1

2.  Loop for i = 1 to size – 1

   (a) If a[maj_index] == a[i]

         count++

   (b) Else

       count--;

   (c) If count == 0

         maj_index = i;

         count = 1

3.  Return a[maj_index]
```

Above algorithm loops through each element and maintains a count of a[maj_index]. If the next element is same then increment the count, if the next element is not same then decrement the count, and if the count reaches 0 then changes the maj_index to the current element and set the count again to 1. So, the first phase of the algorithm gives us a candidate element.

In the second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

Example :
Let the array be A[] = 2, 2, 3, 5, 2, 2, 6

- Initialize maj_index = 0, count = 1
- Next element is 2, which is same as a[maj_index] => count = 2
- Next element is 3, which is different from a[maj_index] => count = 1
- Next element is 5, which is different from a[maj_index] => count = 0
  Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1

- Next element is 2, which is different from a[maj_index] => count = 0
  Since count = 0, change candidate for majority element to 2 => maj_index = 4
- Next element is 2, which is same as a[maj_index] => count = 2
- Next element is 6, which is different from a[maj_index] => count = 1
- Finally candidate for majority element is 2.

First step uses Moore's Voting Algorithm to get a candidate for majority element.

2. Check if the element obtained in step 1 is majority element or not :

```
printMajority (a[], size)

1.  Find the candidate for majority

2.  candidate is majority. i.e., appears more than n/2 times.

       Print If the candidate

3.  Else

      Print "No Majority Element"
```

Below is the implementation of the above approach :

```
/* C++ Program for finding out
   majority element in an array */
#include <bits/stdc++.h>
using namespace std;

/* Function to find the candidate for Majority */
int findCandidate(int a[], int size)
{
    int maj_index = 0, count = 1;
    for (int i = 1; i < size; i++)
    {
        if (a[maj_index] == a[i])
            count++;
        else
            count--;
        if (count == 0)
        {
            maj_index = i;
            count = 1;
        }
    }
    return a[maj_index];
}

/* Function to check if the candidate
   occurs more than n/2 times */
bool isMajority(int a[], int size, int cand)
{
```

```
    int count = 0;
    for (int i = 0; i < size; i++)

    if (a[i] == cand)
    count++;

    if (count > size/2)
    return 1;

    else
    return 0;
}

/* Function to print Majority Element */
void printMajority(int a[], int size)
{
   /* Find the candidate for Majority*/
   int cand = findCandidate(a, size);

   /* Print the candidate if it is Majority*/
   if (isMajority(a, size, cand))
   cout << " " << cand << " ";

   else
   cout << "No Majority Element";
}


/* Driver function to test above functions */
int main()
{
    int a[] = {1, 3, 3, 1, 2};
    int size = (sizeof(a))/sizeof(a[0]);

    // Function calling
    printMajority(a, size);

    return 0;
}
```

Output:

```
No Majority Element
```

Time Complexity: O(n)
Auxiliary Space : O(1)

METHOD 4 (Using Hashmap) :This method is somewhat similar to Moore voting algorithm in terms of time complexity, but in this case there is no need of second step of Moore voting algorithm.But as usual, here space complexity becomes O(n).

In Hashmap(key-value pair), at value,maintain a count for each element(key) and whenever count is greater than half of array length, we are just returning that key(majority element).
Time Complexity : O(n)
Auxiliary Space : O(n)
Below is the implementation.

```
* C++ program for finding out majority
element in an array */
#include <bits/stdc++.h>
using namespace std;

void findMajority(int arr[], int size)
{
    unordered_map<int, int> m;
    for(int i = 0; i < size; i++)
        m[arr[i]]++;
    int count = 0;
    for(auto i : m)
    {
        if(i.second > size / 2)
        {
            count =1;
            cout << "Majority found :- " << i.first<<endl;
            break;
        }
    }
    if(count == 0)
        cout << "No Majority element" << endl;
}

// Driver code
int main()
{
    int arr[] = {2, 2, 2, 2, 5, 5, 2, 3, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function calling
    findMajority(arr, n);

    return 0;
}
```

Output:

```
Majority found :- 2
```

4. Majority Element | Set-2 (Hashing)

Given an array of size N, find the majority element. The majority element is the element that appears more than $\frac{n}{2}$ times in the given array.

Examples:

```
Input: [3, 2, 3]

Output: 3


Input: [2, 2, 1, 1, 1, 2, 2]

Output: 2
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

The problem has been solved using 4 different methods in the previous post. In this post hashing based solution is implemented. We count occurrences of all elements. And if count of any element becomes more than n/2, we return it.

Hence if there is a majority-element, it will be the value of the key.

Below is the implementation of the above approach:

```
#include<bits/stdc++.h>
using namespace std;

#define ll long long int

// function to print the majority Number
int majorityNumber(int arr[], int n)
{
    int ans = -1;
    unordered_map<int, int>freq;
    for (int i = 0; i < n; i++)
    {
        freq[arr[i]]++;
        if (freq[arr[i]] > n / 2)
            ans = arr[i];
    }
    return ans;
}

// Driver code
int main()
{
    int a[] = {2, 2, 1, 1, 1, 2, 2};
    int n = sizeof(a) / sizeof(int);
```

```
    cout << majorityNumber(a, n);
    return 0;
}

// This code is contributed
// by sahishelangia
```

Output:

```
2
```

Below is the time and space complexity of the above algorithm:-

Time Complexity : O(n)
Auxiliary Space : O(n)

5. Maximum difference between two elements such that larger element appears after the smaller number

Given an array arr[] of integers, find out the maximum difference between any two elements such that larger element appears after the smaller number.

Examples :

```
Input : arr = {2, 3, 10, 6, 4, 8, 1}

Output : 8

Explanation : The maximum difference is between 10 and 2.


Input : arr = {7, 9, 5, 6, 3, 2}

Output : 2

Explanation : The maximum difference is between 9 and 7.
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple)
Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far. Below is the implementation of the above approach :

```
// C++ program to find Maximum difference
// between two elements such that larger
// element appears after the smaller number
#include <bits/stdc++.h>
using namespace std;

/* The function assumes that there are
   at least two elements in array. The
   function returns a negative value if the
   array is sorted in decreasing order and
   returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
  int max_diff = arr[1] - arr[0];
  for (int i = 0; i < arr_size; i++)
  {
    for (int j = i+1; j < arr_size; j++)
    {
      if (arr[j] - arr[i] > max_diff)
        max_diff = arr[j] - arr[i];
    }
  }
```

```
  return max_diff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 90, 10, 110};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function calling
  cout << "Maximum difference is " << maxDiff(arr, n);

  return 0;
}
```

Output :

```
Maximum difference is 109
```

Time Complexity : O(n^2)
Auxiliary Space : O(1)
Method 2 (Tricky and Efficient)
In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:
1) Maximum difference found so far (max_diff).
2) Minimum number visited so far (min_element).

```
// C++ program to find Maximum difference
// between two elements such that larger
// element appears after the smaller number
#include <bits/stdc++.h>
using namespace std;

/* The function assumes that there are
   at least two elements in array. The
   function returns a negative value if the
   array is sorted in decreasing order and
   returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
     // Maximum difference found so far
     int max_diff = arr[1] - arr[0];

     // Minimum number visited so far
     int min_element = arr[0];
     for(int i = 1; i < arr_size; i++)
     {
       if (arr[i] - min_element >
max_diff)
       max_diff = arr[i] - min_element;
```

```
        if (arr[i] < min_element)
        min_element = arr[i];
      }

      return max_diff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 90, 10, 110};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function calling
  cout << "Maximum difference is " << maxDiff(arr, n);

  return 0;
}
```

Output:

```
Maximum difference is 109
```

Time Complexity : O(n)
Auxiliary Space : O(1)
Like min element, we can also keep track of max element from right side. Thanks to Katamaran for suggesting this approach. Below is the implementation :

```
// C++ program to find Maximum difference
// between two elements such that larger
// element appears after the smaller number
#include <bits/stdc++.h>
using namespace std;

/* The function assumes that there are
   at least two elements in array. The
   function returns a negative value if the
   array is sorted in decreasing order and
   returns 0 if elements are equal */
int maxDiff(int arr[], int n)
{
    // Initialize Result
    int maxDiff = -1;

    // Initialize max element from right side
    int maxRight = arr[n-1];

    for (int i = n-2; i >= 0; i--)
    {
```

```
        if (arr[i] > maxRight)
            maxRight = arr[i];
        else
        {
            int diff = maxRight - arr[i];
            if (diff > maxDiff)
            {
                maxDiff = diff;
            }
        }
    }
    return maxDiff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 90, 10, 110};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function calling
  cout << "Maximum difference is " << maxDiff(arr, n);

  return 0;
}
```

Output:

```
Maximum difference is 109
```

Method 3 (Another Tricky Solution)
First find the difference between the adjacent elements of the array and store all differences in an auxiliary array diff[] of size n-1. Now this problems turns into finding the maximum sum subarray of this difference array.Thanks to Shubham Mittal for suggesting this solution. Below is the implementation :

```
// C++ program to find Maximum difference
// between two elements such that larger
// element appears after the smaller number
#include <bits/stdc++.h>
using namespace std;

/* The function assumes that there are
   at least two elements in array. The
   function returns a negative value if the
   array is sorted in decreasing order and
   returns 0 if elements are equal */
int maxDiff(int arr[], int n)
{
    // Create a diff array of size n-1.
    // The array will hold the difference
    // of adjacent elements
```

```
    int diff[n-1];
    for (int i=0; i < n-1; i++)
        diff[i] = arr[i+1] - arr[i];

    // Now find the maximum sum
    // subarray in diff array
    int max_diff = diff[0];
    for (int i=1; i<n-1; i++)
    {
        if (diff[i-1] > 0)
            diff[i] += diff[i-1];
        if (max_diff < diff[i])
            max_diff = diff[i];
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {80, 2, 6, 3, 100};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function calling
  cout << "Maximum difference is " << maxDiff(arr, n);

  return 0;
}
```

Output:

```
Maximum difference is 98
```

Time Complexity : O(n)
Auxiliary Space : O(n)
We can modify the above method to work in O(1) extra space. Instead of creating an auxiliary array, we can calculate diff and max sum in same loop. Following is the space optimized version.

```
// C++ program to find Maximum difference
// between two elements such that larger
// element appears after the smaller number
#include <bits/stdc++.h>
using namespace std;

/* The function assumes that there are
   at least two elements in array. The
   function returns a negative value if the
   array is sorted in decreasing order and
   returns 0 if elements are equal */
```

```
int maxDiff (int arr[], int n)
{
    // Initialize diff, current sum and max sum
    int diff = arr[1]-arr[0];
    int curr_sum = diff;
    int max_sum = curr_sum;

    for(int i=1; i<n-1; i++)
    {
        // Calculate current diff
        diff = arr[i+1]-arr[i];

        // Calculate current sum
        if (curr_sum > 0)
        curr_sum += diff;
        else
        curr_sum = diff;

        // Update max sum, if needed
        if (curr_sum > max_sum)
        max_sum = curr_sum;
    }

    return max_sum;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {80, 2, 6, 3, 100};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function calling
  cout << "Maximum difference is " << maxDiff(arr, n);

  return 0;
}
```

Output:

```
Maximum difference is 98
```

Time Complexity : O(n)
Auxiliary Space : O(1)

6. Find the Number Occurring Odd Number of Times

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

Examples :

```
Input : arr = {1, 2, 3, 2, 3, 1, 3}
Output : 3


Input : arr = {5, 7, 2, 7, 5, 2, 5}
Output : 5
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

A Simple Solution is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is O(n2).

Below is the implementation of the brute force approach :

```
// C++ program to find the element
// occurring odd number of times
#include<bits/stdc++.h>
using namespace std;

// Function to find the element
// occurring odd number of times
int getOddOccurrence(int arr[], int arr_size)
{
    for (int i = 0; i < arr_size; i++) {

        int count = 0;

        for (int j = 0; j < arr_size; j++)
        {
            if (arr[i] == arr[j])
                count++;
        }
        if (count % 2 != 0)
            return arr[i];
    }
    return -1;
}

// driver code
int main()
```

```
    {
        int arr[] = { 2, 3, 5, 4, 5, 2,
                      4, 3, 5, 2, 4, 4, 2 };
        int n = sizeof(arr) / sizeof(arr[0]);

        // Function calling
        cout << getOddOccurrence(arr, n);

        return 0;
    }
```

Output :

```
5
```

A Better Solution is to use Hashing. Use array elements as key and their counts as value. Create an empty hash table. One by one traverse the given array elements and store counts. Time complexity of this solution is O(n). But it requires extra space for hashing.

Program :

```
// C++ program to find the element
// occurring odd number of times
#include <bits/stdc++.h>
using namespace std;

// function to find the element
// occurring odd number of times
int getOddOccurrence(int arr[],int size)
{

    // Defining HashMap in C++
    unordered_map<int, int> hash;

    // Putting all elements into the HashMap
    for(int i = 0; i < size; i++)
    {
        hash[arr[i]]++;
    }
    // Iterate through HashMap to check an element
    // occurring odd number of times and return it
    for(auto i : hash)
    {
        if(i.second % 2 != 0)
        {
            return i.first;
        }
    }
    return -1;
}
```

```
// Driver code
int main()
{
    int arr[] = { 2, 3, 5, 4, 5, 2, 4,
                  3, 5, 2, 4, 4, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function calling
    cout << getOddOccurrence(arr, n);

    return 0;
}

// This code is contributed by codeMan_d.
```

Output :

```
5
```

The Best Solution is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x.
Below is the implementation of the above approach.

```
// C++ program to find the element
// occurring odd number of times
#include <bits/stdc++.h>
using namespace std;

// Function to find element occurring
// odd number of times
int getOddOccurrence(int ar[], int ar_size)
{
    int res = 0;
    for (int i = 0; i < ar_size; i++)
        res = res ^ ar[i];

    return res;
}

/* Driver function to test above function */
int main()
{
    int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
    int n = sizeof(ar)/sizeof(ar[0]);

    // Function calling
    cout << getOddOccurrence(ar, n);

    return 0;
```

```
}
```
Output :
```
5
```

Time Complexity: O(n)

7. Segregate 0s and 1s in an array

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

```
Input array   =  [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]

Output array =  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Count 0s or 1s)
Thanks to Naveen for suggesting this method.
1) Count the number of 0s. Let count be C.
2) Once we have count, we can put C 0s at the beginning and 1s at the remaining n – C positions in array.
Time Complexity : O(n)

```
// C++ code to Segregate 0s and 1s in an array
#include <bits/stdc++.h>
using na
mespace std;

// Function to segregate 0s and 1s
void segregate0and1(int arr[], int n)
{
    int count = 0; // Counts the no of zeros in arr

    for (int i = 0; i < n; i++) {
        if (arr[i] == 0)
            count++;
    }

    // Loop fills the arr with 0 until count
    for (int i = 0; i < count; i++)
        arr[i] = 0;

    // Loop fills remaining arr space with 1
    for (int i = count; i < n; i++)
        arr[i] = 1;
}

// Function to print segregated array
void print(int arr[], int n)
{
    cout << "Array after segregation is ";

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}
```

```
// Driver function
int main()
{
    int arr[] = { 0, 1, 0, 1, 1, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);

    segregate0and1(arr, n);
    print(arr, n);

    return 0;
}

// This code is contributed by Sahil_Bansall
```

Output :

```
Array after segregation is 0 0 1 1 1 1
```

The method 1 traverses the array two times. Method 2 does the same in a single pass.

Method 2 (Use two indexes to traverse)
Maintain two indexes. Initialize first index left as 0 and second index right as n-1.
Do following while left < right
a) Keep incrementing index left while there are 0s at it
b) Keep decrementing index right while there are 1s at it
c) If left < right then exchange arr[left] and arr[right]
Implementation:

```
// C++ program to sort a binary array in one pass
#include <bits/stdc++.h>
using namespace std;

/*Function to put all 0s on left and all 1s on right*/
void segregate0and1(int arr[], int size)
{
    /* Initialize left and right indexes */
    int left = 0, right = size-1;

    while (left < right)
    {
        /* Increment left index while we see 0 at left */
        while (arr[left] == 0 && left < right)
            left++;

        /* Decrement right index while we see 1 at right */
        while (arr[right] == 1 && left < right)
            right--;
```

```
        /* If left is smaller than right then there is a 1 at
left
        and a 0 at right. Exchange arr[left] and arr[right]*/
        if (left < right)
        {
            arr[left] = 0;
            arr[right] = 1;
            left++;
            right--;
        }
    }
}

/* Driver code */
int main()
{
    int arr[] = {0, 1, 0, 1, 1, 1};
    int i, arr_size = sizeof(arr)/sizeof(arr[0]);

    segregate0and1(arr, arr_size);

    cout << "Array after segregation ";
    for (i = 0; i < 6; i++)
        cout << arr[i] << " ";
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
Array after segregation is 0 0 1 1 1 1
```

Time Complexity: O(n)

Another approach :

1. Take two pointer type0(for element 0) starting from beginning (index = 0) and type1(for element 1) starting from end (index = array.length-1). Initialize type0 = 0 and type1 = array.length-1
2. It is intended to Put 1 to the right side of the array. Once it is done, then 0 will definitely towards left side of array.

```
// C++ program to sort a
// binary array in one pass
#include <bits/stdc++.h>
using namespace std;

/*Function to put all 0s on
left and all 1s on right*/
void segregate0and1(int arr[],
                    int size)
```

```
{
    int type0 = 0;
    int type1 = size - 1;

    while(type0 < type1)
    {
        if(arr[type0] == 1)
        {
            swap(arr[type0],
                 arr[type1]);
            type1--;
        }
        else
        type0++;
    }
}

// Driver Code
int main()
{
    int arr[] = {0, 1, 0, 1, 1, 1};
    int i, arr_size = sizeof(arr) /
                      sizeof(arr[0]);

    segregate0and1(arr, arr_size);

    cout << "Array after segregation is ";
    for (i = 0; i < arr_size; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output:

```
Array after segregation is 0 0 1 1 1 1
```

Time complexity: O(n)

8. Two elements whose sum is closest to zero

Question: An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.
For the below array, program should print -80 and 85.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

METHOD 1 (Simple)
For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

Implementation:

- C++

```
# include <bits/stdc++.h>
# include <stdlib.h> /* for abs() */
# include <math.h>

using namespace std;
void minAbsSumPair(int arr[], int arr_size)
{
    int inv_count = 0;
    int l, r, min_sum, sum, min_l, min_r;

    /* Array should have at least
       two elements*/
    if(arr_size < 2)
    {
        cout << "Invalid Input";
        return;
    }

    /* Initialization of values */
    min_l = 0;
    min_r = 1;
    min_sum = arr[0] + arr[1];

    for(l = 0; l < arr_size - 1; l++)
    {
        for(r = l + 1; r < arr_size; r++)
        {
        sum = arr[l] + arr[r];
        if(abs(min_sum) > abs(sum))
        {
            min_sum = sum;
            min_l = l;
            min_r = r;
```

```
        }
        }
    }

    cout << "The two elements whose sum is minimum are "
         << arr[min_l] << " and " << arr[min_r];
}

// Driver Code
int main()
{
    int arr[] = {1, 60, -10, 70, -80, 85};
    minAbsSumPair(arr, 6);
    return 0;
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

Output:

```
The two elements whose sum is minimum are -80 and 85
```

Time complexity: O(n^2)

METHOD 2 (Use Sorting)
Thanks to baskin for suggesting this approach. We recommend to read this post for background of this approach.
Algorithm
1) Sort all the elements of the input array.
2) Use two index variables l and r to traverse from left and right ends respectively. Initialize l as 0 and r as n-1.
3) sum = a[l] + a[r]
4) If sum is -ve, then l++
5) If sum is +ve, then r–
6) Keep track of abs min sum.
7) Repeat steps 3, 4, 5 and 6 while l < r
Implementation

```
#include <bits/stdc++.h>
using namespace std;

void quickSort(int *, int, int);

/* Function to print pair of elements
   having minimum sum */
void minAbsSumPair(int arr[], int n)
{

    // Variables to keep track
    // of current sum and minimum sum
```

```
    int sum, min_sum = INT_MAX;

    // left and right index variables
    int l = 0, r = n-1;

    // variable to keep track of
    // the left and right pair for min_sum
    int min_l = l, min_r = n-1;

    /* Array should have at least two elements*/
    if(n < 2)
    {
        cout << "Invalid Input";
        return;
    }

    /* Sort the elements */
    quickSort(arr, l, r);

    while(l < r)
    {
        sum = arr[l] + arr[r];

        /*If abs(sum) is less
          then update the result items*/
        if(abs(sum) < abs(min_sum))
        {
            min_sum = sum;
            min_l = l;
            min_r = r;
        }
        if(sum < 0)
            l++;
        else
            r--;
    }

    cout << "The two elements whose sum is minimum are "
         << arr[min_l] << " and " << arr[min_r];
}

// Driver Code
int main()
{
    int arr[] = {1, 60, -10, 70, -80, 85};
    int n = sizeof(arr) / sizeof(arr[0]);
    minAbsSumPair(arr, n);
    return 0;
}
```

```
/* FOLLOWING FUNCTIONS ARE ONLY FOR
   SORTING PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int si, int ei)
{
    int x = arr[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(arr[j] <= x)
        {
            i++;
            exchange(&arr[i], &arr[j]);
        }
    }
    exchange (&arr[i + 1], &arr[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
arr[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int arr[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(arr, si, ei);
        quickSort(arr, si, pi - 1);
        quickSort(arr, pi + 1, ei);
    }
}

// This code is contributed by rathbhupendra
```

Output:

```
The two elements whose sum is minimum are -80 and 85
```

Time Complexity: complexity to sort + complexity of finding the optimum pair = O(nlogn) + O(n) = O(nlogn)

STL implementation of Method-2:

Algorithm
1) Sort all the elements of the input array using their absolute values.
2) Check absolute sum of arr[i-1] and arr[i] if their absolute sum is less than min update min with their absolute value.
3) Use two variables to store the index of the elements.
Implementation

```
// C++ implementation using STL
#include <bits/stdc++.h>
using namespace std;

// Modified to sort by abolute values
bool compare(int x, int y)
{
    return abs(x) < abs(y);
}

void findMinSum(int arr[], int n)
{
    sort(arr, arr + n, compare);
    int min = INT_MAX, x, y;
    for (int i = 1; i < n; i++) {

        // Absolute value shows how close it is to zero
        if (abs(arr[i - 1] + arr[i]) <= min) {

            // if found an even close value
            // update min and store the index
            min = abs(arr[i - 1] + arr[i]);
            x = i - 1;
            y = i;
        }
    }
    cout << "The two elements whose sum is minimum are "
         << arr[x] << " and " << arr[y];
}

// Driver code
int main()
{

    int arr[] = { 1, 60, -10, 70, -80, 85 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    findMinSum(arr, n);
    return 0;
    // This code is contributed by ceeyesharish
}
```

Output:

```
The two elements whose sum is minimum are -80 and 85
```

Time Complexity: O(nlogn)
Space Complexity: O(1

10. Find a triplet that sum to a given value

Given an array and a value, find if there is a triplet in array whose sum is equal to the given value. If there is such a triplet present in array, then print the triplet and return true. Else return false. For example, if the given array is {12, 3, 4, 1, 6, 9} and given sum is 24, then there is a triplet (12, 3 and 9) present in array whose sum is 24.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Naive)
A simple method is to generate all possible triplets and compare the sum of every triplet with the given value. The following code implements this simple method using three nested loops.

```
#include <bits/stdc++.h>
using namespace std;

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet
bool find3Numbers(int A[], int arr_size, int sum)
{
    int l, r;

    // Fix the first element as A[i]
    for (int i = 0; i < arr_size - 2; i++)
    {

        // Fix the second element as A[j]
        for (int j = i + 1; j < arr_size - 1; j++)
        {

            // Now look for the third number
            for (int k = j + 1; k < arr_size; k++)
            {
                if (A[i] + A[j] + A[k] == sum)
                {
                    cout << "Triplet is " << A[i] <<
                        ", " << A[j] << ", " << A[k];
                    return true;
                }
            }
        }
    }

    // If we reach here, then no triplet was found
    return false;
```

```
}

/* Driver code */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int sum = 22;
    int arr_size = sizeof(A) / sizeof(A[0]);
    find3Numbers(A, arr_size, sum);
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output :

```
Triplet is 4, 10, 8
```

Time Complexity : O(n^3)

Method 2 (Use Sorting)
Time complexity of the method 1 is O(n^3). The complexity can be reduced to O(n^2) by sorting the array first, and then using method 1 of this post in a loop.
1) Sort the input array.
2) Fix the first element as A[i] where i is from 0 to array size – 2. After fixing the first element of triplet, find the other two elements using method 1 of this post.

```
// C++ program to find a triplet
#include <bits/stdc++.h>
using namespace std;

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet
bool find3Numbers(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    sort(A, A + arr_size);

    /* Now fix the first element one by one and find the
       other two elements */
    for (int i = 0; i < arr_size - 2; i++) {

        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        l = i + 1; // index of the first element in the
        // remaining elements
```

```
        r = arr_size - 1; // index of the last element
        while (l < r) {
            if (A[i] + A[l] + A[r] == sum) {
                printf("Triplet is %d, %d, %d", A[i],
                       A[l], A[r]);
                return true;
            }
            else if (A[i] + A[l] + A[r] < sum)
                l++;
            else // A[i] + A[l] + A[r] > sum
                r--;
        }
    }

    // If we reach here, then no triplet was found
    return false;
}

/* Driver program to test above function */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int sum = 22;
    int arr_size = sizeof(A) / sizeof(A[0]);

    find3Numbers(A, arr_size, sum);

    return 0;
}
```

Output :

```
Triplet is 4, 8, 10
```

Time Complexity : O(n^2)
Method 3 (Hashing Based Solution)

```
// C++ program to find a triplet using Hashing
#include <bits/stdc++.h>
using namespace std;

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet
bool find3Numbers(int A[], int arr_size, int sum)
{
    // Fix the first element as A[i]
    for (int i = 0; i < arr_size - 2; i++) {

        // Find pair in subarray A[i+1..n-1]
```

```
        // with sum equal to sum - A[i]
        unordered_set<int> s;
        int curr_sum = sum - A[i];
        for (int j = i + 1; j < arr_size; j++) {
            if (s.find(curr_sum - A[j]) != s.end()) {
                printf("Triplet is %d, %d, %d", A[i],
                       A[j], curr_sum - A[j]);
                return true;
            }
            s.insert(A[j]);
        }
    }

    // If we reach here, then no triplet was found
    return false;
}

/* Driver program to test above function */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int sum = 22;
    int arr_size = sizeof(A) / sizeof(A[0]);

    find3Numbers(A, arr_size, sum);

    return 0;
}
```

Output :

```
Triplet is 4, 8, 10
```

11. Equilibrium index of an array

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an array A:

Example :

Input: A[] = {-7, 1, 5, 2, -4, 3, 0}
Output: 3
3 is an equilibrium index, because:
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]

Input: A[] = {1, 2, 3}
Output: -1

Write a function int equilibrium(int[] arr, int n); that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple but inefficient)
Use two loops. Outer loop iterates through all the element and inner loop finds out whether the current index picked by the outer loop is equilibrium index or not. Time complexity of this solution is O(n^2).

```
// C++ program to find equilibrium
// index of an array
#include <bits/stdc++.h>
using namespace std;

int equilibrium(int arr[], int n)
{
    int i, j;
    int leftsum, rightsum;

    /* Check for indexes one by one until
    an equilibrium index is found */
    for (i = 0; i < n; ++i)
    {

        /* get left sum */
        leftsum = 0;
        for (j = 0; j < i; j++)
            leftsum += arr[j];

        /* get right sum */
        rightsum = 0;
        for (j = i + 1; j < n; j++)
            rightsum += arr[j];
```

```
        /* if leftsum and rightsum
        are same, then we are done */
        if (leftsum == rightsum)
            return i;
    }

    /* return -1 if no equilibrium
    index is found */
    return -1;
}

// Driver code
int main()
{
    int arr[] = { -7, 1, 5, 2, -4, 3, 0 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    cout << equilibrium(arr, arr_size);
    return 0;
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

Time Complexity: O(n^2)
Method 2 (Tricky and Efficient)
The idea is to get the total sum of the array first. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get the right sum by subtracting the elements one by one. Thanks to Sambasiva for suggesting this solution and providing code for this.

```
1) Initialize leftsum  as 0

2) Get the total sum of the array as sum
3) Iterate through the array and for each index i, do following.

   a)  Update sum to get the right sum.
         sum = sum - arr[i]
      // sum is now right sum
   b) If leftsum is equal to sum, then return current index.
      // update leftsum for next iteration.

   c) leftsum = leftsum + arr[i]

4) return -1

// If we come out of loop without returning then

// there is no equilibrium index
```

The image below shows the dry run of the above approach:

Initially:

Sum= 0

Step 1:

Sum = +7
leftSum = -7

Step 2:

Sum = 6
leftSum = -6

Step 3:

Sum = 1
leftSum = -1

Step 4:

Sum = -1
if(true)
↳ index found

Below is the implementation of the above approach:

```
// C++ program to find equilibrium
// index of an array
#include <bits/stdc++.h>
using namespace std;

int equilibrium(int arr[], int n)
{
    int sum = 0; // initialize sum of whole array
    int leftsum = 0; // initialize leftsum

    /* Find sum of the whole array */
    for (int i = 0; i < n; ++i)
        sum += arr[i];

    for (int i = 0; i < n; ++i)
    {
        sum -= arr[i]; // sum is now right sum for index i

        if (leftsum == sum)
            return i;

        leftsum += arr[i];
    }

    /* If no equilibrium index found, then return 0 */
    return -1;
}

// Driver code
int main()
{
    int arr[] = { -7, 1, 5, 2, -4, 3, 0 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    cout << "First equilibrium index is " << equilibrium(arr,
arr_size);
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
First equilibrium index is 3
```

Time Complexity: O(n)

12. Find the index of first 1 in an infinite sorted array of 0s and 1s
Given an infinite sorted array consisting 0s and 1s. The problem is to find the index of first '1' in that array. As the array is infinite, therefore it is guaranteed that number '1' will be present in the array.

Examples:

```
Input : arr[] = {0, 0, 1, 1, 1, 1}

Output : 2


Input : arr[] = {1, 1, 1, 1,, 1, 1}

Output : 0
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

Approach: The problem is closely related to the problem of finding position of an element in a sorted array of infinite numbers. As the array is infinte, therefore we do not know the upper and lower bounds between which we have to find the occurrence of first '1'. Below is an algorithm to find the upper and lower bounds.

Algorithm:

```
posOfFirstOne(arr)
    Declare l = 0, h = 1
    while arr[h] == 0
        l = h

    h = 2*h;

    return indexOfFirstOne(arr, l, h)
}
```

Here **h** and **l** are the required upper and lower bounds. **indexOfFirstOne(arr, l, h)** is used to find the index of occurrence of first '1' between these two bounds. Refer this post.

```
// C++ implementation to find the index of first 1
// in an infinite sorted array of 0's and 1's
#include <bits/stdc++.h>
using namespace std;

// function to find the index of first '1'
// binary search technique is applied
int indexOfFirstOne(int arr[], int low, int high)
{
    int mid;
    while (low <= high) {
        mid = (low + high) / 2;

        // if true, then 'mid' is the index of first '1'
        if (arr[mid] == 1 &&
            (mid == 0 || arr[mid - 1] == 0))
```

```
            break;

        // first '1' lies to the left of 'mid'
        else if (arr[mid] == 1)
            high = mid - 1;

        // first '1' lies to the right of 'mid'
        else
            low = mid + 1;
    }

    // required index
    return mid;
}

// function to find the index of first 1 in
// an infinite sorted array of 0's and 1's
int posOfFirstOne(int arr[])
{
    // find the upper and lower bounds between
    // which the first '1' would be present
    int l = 0, h = 1;

    // as the array is being considered infinite
    // therefore 'h' index will always exist in
    // the array
    while (arr[h] == 0) {

        // lower bound
        l = h;

        // upper bound
        h = 2 * h;
    }

    // required index of first '1'
    return indexOfFirstOne(arr, l, h);
}

// Driver program to test above
int main()
{
    int arr[] = { 0, 0, 1, 1, 1, 1 };
    cout << "Index = "
         << posOfFirstOne(arr);
    return 0;
}
```

Output:

```
Index = 2
```

Let p be the position of element to be searched. Number of steps for finding high index ‘h’ is O(Log p). The value of ‘h’ must be less than 2*p. The number of elements between h/2 and h must be O(p). Therefore, time complexity of Binary Search step is also O(Log p) and overall time complexity is 2*O(Log p) which is O(Log p).

13. Sliding Window Maximum (Maximum of all subarrays of size k)

Given an array and an integer K, find the maximum for each and every contiguous subarray of size k.
Examples :

Input: arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, K = 3
Output: 3 3 4 5 5 5 6

Input: arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}, K = 4
Output: 10 10 10 15 15 90 90

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple)
Run two loops. In the outer loop, take all subarrays of size K. In the inner loop, get the maximum of the current subarray.

```
// C++ Program to find the maximum for
// each and every contiguous subarray of size k.
#include <bits/stdc++.h>
using namespace std;

// Method to find the maximum for each
// and every contiguous subarray of size k.
void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n - k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i + j] > max)
                max = arr[i + j];
        }
        cout << max << " ";
    }
}

// Driver code
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
3 4 5 6 7 8 9 10
```

Time Complexity : The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(N * K).

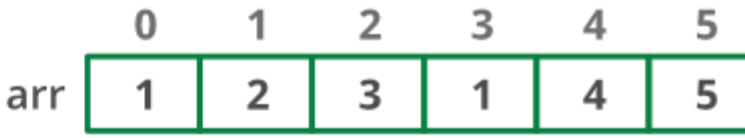
Method 2 (Use Self-Balancing BST)

- Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
- Run a loop for i = 0 to n – k
    1. Get the maximum element from the BST, and print it.
    2. Search for arr[i] in the BST and delete it from the BST.
    3. Insert arr[i+k] into the BST.

Time Complexity: Time Complexity of step 1 is O(K * Log k). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). Since steps 2(a), 2(b) and 2(c) are in a loop that runs n-k+1 times, time complexity of the complete algorithm is O(kLogk + (n-k+1)*Logk) which can also be written as O(N * Log k).

Method 3 (A O(n) method: use Deque) We create a Deque, Qi of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Qi to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and element at rear of Qi is the smallest of current window. Thanks to Aashish for suggesting this method.
Below image is a dry run of the above approach:

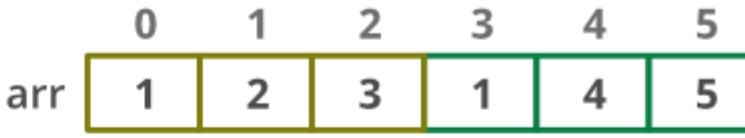
Initially :



Dequeue stores index of the maximu
and stores index of minimum eleme
time, it store indexes which belongs

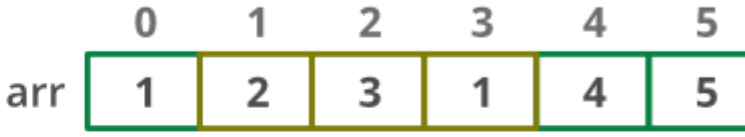
Step 1 :

Take the first window and keep nece



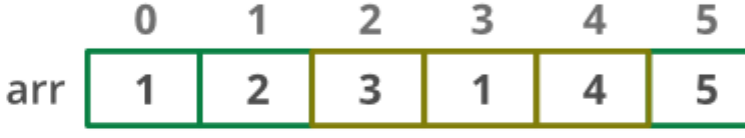
dequeue : { 2 }
Maximum element is arr[2] = 3

Step 2 :



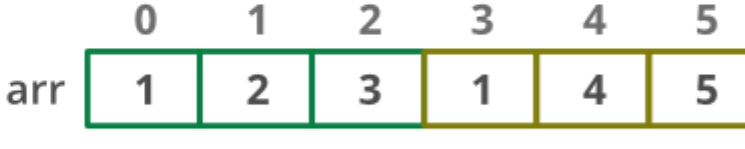
dequeue : { 2,3 }
Maximum element is arr[2] = 3

Step 3 :



dequeue : { 4 }
Maximum element is arr[4] = 4

Step 4 :



dequeue : { 5 }

Below is the implementation of the above approach.

- C++
- Java
- Python3
- C#

filter_none

edit

play_arrow

brightness_4

```
#include <deque>
#include <iostream>

using namespace std;

// A Dequeue (Double ended queue) based method for printing
maximum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of
array elements
    // The queue will store indexes of useful elements in every
window and it will
    // maintain decreasing order of values from front to rear in
Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing
order
    std::deque<int> Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i) {
        // For every element, the previous smaller elements are
useless so
        // remove them from Qi
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back(); // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for (; i < n; ++i) {
```

```
        // The element at the front of the queue is the largest
element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ((!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front(); // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

        // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = { 12, 1, 78, 90, 57, 89, 56 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Output:

```
78 90 90 90 89
```

Output:

```
78 90 90 90 89
```

Below is an extension of this problem.
Sum of minimum and maximum elements of all subarrays of size k.
Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total 2n operations.
Auxiliary Space: O(k)
Method 4 (Use Max-Heap)

1. Pick first k elements and create a max heap of size k.
2. Perform heapify and print the root element.
3. Store the next and last element from the array
4. Run a loop from k – 1 to n
    - Replace the value of element which is got out of the window with new element which came inside the window.
    - Perform heapify.

- Print the root of the Heap.

Time Complexity: Time Complexity of steps 4(a) is O(k), 4(b) is O(Log(k)) and it is in a loop that runs (n – k + 1) times. Hence, the time complexity of the complete algorithm is O((k + Log(k)) * n) i.e. O(n * k).

```
# Python program to find the maximum for
# each and every contiguous subarray of
# size k
import heapq

# Method to find the maximum for each
# and every contiguous subarray of s
# of size k
def max_of_all_in_k(arr, n):
    i = 0
    j = k-1

    # Create the heap and heapify
    heap = arr[i:j + 1]
    heapq._heapify_max(heap)

    # Print the maximum element from
    # the first window of size k
    print(heap[0], end =" ")
    last = arr[i]
    i+= 1
    j+= 1
    nexts = arr[j]

    # For every remaining element
    while j < n:

        # Add the next element of the window
        heap[heap.index(last)] = nexts

        # Heapify to get the maximum
        # of the current window
        heapq._heapify_max(heap)

        # Print the current maximum
        print(heap[0], end =" ")
        last = arr[i]
        i+= 1
        j+= 1
        if j < n:
            nexts = arr[j]

# Driver Function
```

```
n, k = 10, 3
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
max_of_all_in_k(arr, n)
```

Output:

```
3 4 5 6 7 8 9 10
```

14. Count smaller elements on right side

Write a function to count number of smaller elements on right of each element in an array. Given an unsorted array arr[] of distinct integers, construct another array countSmaller[] such that countSmaller[i] contains count of smaller elements on right side of each element arr[i] in array.

Examples:

```
Input:   arr[] =  {12, 1, 2, 3, 0, 11, 4}
Output:  countSmaller[]  =  {6, 1, 1, 1, 0, 1, 0}

(Corner Cases)
Input:   arr[] =  {5, 4, 3, 2, 1}
Output:  countSmaller[]  =  {4, 3, 2, 1, 0}

Input:   arr[] =  {1, 2, 3, 4, 5}
Output:  countSmaller[]  =  {0, 0, 0, 0, 0}
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Simple)
Use two loops. The outer loop picks all elements from left to right. The inner loop iterates through all the elements on right side of the picked element and updates countSmaller[].

```
void constructLowerArray (int *arr[], int *countSmaller, int n)
{
  int i, j;

  // initialize all the counts in countSmaller array as 0
  for  (i = 0; i < n; i++)
     countSmaller[i] = 0;

  for (i = 0; i < n; i++)
  {
    for (j = i+1; j < n; j++)
    {
       if (arr[j] < arr[i])
         countSmaller[i]++;
    }
  }
```

```
}

/* Utility function that prints out an array on a line */
void printArray(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);

  printf("\n");
}

// Driver program to test above functions
int main()
{
  int arr[] = {12, 10, 5, 4, 2, 20, 6, 1, 0, 2};
  int n = sizeof(arr)/sizeof(arr[0]);
  int *low = (int *)malloc(sizeof(int)*n);
  constructLowerArray(arr, low, n);
  printArray(low, n);
  return 0;
}
```

Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Use Self Balancing BST)
A Self Balancing Binary Search Tree (AVL, Red Black,.. etc) can be used to get the solution in O(nLogn) time complexity. We can augment these trees so that every node N contains size the subtree rooted with N. We have used AVL tree in the following implementation.
We traverse the array from right to left and insert all elements one by one in an AVL tree. While inserting a new key in an AVL tree, we first compare the key with root. If key is greater than root, then it is greater than all the nodes in left subtree of root. So we add the size of left subtree to the count of smaller element for the key being inserted. We recursively follow the same approach for all nodes down the root.

Following is C implementation.

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
```

```
    int size; // size of the tree rooted with this node
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree rooted with N
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to size of the tree of rooted with N
int size(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->size;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key
and
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->key   = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1;  // new node is initially added at leaf
    node->size = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
```

```
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Update sizes
    y->size = size(y->left) + size(y->right) + 1;
    x->size = size(x->left) + size(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Update sizes
    x->size = size(x->left) + size(x->right) + 1;
    y->size = size(y->left) + size(y->right) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Inserts a new key to the tree rotted with node. Also, updates
*count
// to contain count of smaller elements for the new key
struct node* insert(struct node* node, int key, int *count)
{
```

```
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key, count);
    else
    {
        node->right = insert(node->right, key, count);

        // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
        *count = *count + size(node->left) + 1;
    }


    /* 2. Update height and size of this ancestor node */
    node->height = max(height(node->left), height(node->right)) +
1;
    node->size   = size(node->left) + size(node->right) + 1;

    /* 3. Get the balance factor of this ancestor node to check
whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
```

```
    /* return the (unchanged) node pointer */
    return node;
}

// The following function updates the countSmaller array to
contain count of
// smaller elements on right side.
void constructLowerArray (int arr[], int countSmaller[], int n)
{
  int i, j;
  struct node *root = NULL;

  // initialize all the counts in countSmaller array as 0
  for  (i = 0; i < n; i++)
     countSmaller[i] = 0;

  // Starting from rightmost element, insert all elements one by
one in
  // an AVL tree and get the count of smaller elements
  for (i = n-1; i >= 0; i--)
  {
     root = insert(root, arr[i], &countSmaller[i]);
  }
}

/* Utility function that prints out an array on a line */
void printArray(int arr[], int size)
{
  int i;
  printf("\n");
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);
}

// Driver program to test above functions
int main()
{
  int arr[] = {10, 6, 15, 20, 30, 5, 7};
  int n = sizeof(arr)/sizeof(arr[0]);

  int *low = (int *)malloc(sizeof(int)*n);

  constructLowerArray(arr, low, n);

  printf("Following is the constructed smaller count array");
  printArray(low, n);
  return 0;
}
```

Output:

```
Following is the constructed smaller count array

3 1 2 2 2 0 0
```

Time Complexity: O(nLogn)
Auxiliary Space: O(n)

14. Find the largest multiple of 3 | Set 1 (Using Queue)

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be “9 8 1”, and if the input array is {8, 1, 7, 6, 0}, output should be “8 7 6 0”.

Method 1 (Brute Force)
The simple & straight forward approach is to generate all the combinations of the elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: O(n x 2^n). There will be 2^n combinations of array elements. To compare each combination with the largest number so far may take O(n) time.
Auxiliary Space: O(n) // to avoid integer overflow, the largest number is assumed to be stored in the form of array.

Method 2 (Tricky)
This problem can be solved efficiently with the help of O(n) extra space. This method is based on the following facts about numbers which are multiple of 3.
1) A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is 8 + 7+ 6+ 0 = 21, which is a multiple of 3.
2) If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, ….. are also multiples of 3.
3) We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same remainder 1.
What is the idea behind above facts?
The value of 10%3 and 100%3 is 1. The same is true for all the higher powers of 10, because 3 divides 9, 99, 999, … etc.
Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be ‘a’, ‘b’ and ‘c’ respectively. n can be written as

```
n = 100.a + 10.b + c
```

Since (10^x)%3 is 1 for any x, the above expression gives the same remainder as following expression

```
 1.a + 1.b + c
```

So the remainder obtained by sum of digits and ‘n’ is same.

Following is a solution based on the above observation.

1. Sort the array in non-decreasing order.

2. Take three queues. One for storing elements which on dividing by 3 gives remainder as 0.The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2

3. Find the sum of all the digits.
4. Three cases arise:
……4.1 The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.
……4.2 The sum of digits produces remainder 1 when divided by 3.
Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.
……4.3 The sum of digits produces remainder 2 when divided by 3.
Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.
5. Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.
The below code works only if the input arrays has numbers from 0 to 9. It can be easily extended for any positive integer array. We just have to modify the part where we sort the array in decreasing order, at the end of code.

```
// C++ implementation of the above approach
#include <bits/stdc++.h>
using namespace std;

// This function puts all elements of 3 queues in the auxiliary
array
void populateAux(int aux[], queue<int> queue0, queue<int> queue1,
                 queue<int> queue2, int* top)
{
    // Put all items of first queue in aux[]
    while (!queue0.empty()) {
        aux[(*top)++] = queue0.front();
        queue0.pop();
    }

    // Put all items of second queue in aux[]
    while (!queue1.empty()) {
        aux[(*top)++] = queue1.front();
        queue1.pop();
    }

    // Put all items of third queue in aux[]
    while (!queue2.empty()) {
        aux[(*top)++] = queue2.front();
        queue2.pop();
    }
}

// The main function that finds the largest possible multiple of
// 3 that can be formed by arr[] elements
int findMaxMultupleOf3(int arr[], int size)
{
    // Step 1: sort the array in non-decreasing order
```

```
    sort(arr, arr + size);

    // Create 3 queues to store numbers with remainder 0, 1
    // and 2 respectively
    queue<int> queue0, queue1, queue2;

    // Step 2 and 3 get the sum of numbers and place them in
    // corresponding queues
    int i, sum;
    for (i = 0, sum = 0; i < size; ++i) {
        sum += arr[i];
        if ((arr[i] % 3) == 0)
            queue0.push(arr[i]);
        else if ((arr[i] % 3) == 1)
            queue1.push(arr[i]);
        else
            queue2.push(arr[i]);
    }

    // Step 4.2: The sum produces remainder 1
    if ((sum % 3) == 1) {
        // either remove one item from queue1
        if (!queue1.empty())
            queue1.pop();

        // or remove two items from queue2
        else {
            if (!queue2.empty())
                queue2.pop();
            else
                return 0;

            if (!queue2.empty())
                queue2.pop();
            else
                return 0;
        }
    }

    // Step 4.3: The sum produces remainder 2
    else if ((sum % 3) == 2) {
        // either remove one item from queue2
        if (!queue2.empty())
            queue2.pop();

        // or remove two items from queue1
        else {
            if (!queue1.empty())
                queue1.pop();
```

```
            else
                return 0;

            if (!queue1.empty())
                queue1.pop();
            else
                return 0;
        }
    }

    int aux[size], top = 0;

    // Empty all the queues into an auxiliary array.
    populateAux(aux, queue0, queue1, queue2, &top);

    // sort the array in non-increasing order
    sort(aux, aux + top, greater<int>());

    // print the result
    for (int i = 0; i < top; ++i)
        cout << aux[i] << " ";

    return top;
}
int main()
{

    int arr[] = { 8, 1, 7, 6, 0 };
    int size = sizeof(arr) / sizeof(arr[0]);

    if (findMaxMultupleOf3(arr, size) == 0)
        cout << "Not Possible";

    return 0;
}
```

Output:

```
8 7 6 0
```

The above method can be optimized in following ways.
1) We can use Heap Sort or Merge Sort to make the time complexity O(nLogn).

2) We can avoid extra space for queues. We know at most two items will be removed from the input array. So we can keep track of two items in two variables.

3) At the end, instead of sorting the array again in descending order, we can print the ascending sorted array in reverse order. While printing in reverse order, we can skip the two elements to be removed.

Time Complexity: O(nLogn), assuming a O(nLogn) algorithm is used for sorting.

15. Find the largest multiple of 3 from array of digits | Set 2 (In O(n) time and O(1) space)

Given an array of digits (contain elements from 0 to 9). Find the largest number that can be made from some or all digits of array and is divisible by 3. The same element may appear multiple times in the array, but each element in the array may only be used once.
Examples:

```
Input : arr[] = {5, 4, 3, 1, 1}

Output : 4311


Input : Arr[] = {5, 5, 5, 7}

Output : 555
```

Asked In : Google Interview

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

We have discussed a queue based solution. Both solutions (discussed in previous and this posts) are based on the fact that a number is divisible by 3 if and only if sum of digits of the number is divisible by 3.
For example, let us consider 555, it is divisible by 3 because sum of digits is 5 + 5 + 5 = 15, which is divisible by 3. If a sum of digits is not divisible by 3 then the remainder should be either 1 or 2.
If we get remainder either '1' or '2', we have to remove maximum two digits to make a number that is divisible by 3:

1. If remainder is '1' : We have to remove single digit that have remainder '1' or we have to remove two digit that have remainder '2' (2 + 2 => 4 % 3 => '1')
2. If remainder is '2' : .We have to remove single digit that have remainder '2' or we have to remove two digit that have remainder '1' (1 + 1 => 2 % 3 => 2).

Examples :

```
Input : arr[] = 5, 5, 5, 7

Sum of digits = 5 + 5 + 7 = 22

Remainder = 22 % 3 = 1

We remove smallest single digit that

has remainder '1'. We remove 7 % 3 = 1

So largest number divisible by 3 is : 555


Let's take an another example :

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

Sum of digits  = 4 + 4 + 1 + 1 + 1 + 3 = 14
```

```
Reminder = 14 % 3 = 2
We have to remove the smallest digit that
has remainder ' 2 ' or two digits that have
remainder '1'. Here there is no digit with
reminder '2', so we have to remove two smallest
digits that have remainder '1'. The digits are :
1, 1. So largest number divisible by 3 is 4 4 3 1
```

Below are implementation of above idea.

```
// C++ program to find the largest number
// that can be mode from elements of the
// array and is divisible by 3
#include<bits/stdc++.h>
using namespace std;

// Number of digits
#define MAX_SIZE 10

// function to sort array of digits using
// counts
void sortArrayUsingCounts(int arr[], int n)
{
    // Store count of all elements
    int count[MAX_SIZE] = {0};
    for (int i = 0; i < n; i++)
        count[arr[i]]++;

    // Store
    int index = 0;
    for (int i = 0; i < MAX_SIZE; i++)
        while (count[i] > 0)
            arr[index++] = i, count[i]--;
}

// Remove elements from arr[] at indexes ind1 and ind2
bool removeAndPrintResult(int arr[], int n, int ind1,
                                          int ind2 = -1)
{
    for (int i = n-1; i >=0; i--)
        if (i != ind1 && i != ind2)
            cout << arr[i] ;
}

// Returns largest multiple of 3 that can be formed
// using arr[] elements.
bool largest3Multiple(int arr[], int n)
```

```
{
    // Sum of all array element
    int sum = accumulate(arr, arr+n, 0);

    // Sum is divisible by 3 , no need to
    // delete an element
    if (sum%3 == 0)
        return true ;

    // Sort array element in increasing order
    sortArrayUsingCounts(arr, n);

    // Find reminder
    int remainder = sum % 3;

    // If remainder is '1', we have to delete either
    // one element of remainder '1' or two elements
    // of remainder '2'
    if (remainder == 1)
    {
        int rem_2[2];
        rem_2[0] = -1, rem_2[1] = -1;

        // Traverse array elements
        for (int i = 0 ; i < n ; i++)
        {
            // Store first element of remainder '1'
            if (arr[i]%3 == 1)
            {
                removeAndPrintResult(arr, n, i);
                return true;
            }

            if (arr[i]%3 == 2)
            {
                // If this is first occurrence of remainder 2
                if (rem_2[0] == -1)
                    rem_2[0] = i;

                // If second occurrence
                else if (rem_2[1] == -1)
                    rem_2[1] = i;
            }
        }

        if (rem_2[0] != -1 && rem_2[1] != -1)
        {
            removeAndPrintResult(arr, n, rem_2[0], rem_2[1]);
            return true;
```

```
        }
    }

    // If remainder is '2', we have to delete either
    // one element of remainder '2' or two elements
    // of remainder '1'
    else if (remainder == 2)
    {
        int rem_1[2];
        rem_1[0] = -1, rem_1[1] = -1;

        // traverse array elements
        for (int i = 0; i < n; i++)
        {
            // store first element of remainder '2'
            if (arr[i]%3 == 2)
            {
                removeAndPrintResult(arr, n, i);
                return true;
            }

            if (arr[i]%3 == 1)
            {
                // If this is first occurrence of remainder 1
                if (rem_1[0] == -1)
                    rem_1[0] = i;

                // If second occurrence
                else if (rem_1[1] == -1)
                    rem_1[1] = i;
            }
        }

        if (rem_1[0] != -1 && rem_1[1] != -1)
        {
            removeAndPrintResult(arr, n, rem_1[0], rem_1[1]);
            return true;
        }
    }

    cout << "Not possible";
    return false;
}

// Driver code
int main()
{
    int arr[] = {4 , 4 , 1 , 1 , 1 , 3} ;
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    largest3Multiple(arr, n);
    return 0;
}
```

Output:

```
4431
```

Time Complexity : O(n)

16. Find subarray with given sum | Set 1 (Nonnegative Numbers)

Given an unsorted array of nonnegative integers, find a continuous subarray which adds to a given number.

Examples :

Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Ouptut: Sum found between indexes 2 and 4

Input: arr[] = {1, 4, 0, 0, 3, 10, 5}, sum = 7
Ouptut: Sum found between indexes 1 and 4
Input: arr[] = {1, 4}, sum = 0
Output: No subarray found

There may be more than one subarrays with sum as the given sum. The following solutions print first such subarray.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple)
A simple solution is to consider all subarrays one by one and check the sum of every subarray. Following program implements the simple solution. We run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i.
Following is the implementation of the above approach.

```
/* A simple program to print subarray
with sum as given sum */
#include <bits/stdc++.h>
using namespace std;

/* Returns true if the there is a subarray
of arr[] with sum equal to 'sum' otherwise
returns false. Also, prints the result */
int subArraySum(int arr[], int n, int sum)
{
    int curr_sum, i, j;

    // Pick a starting point
    for (i = 0; i < n; i++)
    {
        curr_sum = arr[i];

        // try all subarrays starting with 'i'
        for (j = i + 1; j <= n; j++)
        {
            if (curr_sum == sum)
```

```
            {
                cout << "Sum found between indexes "
                     << i << " and " << j - 1;
                return 1;
            }
            if (curr_sum > sum || j == n)
                break;
        curr_sum = curr_sum + arr[j];
        }
    }

    cout << "No subarray found";
    return 0;
}

// Driver Code
int main()
{
    int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};
    int n = sizeof(arr) / sizeof(arr[0]);
    int sum = 23;
    subArraySum(arr, n, sum);
    return 0;
}

// This code is contributed
// by rathbhupendra
```

Output :

```
Sum found between indexes 1 and 4
```

Time Complexity : O(n^2) in worst case.

Method 2 (Efficient)
Initialize a variable curr_sum as the first element. curr_sum indicates the sum of the current subarray. Start from the second element and add all elements one by one to the curr_sum. If curr_sum becomes equal to the sum, then print the solution. If curr_sum exceeds the sum, then remove trailing elements while curr_sum is greater than the sum.
Following is the implementation of the above approach.

```
/* An efficient program to print
subarray with sum as given sum */
#include <iostream>
using namespace std;

/* Returns true if the there is a subarray of
arr[] with a sum equal to 'sum' otherwise
```

```
returns false. Also, prints the result */
int subArraySum(int arr[], int n, int sum)
{
    /* Initialize curr_sum as value of
    first element and starting point as 0 */
    int curr_sum = arr[0], start = 0, i;

    /* Add elements one by one to curr_sum and
    if the curr_sum exceeds the sum,
    then remove starting element */
    for (i = 1; i <= n; i++)
    {
        // If curr_sum exceeds the sum,
        // then remove the starting elements
        while (curr_sum > sum && start < i - 1)
        {
            curr_sum = curr_sum - arr[start];
            start++;
        }

        // If curr_sum becomes equal to sum,
        // then return true
        if (curr_sum == sum)
        {
            cout << "Sum found between indexes "
                 << start << " and " << i - 1;
            return 1;
        }

        // Add this element to curr_sum
        if (i < n)
        curr_sum = curr_sum + arr[i];
    }

    // If we reach here, then no subarray
    cout << "No subarray found";
    return 0;
}

// Driver Code
int main()
{
    int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};
    int n = sizeof(arr) / sizeof(arr[0]);
    int sum = 23;
    subArraySum(arr, n, sum);
    return 0;
}
```

```
// This code is contributed by SHUBHAMSINGH10
```

Output :

```
Sum found between indexes 1 and 4
```

The time complexity of method 2 looks more than O(n), but if we take a closer look at the program, then we can figure out the time complexity is O(n). We can prove it by counting the number of operations performed on every element of arr[] in the worst case. There are at most 2 operations performed on every element: (a) the element is added to the curr_sum (b) the element is subtracted from curr_sum. So the upper bound on the number of operations is 2n which is O(n).

17. Find subarray with given sum | Set 2 (Handles Negative Numbers)

Given an unsorted array of integers, find a subarray which adds to a given number. If there are more than one subarrays with the sum as the given number, print any of them.

Examples:

```
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Output: Sum found between indexes 2 and 4


Input: arr[] = {10, 2, -2, -20, 10}, sum = -10
Output: Sum found between indexes 0 to 3


Input: arr[] = {-10, 0, 2, -2, -20, 10}, sum = 20
Output: No subarray with given sum exists
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

We have discussed a solution that do not handles negative integers here. In this post, negative integers are also handled.

A simple solution is to consider all subarrays one by one and check if the sum of every subarray is equal to the given sum or not. The complexity of this solution would be O(n^2).

An efficient way is to use a map. The idea is to maintain the sum of elements encountered so far in a variable (say curr_sum). Let the given number is the sum. Now for each element, we check if curr_sum – sum exists in the map or not. If we found it in the map that means, we have a subarray present with the given sum, else we insert curr_sum into the map and proceed to the next element. If all elements of the array are processed and we didn't find any subarray with the given sum, then subarray doesn't exist.

Below image is a dry run of the above approach:

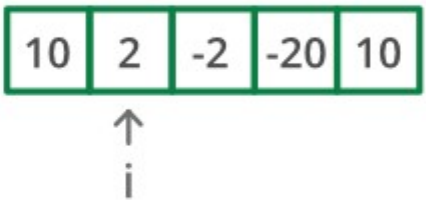
Step 0: curr-Sum = 0, Sum = -10, map→ emp

Step 1:

curr-sum = 10

map

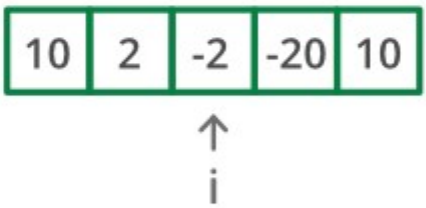
Step 2:



curr-sum = 12

map

Step 3:



curr-sum = 10

map

Step 4:

curr-sum = -10

if(true) → Sub array [10, 2, -2, 20]

Below is the implementation of the above approach :

```
// C++ program to print subarray with sum as given sum
#include<bits/stdc++.h>
using namespace std;

// Function to print subarray with sum as given sum
void subArraySum(int arr[], int n, int sum)
{
    // create an empty map
    unordered_map<int, int> map;

    // Maintains sum of elements so far
    int curr_sum = 0;

    for (int i = 0; i < n; i++)
    {
        // add current element to curr_sum
        curr_sum = curr_sum + arr[i];

        // if curr_sum is equal to target sum
        // we found a subarray starting from index 0
        // and ending at index i
        if (curr_sum == sum)
        {
            cout << "Sum found between indexes "
                 << 0 << " to " << i << endl;
            return;
        }

        // If curr_sum - sum already exists in map
        // we have found a subarray with target sum
        if (map.find(curr_sum - sum) != map.end())
        {
            cout << "Sum found between indexes "
                 << map[curr_sum - sum] + 1
                 << " to " << i << endl;
            return;
        }

        map[curr_sum] = i;
    }

    // If we reach here, then no subarray exists
    cout << "No subarray with given sum exists";
}

// Driver program to test above function
```

```
int main()
{
    int arr[] = {10, 2, -2, -20, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = -10;

    subArraySum(arr, n, sum);

    return 0;
}
```

Output:

```
Sum found between indexes 0 to 3
```

Time complexity of above solution is O(N) if we perform hashing with the help of an array. In case the elements cannot be hashed in an array we use a hash map as shown in the above code.

Auxiliary space used by the program is O(n).

18. Find if there is a subarray with 0 sum

Given an array of positive and negative numbers, find if there is a subarray (of size at-least one) with 0 sum.

Examples :

```
Input: {4, 2, -3, 1, 6}

Output: true

There is a subarray with zero sum from index 1 to 3.


Input: {4, 2, 0, 1, 6}

Output: true

There is a subarray with zero sum from index 2 to 2.


Input: {-3, 2, 3, 1, 6}

Output: false

There is no subarray with zero sum.
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

A simple solution is to consider all subarrays one by one and check the sum of every subarray. We can run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i (See this for implementation). Time complexity of this method is O(n2).

We can also use hashing. The idea is to iterate through the array and for every element arr[i], calculate sum of elements form 0 to i (this can simply be done as sum += arr[i]). If the current sum has been seen before, then there is a zero sum array. Hashing is used to store the sum values, so that we can quickly store sum and find out whether the current sum is seen before or not.
Example :

```
arr[] = {1, 4, -2, -2, 5, -4, 3}


If we consider all prefix sums, we can

notice that there is a subarray with 0

sum when :

1) Either a prefix sum repeats or

2) Or prefix sum becomes 0.


Prefix sums for above array are:

1, 5, 3, 1, 6, 2, 5
```

```
Since prefix sum 1 repeats, we have a subarray
with 0 sum.
```

Following is implementation of the above approach.

```
// A C++ program to find if there is a zero sum
// subarray
#include <bits/stdc++.h>
using namespace std;

bool subArrayExists(int arr[], int n)
{
    unordered_set<int> sumSet;

    // Traverse through array and store prefix sums
    int sum = 0;
    for (int i = 0 ; i < n ; i++)
    {
        sum += arr[i];

        // If prefix sum is 0 or it is already present
        if (sum == 0 || sumSet.find(sum) != sumSet.end())
            return true;

        sumSet.insert(sum);
    }
    return false;
}

// Driver code
int main()
{
    int arr[] =  {-3, 2, 3, 1, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (subArrayExists(arr, n))
        cout << "Found a subarray with 0 sum";
    else
        cout << "No Such Sub Array Exists!";
    return 0;
}
```

Output :

```
No Such Sub Array Exists!
```

Time Complexity of this solution can be considered as O(n) under the assumption that we have good hashing function that allows insertion and retrieval operations in O(1) time.

19. Largest subarray with equal number of 0s and 1s

Given an array containing only 0s and 1s, find the largest subarray which contain equal no of 0s and 1s. Expected time complexity is O(n).

Examples:

```
Input: arr[] = {1, 0, 1, 1, 1, 0, 0}

Output: 1 to 6 (Starting and Ending indexes of output subarray)


Input: arr[] = {1, 1, 1, 1}

Output: No such subarray


Input: arr[] = {0, 0, 1, 1, 0}

Output: 0 to 3 Or 1 to 4
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple)

A simple method is to use two nested loops. The outer loop picks a starting point i. The inner loop considers all subarrays starting from i. If size of a subarray is greater than maximum size so far, then update the maximum size. In the below code, 0s are considered as -1 and sum of all values from i to j is calculated. If sum becomes 0, then size of this subarray is compared with largest size so far.

```
// A simple C++ program to find the largest subarray
// with equal number of 0s and 1s
#include<bits/stdc++.h>

using namespace std;

// This function Prints the starting and ending
// indexes of the largest subarray with equal
// number of 0s and 1s. Also returns the size
// of such subarray.

int findSubArray(int arr[], int n)
{
    int sum = 0;
    int maxsize = -1, startindex;

    // Pick a starting point as i
    for (int i = 0; i < n-1; i++)
    {
        sum = (arr[i] == 0)? -1 : 1;

        // Consider all subarrays starting from i
```

```
        for (int j = i+1; j < n; j++)
        {
            (arr[j] == 0)? (sum += -1): (sum += 1);

            // If this is a 0 sum subarray, then
            // compare it with maximum size subarray
            // calculated so far
            if (sum == 0 && maxsize < j-i+1)
            {
                maxsize = j - i + 1;
                startindex = i;
            }
        }
    }
    if (maxsize == -1)
        cout << "No such subarray";
    else
        cout << startindex << " to " << startindex + maxsize - 1;

    return maxsize;
}

/* Driver code*/
int main()
{
    int arr[] = {1, 0, 0, 1, 0, 1, 1};
    int size = sizeof(arr)/sizeof(arr[0]);

    findSubArray(arr, size);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
 0 to 5
```

Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Tricky)
Following is a solution that uses O(n) extra space and solves the problem in O(n) time complexity.
Let input array be arr[] of size n and maxsize be the size of output subarray.
1) Consider all 0 values as -1. The problem now reduces to find out the maximum length subarray with sum = 0.
2) Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i]. This can be done in O(n) time.
3) There are two cases, the output subarray may start from 0th index or may start from some other index. We will return the max of the values obtained by

two cases.
4) To find the maximum length subarray starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.
5) Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we can create a hash table with size = max-min+1 where min is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. The idea is to hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because there can be these many different possible values in sumleft[]. Initialize all values in hash as -1
6) To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.
7) To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

```
// A O(n) program to find the largest subarray
// with equal number of 0s and 1s

#include <stdio.h>
#include <stdlib.h>

// A utility function to get maximum of two
// integers

int max(int a, int b) { return a>b? a: b; }

// This function Prints the starting and ending
// indexes of the largest subarray with equal
// number of 0s and 1s. Also returns the size
// of such subarray.

int findSubArray(int arr[], int n)
{
    // variables to store result values

    int maxsize = -1, startindex;

    // Create an auxiliary array sunmleft[].
    // sumleft[i] will be sum of array
    // elements from arr[0] to arr[i]

    int sumleft[n];

    // For min and max values in sumleft[]

    int min, max;
```

```
    int i;

    // Fill sumleft array and get min and max
    // values in it.  Consider 0 values in arr[]
    // as -1

    sumleft[0] = ((arr[0] == 0)? -1: 1);
    min = arr[0]; max = arr[0];
    for (i=1; i<n; i++)
    {
        sumleft[i] = sumleft[i-1] + ((arr[i] == 0)?
                     -1: 1);
        if (sumleft[i] < min)
            min = sumleft[i];
        if (sumleft[i] > max)
            max = sumleft[i];
    }

    // Now calculate the max value of j - i such
    // that sumleft[i] = sumleft[j]. The idea is
    // to create a hash table to store indexes of all
    // visited values.
    // If you see a value again, that it is a case of
    // sumleft[i] = sumleft[j]. Check if this j-i is
    // more than maxsize.
    // The optimum size of hash will be max-min+1 as
    // these many different values of sumleft[i] are
    // possible. Since we use optimum size, we need
    // to shift all values in sumleft[] by min before
    // using them as an index in hash[].

    int hash[max-min+1];

    // Initialize hash table

    for (i=0; i<max-min+1; i++)
        hash[i] = -1;

    for (i=0; i<n; i++)
    {
        // Case 1: when the subarray starts from
        //         index 0

        if (sumleft[i] == 0)
        {
           maxsize = i+1;
           startindex = 0;
        }
```

```
        // Case 2: fill hash table value. If already
        //         filled, then use it

        if (hash[sumleft[i]-min] == -1)
            hash[sumleft[i]-min] = i;
        else
        {
            if ((i - hash[sumleft[i]-min]) > maxsize)
            {
                maxsize = i - hash[sumleft[i]-min];
                startindex = hash[sumleft[i]-min] + 1;
            }
        }
    }
    if (maxsize == -1)
        printf("No such subarray");
    else
        printf("%d to %d", startindex, startindex+maxsize-1);

    return maxsize;
}

/* Driver program to test above functions */
int main()
{
    int arr[] =  {1, 0, 0, 1, 0, 1, 1};
    int size = sizeof(arr)/sizeof(arr[0]);

    findSubArray(arr, size);
    return 0;
}
```

Output:

```
0 to 5
```

Time Complexity: O(n)
Auxiliary Space: O(n)

19. A Product Array Puzzle

Given an array arr[] of n integers, construct a Product Array prod[] (of same size) such that prod[i] is equal to the product of all the elements of arr[] except arr[i]. Solve it without division operator and in O(n).
Example :
arr[] = {10, 3, 5, 6, 2}
prod[] = {180, 600, 360, 300, 900}
Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.
Algorithm:
1) Construct a temporary array left[] such that left[i] contains product of all elements on left of arr[i] excluding arr[i].
2) Construct another temporary array right[] such that right[i] contains product of all elements on on right of arr[i] excluding arr[i].
3) To get prod[], multiply left[] and right[].

Implementation:

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

/* Function to print product array for a given array
arr[] of size n */
void productArray(int arr[], int n)
{

    // Base case
    if (n == 1) {
        cout << 0;
        return;
    }
    /* Allocate memory for temporary arrays left[] and right[] */
    int* left = new int[sizeof(int) * n];
    int* right = new int[sizeof(int) * n];

    /* Allocate memory for the product array */
    int* prod = new int[sizeof(int) * n];

    int i, j;

    /* Left most element of left array is always 1 */
    left[0] = 1;

    /* Rightmost most element of right array is always 1 */
    right[n - 1] = 1;
```

```
    /* Construct the left array */
    for (i = 1; i < n; i++)
        left[i] = arr[i - 1] * left[i - 1];

    /* Construct the right array */
    for (j = n - 2; j >= 0; j--)
        right[j] = arr[j + 1] * right[j + 1];

    /* Construct the product array using
        left[] and right[] */
    for (i = 0; i < n; i++)
        prod[i] = left[i] * right[i];

    /* print the constructed prod array */
    for (i = 0; i < n; i++)
        cout << prod[i] << " ";

    return;
}

/* Driver code*/
int main()
{
    int arr[] = { 10, 3, 5, 6, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The product array is: \n";
    productArray(arr, n);
}

// This is code is contributed by rathbhupendra
```

Output :

```
The product array is :

180 600 360 300 900
```

Time Complexity: O(n)
Space Complexity: O(n)
Auxiliary Space: O(n)
The above method can be optimized to work in space complexity O(1). Thanks to Dileep for suggesting the below solution.

- C++

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

/* Function to print product array
```

```
for a given array arr[] of size n */
void productArray(int arr[], int n)
{

    // Base case
    if (n == 1) {
        cout << 0;
        return;
    }

    int i, temp = 1;

    /* Allocate memory for the product array */
    int* prod = new int[(sizeof(int) * n)];

    /* Initialize the product array as 1 */
    memset(prod, 1, n);

    /* In this loop, temp variable contains product of
       elements on left side excluding arr[i] */
    for (i = 0; i < n; i++) {
        prod[i] = temp;
        temp *= arr[i];
    }

    /* Initialize temp to 1
    for product on right side */
    temp = 1;

    /* In this loop, temp variable contains product of
       elements on right side excluding arr[i] */
    for (i = n - 1; i >= 0; i--) {
        prod[i] *= temp;
        temp *= arr[i];
    }

    /* print the constructed prod array */
    for (i = 0; i < n; i++)
        cout << prod[i] << " ";

    return;
}

// Driver Code
int main()
{
    int arr[] = { 10, 3, 5, 6, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The product array is: \n";
```

```
    productArray(arr, n);
}

// This code is contributed by rathbhupendra
```

Output :

```
The product array is :

180 600 360 300 900
```

Time Complexity: O(n)
Space Complexity: O(n)
Auxiliary Space: O(1)

20. A product array puzzle | Set 2 (O(1) Space)
Given an array arr[] of n integers, construct a Product Array prod[] (of same size) such that prod[i] is equal to the product of all the elements of arr[] except arr[i]. Solve it without division operator and in O(n).

Example:

```
Input : arr[] = {10, 3, 5, 6, 2}

Output : prod[] = {180, 600, 360, 300, 900}
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.
We have already discussed O(n) approach in A product array puzzle | set 1. The previous approach uses extra O(n) space for constructing product array.
In this post, a better approach has been discussed which uses log property to find the product of all elements of array except at particular index. This approach uses no extra space.

Use property of log to multiply large numbers
x = a * b * c * d
log(x) = log(a * b * c * d)
log(x) = log(a) + log(b) + log(c) + log(d)
x = antilog(log(a) + log(b) + log(c) + log(d))

```
// C++ program for product array puzzle
// with O(n) time and O(1) space.
#include <bits/stdc++.h>
using namespace std;

// epsilon value to maintain precision
#define EPS 1e-9

void productPuzzle(int a[], int n)
{
    // to hold sum of all values
    long double sum = 0;
    for (int i = 0; i < n; i++)
        sum += (long double)log10(a[i]);

    // output product for each index
    // antilog to find original product value
    for (int i = 0; i < n; i++)
        cout << (int)(EPS + pow((long double)10.00,
                          sum - log10(a[i]))) << " ";
}

// Driver code
int main()
{
    int a[] = { 10, 3, 5, 6, 2 };
```

```
    int n = sizeof(a)/sizeof(a[0]);
    cout << "The product array is: \n";
    productPuzzle(a, n);
    return 0;
}
```

Output:

```
The product array is:

180 600 360 300 900
```

Time complexity : O(n)
Space complexity: O(1)

22. Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Examples:

```
Input : arr[] = {6, 5, 3, 2, 8, 10, 9}
           k = 3
Output : arr[] = {2, 3, 5, 6, 8, 9, 10}


Input : arr[] = {10, 9, 8, 7, 4, 70, 60, 50}
         k = 4
Output : arr[] = {4, 7, 8, 9, 10, 50, 60, 70}
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

We can use Insertion Sort to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
   int i, key, j;
   for (i = 1; i < size; i++)
   {
       key = A[i];
       j = i-1;

       /* Move elements of A[0..i-1], that are greater than key,
to one
          position ahead of their current position.
          This loop will run at most k times */
       while (j >= 0 && A[j] > key)
       {
           A[j+1] = A[j];
           j = j-1;
       }
       A[j+1] = key;
   }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be O(nk)

We can sort such arrays more efficiently with the help of Heap data structure. Following is the detailed process that uses Heap.
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time (See this GFact)
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.
Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)*logK)

```
// A STL based C++ program to sort a nearly sorted array.
#include <bits/stdc++.h>
using namespace std;

// Given an array of size n, where every element
// is k away from its target position, sorts the
// array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Insert first k+1 items in a priority queue (or min heap)
    //(A O(k) operation). We assume, k < n.
    priority_queue<int, vector<int>, greater<int> > pq(arr, arr +
k + 1);

    // i is index for remaining elements in arr[] and index
    // is target index of for current minimum element in
    // Min Heapm 'hp'.
    int index = 0;
    for (int i = k + 1; i < n; i++) {
        arr[index++] = pq.top();
        pq.pop();
        pq.push(arr[i]);
    }

    while (pq.empty() == false) {
        arr[index++] = pq.top();
        pq.pop();
    }
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
```

```
{
    int k = 3;
    int arr[] = { 2, 6, 3, 12, 56, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array" << endl;
    printArray(arr, n);

    return 0;
}
```

Output:

```
Following is sorted array

2 3 6 8 12 56
```

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also use a Balanced Binary Search Tree instead of Heap to store K+1 elements. The insert and delete operations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn’t need extra space for left and right pointers.

23. Find duplicates in O(n) time and O(1) extra space | Set 1

Given an array of n elements which contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space.

For example, let n be 7 and array be {1, 2, 3, 1, 3, 6, 6}, the answer should be 1, 3 and 6.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.
This problem is an extended version of the following problem.

Find the two repeating elements in a given array
Method 1 and Method 2 of the above link are not applicable as the question says O(n) time complexity and O(1) constant space. Also, Method 3 and Method 4 cannot be applied here because there can be more than 2 repeating elements in this problem. Method 5 can be extended to work for this problem. Below is the solution that is similar to Method 5.

Algorithm:

```
traverse the list for i= 0 to n-1 elements
{
  check for sign of A[abs(A[i])] ;
  if positive then
     make it negative by   A[abs(A[i])]=-A[abs(A[i])];
  else  // i.e., A[abs(A[i])] is negative
     this   element (ith element of list) is a repetition
}
```

Implementation:

```
// C++ code to find
// duplicates in O(n) time
#include <bits/stdc++.h>
using namespace std;

// Function to print duplicates
void printRepeating(int arr[], int size)
{
int i;
cout << "The repeating elements are:" << endl;
for (i = 0; i < size; i++)
{
```

```
    if (arr[abs(arr[i])] >= 0)
    arr[abs(arr[i])] = -arr[abs(arr[i])];
    else
    cout << abs(arr[i]) << " ";
}
}

// Driver Code
int main()
{
    int arr[] = {1, 2, 3, 1, 3, 6, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    return 0;
}

// This code is contributed
// by Akanksha Rai
```

Output:

```
The repeating elements are:
1  3  6
```

Note: The above program doesn't handle 0 case (If 0 is present in array). The program can be easily modified to handle that also. It is not handled to keep the code simple.

Time Complexity: O(n)
Auxiliary Space: O(1)

There is a problem in the above approach. It prints the repeated number more than once. For example: {1, 6, 3, 1, 3, 6, 6} it will give output as : 1 3 6 6. In below set, another approach is discussed that prints repeating elements only once.

Duplicates in an array in O(n) and by using O(1) extra space | Set-2
Another O(n) & O(1) code:

play_arrow
brightness_4

```
// C++ code to find
// duplicates in O(n) time
#include <bits/stdc++.h>
using namespace std;

int main()
{

    int numRay[] = {0, 4, 3, 2, 7, 8, 2, 3, 1};
```

```
    int arr_size = sizeof(numRay) /
                   sizeof(numRay[0]);
    for (int i = 0; i < arr_size; i++)
    {
        numRay[numRay[i] % 10] = numRay[numRay[i] % 10] + 10;
    }
    cout << "The repeating elements are : " << endl;
    for (int i = 0; i < arr_size; i++)
    {
        if (numRay[i] > 19)
        {
            cout << i << " " << endl;
        }
    }
    return 0;
}

// This code is contributed by PrinciRaj1992
```

Output:

```
The repeating elements are :
2
3
```

24. Duplicates in an array in O(n) time and by using O(1) extra space | Set-3

Given an array of n elements which contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space. It is required that the order in which elements repeat should be maintained. If there is no repeating element present then print -1.

Examples:

```
Input : arr[] = {1, 2, 3, 1, 3, 6, 6}

Output : 1, 3, 6

Elements 1, 3 and 6 are repeating.

Second occurrence of 1 is found

first followed by repeated occurrence

of 3 and 6.


Input : arr[] = {0, 3, 1, 3, 0}

Output : 3, 0

Second occurrence of 3 is found

first followed by second occurrence

of 0.
```

Recommended: Please solve it on “PRACTICE” first, before moving on to the solution.

We have discussed two approaches for this question in below posts:

Find duplicates in O(n) time and O(1) extra space | Set 1

Duplicates in an array in O(n) time and by using O(1) extra space | Set-2

There is a problem in first approach. It prints the repeated number more than once. For example: {1, 6, 3, 1, 3, 6, 6} it will give output as : 3 6 6. In second approach although each repeated item is printed only once, the order in which their repetition occurs is not maintained. To print elements in order in which they are repeating, the second approach is modified. To mark the presence of an element size of the array, n is added to the index position arr[i] corresponding to array element arr[i]. Before adding n, check if value at index arr[i] is greater than or equal to n or not. If it is greater than or equal to, then this means that element arr[i] is repeating. To avoid printing repeating element multiple times, check if it is the first repetition of arr[i] or not. It is first repetition if value at index position arr[i] is less than 2*n. This is because, if element arr[i] has occurred only once before then n is added to index arr[i] only once and thus value at index arr[i] is less than 2*n. Add n to index arr[i] so that value becomes greater than or equal to 2*n and it will prevent further printing of current repeating element. Also if value at index arr[i] is less than n then it is

first occurrence (not repetition) of element arr[i]. So to mark this add n to element at index arr[i].

Below is the implementation of above approach:

```
// C++ program to print all elements that
// appear more than once.
#include <bits/stdc++.h>
using namespace std;

// Function to find repeating elements
void printDuplicates(int arr[], int n)
{
    int i;

    // Flag variable used to
    // represent whether repeating
    // element is found or not.
    int fl = 0;

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

        // Check if current element is
        // repeating or not. If it is
        // repeating then value will
        // be greater than or equal to n.
        if (arr[arr[i] % n] >= n) {

            // Check if it is first
            // repetition or not. If it is
            // first repetition then value
            // at index arr[i] is less than
            // 2*n. Print arr[i] if it is
            // first repetition.
            if (arr[arr[i] % n] < 2 * n) {
                cout << arr[i] % n << " ";
                fl = 1;
            }
        }

        // Add n to index arr[i] to mark
        // presence of arr[i] or to
        // mark repetition of arr[i].
        arr[arr[i] % n] += n;
    }

    // If flag variable is not set
    // then no repeating element is
```

```
    // found. So print -1.
    if (!fl)
        cout << "-1";
}

// Driver Function
int main()
{
    int arr[] = { 1, 6, 3, 1, 3, 6, 6 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printDuplicates(arr, arr_size);
    return 0;
}
```

Output:

```
1 3 6
```

Time Complexity: O(n)
Auxiliary Space: O(1)

25. Find the two non-repeating elements in an array of repeating elements

Asked by SG
Given an array in which all numbers except two are repeated once. (i.e. we have 2n+2 numbers and n numbers are occurring twice and remaining two have occurred once). Find those two numbers in the most efficient way.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1(Use Sorting)
First, sort all the elements. In the sorted array, by comparing adjacent elements we can easily get the non-repeating elements. Time complexity of this method is O(nLogn)
Method 2(Use XOR)
Let x and y be the non-repeating elements we are looking for and arr[] be the input array. First, calculate the XOR of all the array elements.

```
    xor = arr[0]^arr[1]^arr[2].....arr[n-1]
```

All the bits that are set in xor will be set in one non-repeating element (x or y) and not in others. So if we take any set bit of xor and divide the elements of the array in two sets – one set of elements with same bit set and another set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in the first set, we will get the first non-repeating element, and by doing same in other sets we will get the second non-repeating element.

```
Let us see an example.
  arr[] = {2, 4, 7, 9, 2, 4}
1) Get the XOR of all the elements.
    xor = 2^4^7^9^2^4 = 14 (1110)
2) Get a number which has only one set bit of the xor.
  Since we can easily get the rightmost set bit, let us use it.
    set_bit_no = xor & ~(xor-1) = (1110) & ~(1101) = 0010
  Now set_bit_no will have only set as rightmost set bit of xor.
3) Now divide the elements in two sets and do xor of
  elements in each set and we get the non-repeating
  elements 7 and 9. Please see the implementation for this step.
```

Implementation:

- C++
- C

filter_none

edit
play_arrow
brightness_4

```
#include <bits/stdc++.h>
using namespace std;

/* This finction sets the values of
*x and *y to nonr-epeating elements
in an array arr[] of size n*/
void get2NonRepeatingNos(int arr[], int n, int *x, int *y)
{
    int Xor = arr[0]; /* Will hold Xor of all elements */
    int set_bit_no; /* Will have only single set bit of Xor */
    int i;
    *x = 0;
    *y = 0;

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

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

    /* Now divide elements in two sets by comparing rightmost set
    bit of Xor with bit at same position in each element. */
    for(i = 0; i < n; i++)
    {
        if(arr[i] & set_bit_no)
        *x = *x ^ arr[i]; /*Xor of first set */
        else
        *y = *y ^ arr[i]; /*Xor of second set*/
    }
}

/* Driver code */
int main()
{
    int arr[] = {2, 3, 7, 9, 11, 2, 3, 11};
    int *x = new int[(sizeof(int))];
    int *y = new int[(sizeof(int))];
    get2NonRepeatingNos(arr, 8, x, y);
    cout<<"The non-repeating elements are "<<*x<<" and "<<*y;
}

// This code is contributed by rathbhupendra
```

Output:

```
The non-repeating elements are 7 and 9
```

Time Complexity: O(n)
Auxiliary Space: O(1)

26. Program for array rotation

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.

Rotation of the above array by 2 will make array

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

METHOD 1 (Using temp array)

```
Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n =7
1) Store d elements in a temp array
   temp[] = [1, 2]
2) Shift rest of the arr[]
   arr[] = [3, 4, 5, 6, 7, 6, 7]
3) Store back the d elements
   arr[] = [3, 4, 5, 6, 7, 1, 2]
```

Time complexity : O(n)
Auxiliary Space : O(d)

METHOD 2 (Rotate one by one)

```
leftRotate(arr[], d, n)
start
  For i = 0 to i < d
    Left rotate all elements of arr[] by one
end
```

To rotate by one, store arr[0] in a temporary variable temp, move arr[1] to arr[0], arr[2] to arr[1] ...and finally temp to arr[n-1]

Let us take the same example arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2
Rotate arr[] by one 2 times
We get [2, 3, 4, 5, 6, 7, 1] after first rotation and [3, 4, 5, 6, 7, 1, 2] after second rotation.

Below is the implementation of the above approach :

```cpp
// C++ program to rotate an array by
// d elements
#include <bits/stdc++.h>
using namespace std;

/*Function to left Rotate arr[] of
  size n by 1*/
void leftRotatebyOne(int arr[], int n)
{
    int temp = arr[0], i;
    for (i = 0; i < n - 1; i++)
        arr[i] = arr[i + 1];

    arr[i] = temp;
}

/*Function to left rotate arr[] of size n by d*/
void leftRotate(int arr[], int d, int n)
{
    for (int i = 0; i < d; i++)
        leftRotatebyOne(arr, n);
}

/* utility function to print an array */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

/* Driver program to test above functions */
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function calling
    leftRotate(arr, 2, n);
    printArray(arr, n);

    return 0;
}
```

Output :

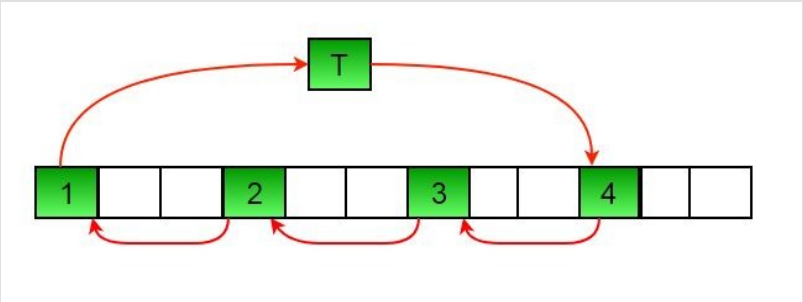
```
3 4 5 6 7 1 2
```

Time complexity : O(n * d)
Auxiliary Space : O(1)

METHOD 3 (A Juggling Algorithm)
This is an extension of method 2. Instead of moving one by one, divide the array in different sets
where number of sets is equal to GCD of n and d and move the elements within sets.
If GCD is 1 as is for the above example array (n = 7 and d =2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.
Here is an example for n =12 and d = 3. GCD is 3 and

```
Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

a) Elements are first moved in first set – (See below
   diagram for this movement)
```




```
         arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}

b)    Then in second set.
         arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}

c)    Finally in third set.
         arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}
```

Below is the implementation of the above approach :

```
// C++ program to rotate an array by
// d elements
#include <bits/stdc++.h>
using namespace std;

/*Fuction to get gcd of a and b*/
int gcd(int a, int b)
{
```

```
    if (b == 0)
        return a;

    else
        return gcd(b, a % b);
}

/*Function to left rotate arr[] of siz n by d*/
void leftRotate(int arr[], int d, int n)
{
    int g_c_d = gcd(d, n);
    for (int i = 0; i < g_c_d; i++) {
        /* move i-th values of blocks */
        int temp = arr[i];
        int j = i;

        while (1) {
            int k = j + d;
            if (k >= n)
                k = k - n;

            if (k == i)
                break;

            arr[j] = arr[k];
            j = k;
        }
        arr[j] = temp;
    }
}

// Function to print an array
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
}

/* Driver program to test above functions */
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function calling
    leftRotate(arr, 2, n);
    printArray(arr, n);

    return 0;
}
```

Output :

```
3  4  5  6  7  1  2
```

Time complexity : O(n)
Auxiliary Space : O(1)

26. Sort an array in wave form

Given an unsorted array of integers, sort the array into a wave like array. An array ‘arr[0..n-1]’ is sorted in wave form if arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= .....

Examples:

```
Input:  arr[] = {10, 5, 6, 3, 2, 20, 100, 80}
Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80} OR
                {20, 5, 10, 2, 80, 6, 100, 3} OR
                any other array that is in wave form

Input:  arr[] = {20, 10, 8, 6, 4, 2}
Output: arr[] = {20, 8, 10, 4, 6, 2} OR
                {10, 8, 20, 2, 6, 4} OR
                any other array that is in wave form

Input:  arr[] = {2, 4, 6, 8, 10, 20}
Output: arr[] = {4, 2, 8, 6, 20, 10} OR
                any other array that is in wave form

Input:  arr[] = {3, 6, 5, 10, 7, 20}
Output: arr[] = {6, 3, 10, 5, 20, 7} OR
                any other array that is in wave form
```

Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

A Simple Solution is to use sorting. First sort the input array, then swap all adjacent elements.

For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.

Below are implementations of this simple approach.

```
// A C++ program to sort an array in wave form using
// a sorting function
#include<iostream>
#include<algorithm>
using namespace std;

// A utility method to swap two numbers.
```

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e.,
// arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]..
void sortInWave(int arr[], int n)
{
    // Sort the input array
    sort(arr, arr+n);

    // Swap adjacent elements
    for (int i=0; i<n-1; i += 2)
        swap(&arr[i], &arr[i+1]);
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

Output:

```
2 1 10 5 49 23 90
```

The time complexity of the above solution is O(nLogn) if a O(nLogn) sorting algorithm like Merge Sort, Heap Sort, .. etc is used.
This can be done in O(n) time by doing a single traversal of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about odd positioned element. Following are simple steps.
1) Traverse all even positioned elements of input array, and do following.
....a) If current element is smaller than previous odd element, swap previous and current.
....b) If current element is smaller than next odd element, swap next and current.
Below are implementations of above simple algorithm.

```
// A O(n) program to sort an input array in wave form
#include<iostream>
using namespace std;
```

```
// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=
// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....
void sortInWave(int arr[], int n)
{
    // Traverse all even elements
    for (int i = 0; i < n; i+=2)
    {
        // If current even element is smaller than previous
        if (i>0 && arr[i-1] > arr[i] )
            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next
        if (i<n-1 && arr[i] < arr[i+1] )
            swap(&arr[i], &arr[i + 1]);
    }
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

Output:

```
90 10 49 1 5 2 23
```

27. Find next greater number with same set of digits

Given a number n, find the smallest number that has same set of digits as n and is greater than n. If n is the greatest possible number with its set of digits, then print “not possible”.

Examples:
For simplicity of implementation, we have considered input number as a string.

```
Input:  n = "218765"
Output: "251678"

Input:  n = "1234"
Output: "1243"

Input: n = "4321"
Output: "Not Possible"

Input: n = "534976"
Output: "536479"
```

Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

Following are few observations about the next greater number.
1) If all digits sorted in descending order, then output is always “Not Possible”. For example, 4321.
2) If all digits are sorted in ascending order, then we need to swap last two digits. For example, 1234.
3) For other cases, we need to process the number from rightmost side (why? because we need to find the smallest of all greater numbers)

You can now try developing an algorithm yourself.

Following is the algorithm for finding the next greater number.
I) Traverse the given number from rightmost digit, keep traversing till you find a digit which is smaller than the previously traversed digit. For example, if the input number is “534976”, we stop at 4 because 4 is smaller than next digit 9. If we do not find such a digit, then output is “Not Possible”.
II) Now search the right side of above found digit ‘d’ for the smallest digit greater than ‘d’. For “534976″, the right side of 4 contains “976”. The smallest digit greater than 4 is 6.
III) Swap the above found two digits, we get 536974 in above example.
IV) Now sort all digits from position next to ‘d’ to the end of number. The number that we get after sorting is the output. For above example, we sort digits in bold 536974. We get “536479” which is the next greater number for input 534976.
Following are the implementation of above approach.

```cpp
// C++ program to find the smallest number which greater than a
given number
// and has same set of digits as given number
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Utility function to swap two digits
void swap(char *a, char *b)
{
    char temp = *a;
    *a = *b;
    *b = temp;
}

// Given a number as a char array number[], this function finds
the
// next greater number.  It modifies the same array to store the
result
void findNext(char number[], int n)
{
    int i, j;

    // I) Start from the right most digit and find the first digit
that is
    // smaller than the digit next to it.
    for (i = n-1; i > 0; i--)
        if (number[i] > number[i-1])
           break;

    // If no such digit is found, then all digits are in
descending order
    // means there cannot be a greater number with same set of
digits
    if (i==0)
    {
        cout << "Next number is not possible";
        return;
    }

    // II) Find the smallest digit on right side of (i-1)'th digit
that is
    // greater than number[i-1]
    int x = number[i-1], smallest = i;
    for (j = i+1; j < n; j++)
        if (number[j] > x && number[j] < number[smallest])
            smallest = j;
```

```
    // III) Swap the above found smallest digit with number[i-1]
    swap(&number[smallest], &number[i-1]);

    // IV) Sort the digits after (i-1) in ascending order
    sort(number + i, number + n);

    cout << "Next number with same set of digits is " << number;

    return;
}

// Driver program to test above function
int main()
{
    char digits[] = "534976";
    int n = strlen(digits);
    findNext(digits, n);
    return 0;
}
```

Output:

```
Next number with same set of digits is 536479
```

The above implementation can be optimized in following ways.
1) We can use binary search in step II instead of linear search.
2) In step IV, instead of doing simple sort, we can apply some clever technique to do it in linear time. Hint: We know that all digits are linearly sorted in reverse order except one digit which was swapped.

With above optimizations, we can say that the time complexity of this method is O(n).

29. Trapping Rain Water

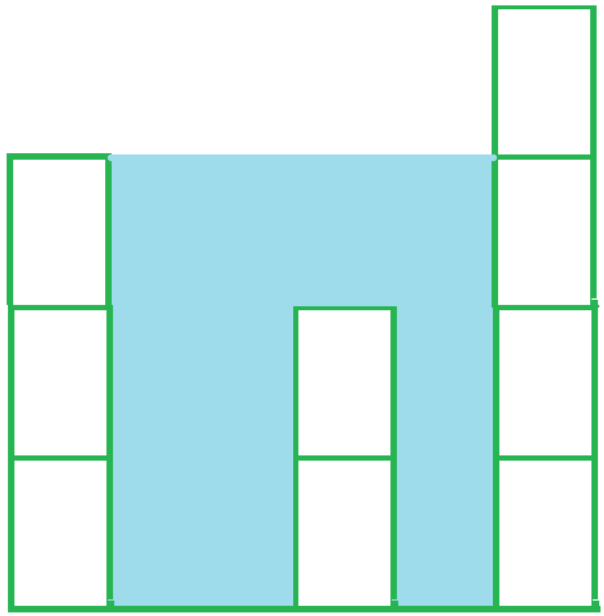
Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

Examples:

```
Input: arr[]   = {2, 0, 2}
Output: 2
Structure is like below
| |
|_|
We can trap 2 units of water in the middle gap.

Input: arr[]   = {3, 0, 0, 2, 0, 4}
Output: 10
Structure is like below
     |
|    |
|  | |
|__|_|
We can trap "3*2 units" of water between 3 an 2,
"1 unit" on top of bar 2 and "3 units" between 2
and 4.  See below diagram also.

Input: arr[] = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
Output: 6
      |
   |   || |
_|_||_||||||
Trap "1 unit" between first 1 and 2, "4 units" between
first 2 and 3 and "1 unit" between second last 1 and last 2
```



Bars for input {3, 0, 0, 2, 0, 4}
Total trapped water = 3 + 3 + 1 + 3 = 10

We strongly recommend that you click here and practice it, before moving on to the solution.

An element of the array can store water if there are higher bars on left and right. We can find the amount of water to be stored in every element by finding the heights of bars on left and right sides. The idea is to compute the amount of water that can be stored in every element of array. For example, consider the array {3, 0, 0, 2, 0, 4}, we can store three units of water at indexes 1 and 2, and one unit of water at index 3, and three units of water at index 4.

A Simple Solution is to traverse every array element and find the highest bars on left and right sides. Take the smaller of two heights. The difference between the smaller height and height of the current element is the amount of water that can be stored in this array element. Time complexity of this solution is $O(n^2)$.
Below is the implementation of the above approach:

```
// C++ implementation of the approach
#include<bits/stdc++.h>
using namespace std;

// Function to return the maximum
// water that can be stored
int maxWater(int arr[], int n)
{
```

```
    // To store the maximum water
    // that can be stored
    int res = 0;

    // For every element of the array
    for (int i = 1; i < n-1; i++) {

        // Find the maximum element on its left
        int left = arr[i];
        for (int j=0; j<i; j++)
           left = max(left, arr[j]);

        // Find the maximum element on its right
        int right = arr[i];
        for (int j=i+1; j<n; j++)
           right = max(right, arr[j]);

       // Update the maximum water
       res = res + (min(left, right) - arr[i]);
    }

    return res;
}

// Driver code
int main()
{
    int arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << maxWater(arr, n);

    return 0;
}
```

Output:

```
6
```

An Efficient Solution is to pre-compute highest bar on left and right of every bar in O(n) time. Then use these pre-computed values to find the amount of water in every array element.
Below is the implementation of this solution.

```
// C++ program to find maximum amount of water that can
// be trapped within given set of bars.
#include <bits/stdc++.h>
using namespace std;

int findWater(int arr[], int n)
{
```

```
    // left[i] contains height of tallest bar to the
    // left of i'th bar including itself
    int left[n];

    // Right [i] contains height of tallest bar to
    // the right of ith bar including itself
    int right[n];

    // Initialize result
    int water = 0;

    // Fill left array
    left[0] = arr[0];
    for (int i = 1; i < n; i++)
        left[i] = max(left[i - 1], arr[i]);

    // Fill right array
    right[n - 1] = arr[n - 1];
    for (int i = n - 2; i >= 0; i--)
        right[i] = max(right[i + 1], arr[i]);

    // Calculate the accumulated water element by element
    // consider the amount of water on i'th bar, the
    // amount of water accumulated on this particular
    // bar will be equal to min(left[i], right[i]) - arr[i] .
    for (int i = 0; i < n; i++)
        water += min(left[i], right[i]) - arr[i];

    return water;
}

// Driver program
int main()
{
    int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum water that can be accumulated is "
         << findWater(arr, n);
    return 0;
}
```

Output:

```
Maximum water that can be accumulated is 6
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Space Optimization in above solution: Instead of maintaing two arrays of size n for storing left and right max of each element, we will just maintain two variables to store the maximum till that point. Since water trapped at any

element = min(max_left, max_right) – arr[i] we will calculate water trapped on smaller element out of A[lo] and A[hi] first and move the pointers till lo doesn't cross hi.
Below is the implementation of the above approach:

```
// C++ program to find maximum amount of water that can
// be trapped within given set of bars.
// Space Complexity : O(1)

#include <iostream>
using namespace std;

int findWater(int arr[], int n)
{
    // initialize output
    int result = 0;

    // maximum element on left and right
    int left_max = 0, right_max = 0;

    // indices to traverse the array
    int lo = 0, hi = n - 1;

    while (lo <= hi) {
        if (arr[lo] < arr[hi]) {
            if (arr[lo] > left_max)
                // update max in left
                left_max = arr[lo];
            else
                // water on curr element = max - curr
                result += left_max - arr[lo];
            lo++;
        }
        else {
            if (arr[hi] > right_max)
                // update right maximum
                right_max = arr[hi];
            else
                result += right_max - arr[hi];
            hi--;
        }
    }

    return result;
}

int main()
{
```

```
    int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum water that can be accumulated is "
         << findWater(arr, n);
}

// This code is contributed by Aditi Sharma
```

Output:

```
Maximum water that can be accumulated is 6
```

Time Complexity: O(n)
Auxiliary Space: O(1)
Thanks to Gaurav Ahirwar and Aditi Sharma for above solution.
Here is another approach for O(n) time and O(1) space complexity.

- Loop from index 0 to the end of the given array.
- If a wall greater than or equal to the previous wall is encountered then make note of the index of that wall in a var called prev_index.
- Keep adding previous wall's height minus the current (ith) wall to the variable water.
- Have a temporary variable that stores the same value as water.
- If no wall greater than or equal to the previous wall is found then quit.
- If prev_index < size of the input array then subtract the temp variable from water, and loop from end of the input array to prev_index and find a wall greater than or equal to the previous wall (in this case, the last wall from backwards).

The concept here is that if there is a larger wall to the right then the water can be retained with height equal to the smaller wall on the left. If there are no larger walls to the right then start from the left. There must be a larger wall to the left now.

Below is the implementation of the above approach:

```
// Java implementation of the approach
class GFG {

    // Function to return the maximum
    // water that can be stored
    public static int maxWater(int arr[], int n)
    {
        int size = n - 1;

        // Let the first element be stored as
        // previous, we shall loop from index 1
        int prev = arr[0];

        // To store previous wall's index
        int prev_index = 0;
        int water = 0;
```

```
// To store the water until a larger wall
// is found, if there are no larger walls
// then delete temp value from water
int temp = 0;
for (int i = 1; i <= size; i++) {

    // If the current wall is taller than
    // the previous wall then make current
    // wall as the previous wall and its
    // index as previous wall's index
    // for the subsequent loops
    if (arr[i] >= prev) {
        prev = arr[i];
        prev_index = i;

        // Because larger or same height wall is found
        temp = 0;
    }
    else {

        // Since current wall is shorter than
        // the previous, we subtract previous
        // wall's height from the current wall's
        // height and add it to the water
        water += prev - arr[i];

        // Store the same value in temp as well
        // If we dont find any larger wall then
        // we will subtract temp from water
        temp += prev - arr[i];
    }
}

// If the last wall was larger than or equal
// to the previous wall then prev_index would
// be equal to size of the array (last element)
// If we didn't find a wall greater than or equal
// to the previous wall from the left then
// prev_index must be less than the index
// of the last element
if (prev_index < size) {

    // Temp would've stored the water collected
    // from previous largest wall till the end
    // of array if no larger wall was found then
    // it has excess water and remove that
    // from 'water' var
    water -= temp;
```

```
            // We start from the end of the array, so previous
            // should be assigned to the last element
            prev = arr[size];

            // Loop from the end of array up to the 'previous
index'
            // which would contain the "largest wall from the
left"
            for (int i = size; i >= prev_index; i--) {

                // Right end wall will be definitely smaller
                // than the 'previous index' wall
                if (arr[i] >= prev) {
                    prev = arr[i];
                }
                else {
                    water += prev - arr[i];
                }
            }
        }

        // Return the maximum water
        return water;
    }

    // Driver code
    public static void main(String[] args)
    {
        int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };
        int n = arr.length;
        System.out.print(maxWater(arr, n));
    }
}
```

Output:

```
6
```

Time Complexity: O(n)
Auxiliary Space: O(1)

30. Count the number of possible triangles

Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).
For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.
As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Brute force)
The brute force method is to run three loops and keep track of the number of triangles possible so far. The three loops select three different values from array, the innermost loop checks for the triangle property (the sum of any two sides must be greater than the value of third side).
Time Complexity: O(N^3) where N is the size of input array.

Method 2 (Tricky and Efficient)
Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)
i) a + b > c
ii) b + c > a
iii) a + c > b
Following are steps to count triangle.

1. Sort the array in non-decreasing order.
2. Initialize two pointers 'i' and 'j' to first and second elements respectively, and initialize count of triangles as 0.
3. Fix 'i' and 'j' and find the rightmost index 'k' (or largest 'arr[k]') such that 'arr[i] + arr[j] > arr[k]'. The number of triangles that can be formed with 'arr[i]' and 'arr[j]' as two sides is 'k – j'. Add 'k – j' to count of triangles.
Let us consider 'arr[i]' as 'a', 'arr[j]' as b and all elements between 'arr[j+1]' and 'arr[k]' as 'c'. The above mentioned conditions (ii) and (iii) are satisfied because 'arr[i] < arr[j] < arr[k]'. And we check for condition (i) when we pick 'k'

4. Increment 'j' to fix the second element again.
Note that in step 3, we can use the previous value of 'k'. The reason is simple, if we know that the value of 'arr[i] + arr[j-1]' is greater than 'arr[k]', then we can say 'arr[i] + arr[j]' will also be greater than 'arr[k]', because the array is sorted in increasing order.

5. If 'j' has reached end, then increment 'i'. Initialize 'j' as 'i + 1', 'k' as 'i+2' and repeat the steps 3 and 4.
Following is implementation of the above approach.

```cpp
// C++ program to count number of triangles that can be
// formed from given array
#include <bits/stdc++.h>
using namespace std;

/* Following function is needed for library function
qsort(). Refer
http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int comp(const void* a, const void* b)
{ return *(int*)a > *(int*)b ; }

// Function to count all possible triangles with arr[]
// elements
int findNumberOfTriangles(int arr[], int n)
{
    // Sort the array elements in non-decreasing order
    qsort(arr, n, sizeof( arr[0] ), comp);

    // Initialize count of triangles
    int count = 0;

    // Fix the first element. We need to run till n-3
    // as the other two elements are selected from
    // arr[i+1...n-1]
    for (int i = 0; i < n - 2; ++i)
    {
        // Initialize index of the rightmost third
        // element
        int k = i + 2;

        // Fix the second element
        for (int j = i + 1; j < n; ++j)
        {
            // Find the rightmost element which is
            // smaller than the sum of two fixed elements
            // The important thing to note here is, we
            // use the previous value of k. If value of
            // arr[i] + arr[j-1] was greater than arr[k],
            // then arr[i] + arr[j] must be greater than k,
            // because the array is sorted.
            while (k < n && arr[i] + arr[j] > arr[k])
            ++k;

            // Total number of possible triangles that can
            // be formed with the two fixed elements is
            // k - j - 1. The two fixed elements are arr[i]
            // and arr[j]. All elements between arr[j+1]/ to
            // arr[k-1] can form a triangle with arr[i] and
arr[j].
            // One is subtracted from k because k is incremented
```

```
            // one extra in above while loop.
            // k will always be greater than j. If j becomes
equal
            // to k, then above loop will increment k, because
arr[k]
            // + arr[i] is always greater than arr[k]
            if(k > j)
            count += k - j - 1;
        }
    }

    return count;
}

// Driver code
int main()
{
    int arr[] = {10, 21, 22, 100, 101, 200, 300};
    int size = sizeof( arr ) / sizeof( arr[0] );

    cout<<"Total number of triangles possible is
"<<findNumberOfTriangles( arr, size );

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Total number of triangles possible is 6
```

Time Complexity: O(n^2). The time complexity looks more because of 3 nested loops. If we take a closer look at the algorithm, we observe that k is initialized only once in the outermost loop. The innermost loop executes at most O(n) time for every iteration of outer most loop, because k starts from i+2 and goes upto n for all values of j. Therefore, the time complexity is O(n^2).

Two Pointer Technique:

The time complexity can be greatly reduced using Two pointer method in just two nested loops.

Lets suppose the array given is {4,3,5,7,6}. On sorting the array becomes {3,4,5,6,7}.

We take three variables l ,r and i, pointing to start , end-1 and array element starting
from end of the array.

Now if we can form triange using arr[l] and arr[r] then we can obviously form triangles
from a[l+1],a[l+2].....a[r-1], arr[r] and a[i], because the array is sorted
which we can directly calculate using (r-l). So the overall complexity of iterating through all array elements reduces.

Below is the implementation of the above approach:

```cpp
// C++ implementation of the above approach
#include<bits/stdc++.h>
using namespace std;

void CountTriangles(vector <int> A){

    int n=A.size();

    sort(A.begin(), A.end());

    int count = 0;

    for( int i = n-1; i >= 1; i--){
        int l = 0, r = i-1;
        while( l < r){
            if( A[l] + A[r] > A[i]){

                // If it is possible with a[l], a[r]
                // and a[i] then it is also possible
                // with a[l+1]..a[r-1],a[r] and a[i]
                count += r - l ;

                //checking for more possible solutions
                r--;
            }
            else

                // if not possible check for
                // higher values of arr[l]
                l++;
        }
    }
    cout<<"No of possible solutions: "<<count;
}
int main(){

    vector<int> A = { 4, 3, 5, 7, 6 };

    CountTriangles(A);

}
```

Output:

```
No of possible solutions: 9
```

Time complexity: O(n^2), because we use two nested loops , but overall iterations in comparison to
above method reduces greatly.

31. Count number of unique Triangles using STL | Set 1 (Using set)

We are given n triangles along with length of their three sides as a,b,c. Now we need to count number of unique triangles out of these n given triangles. Two triangles are different from one another if they have at least one of the sides different.

Example:

```
Input: arr[] = {{1, 2, 2},
               {4, 5, 6},
               {4, 5, 6}
Output: 2


Input: arr[] = {{4, 5, 6}, {6, 5, 4},
               {1, 2, 2}, {8, 9, 12}
Output: 3
```

We strongly recommend you to minimize your browser and try this yourself first.

Since we are asked to find number of “unique” triangles, we can use either set or unordered_set. In this post, set based approach is discussed. How to store three sides as an element in the container? We use STL pair to store all the three sides together as

```
pair <int, pair<int, int> >
```

We one by one insert all triangles into the set. But the problem with this approach is that a triangle with sides as {4, 5, 6} is different from a triangle with sides {5, 4, 6} although they refer to a same triangle.

In order to handle such cases, we store sides in sorted order (on the basis of length of sides), here sorting won’t be an issue since we have only 3 sides and we can sort them in constant time. For example {5, 4, 6} is inserted into set as {4, 5, 6}

Note : We can make pair either by make_pair(a,b) or we can simply use {a, b}.

Below is C++ implementation of the above idea:

```
// A C++ program to find number of unique Triangles
#include <bits/stdc++.h>
using namespace std;

// Creating shortcut for an integer pair.
typedef pair<int, int> iPair;

// A structure to represent a Triangle with
// three sides as a, b, c
```

```
struct Triangle
{
    int a, b, c;
};

// A function to sort three numbers a, b and c.
// This function makes 'a' smallest, 'b' middle
// and 'c' largest (Note that a, b and c are passed
// by reference)
int sort3(int &a, int &b, int &c)
{
    if (a > b) swap(a, b);
    if (b > c) swap(b, c);
    if (a > b) swap(a, b);
}

// Function returns the number of unique Triangles
int countUniqueTriangles(struct Triangle arr[],
                         int n)
{
    // A set which consists of unique Triangles
    set < pair< int, iPair > > s;

    // Insert all triangles one by one
    for (int i=0; i<n; i++)
    {
        // Find three sides and sort them
        int a = arr[i].a, b = arr[i].b, c = arr[i].c;
        sort3(a, b, c);

        // Insert a triangle into the set
        s.insert({a, {b, c}});
    }

    // Return set size
    return s.size();
}

// Driver program to test above function
int main()
{
    // An array of structure to store sides of 6 Triangles
    struct Triangle arr[] = {{3, 2, 2}, {3, 4, 5}, {1, 2, 2},
                             {2, 2, 3}, {5, 4, 3}, {6, 4, 5}};
    int n = sizeof(arr)/sizeof(Triangle);

    cout << "Number of Unique Triangles are "
         << countUniqueTriangles(arr, n);
    return 0;
}
```

Output:

```
Number of Unique Triangles are 4
```

The time complexity of above solution is O(n Log n) as set requires O(Log n) time for insertion.

32. Rearrange an array so that arr[i] becomes arr[arr[i]] with O(1) extra space

Given an array arr[] of size n where every element is in range from 0 to n-1. Rearrange the given array so that arr[i] becomes arr[arr[i]]. This should be done with O(1) extra space.
Examples:

```
Input: arr[]  = {3, 2, 0, 1}
Output: arr[] = {1, 0, 3, 2}

Input: arr[] = {4, 0, 2, 1, 3}
Output: arr[] = {3, 4, 2, 0, 1}

Input: arr[] = {0, 1, 2, 3}
Output: arr[] = {0, 1, 2, 3}
```

If the extra space condition is removed, the question becomes very easy. The main part of the question is to do it without extra space.

We strongly recommend that you click here and practice it, before moving on to the solution.

The credit for following solution goes to Ganesh Ram Sundaram . Following are the steps.
1) Increase every array element arr[i] by (arr[arr[i]] % n)*n.
2) Divide every element by n.

```
Let us understand the above steps by an example array {3, 2, 0, 1}
In first step, every value is incremented by (arr[arr[i]] % n)*n
After first step array becomes {7, 2, 12, 9}.
The important thing is, after the increment operation
of first step, every element holds both old values and new values.
Old value can be obtained by arr[i]%n and new value can be obtained
by arr[i]/n.

In second step, all elements are updated to new or output values
by doing arr[i] = arr[i]/n.
After second step, array becomes {1, 0, 3, 2}
```

Following is the implementation of the above approach.

```cpp
#include <iostream>
using namespace std;

// The function to rearrange an array in-place so that arr[i]
// becomes arr[arr[i]].
void rearrange(int arr[], int n)
{
    // First step: Increase all values by (arr[arr[i]]%n)*n
    for (int i=0; i < n; i++)
        arr[i] += (arr[arr[i]]%n)*n;

    // Second Step: Divide all values by n
    for (int i=0; i<n; i++)
        arr[i] /= n;
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}


/* Driver program to test above functions*/
int main()
{
    int arr[] = {3, 2, 0, 1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Given array is \n";
    printArr(arr, n);

    rearrange(arr, n);

    cout << "Modified array is \n";
    printArr(arr, n);
    return 0;
}
```

Output:

```
Given array is

3 2 0 1

Modified array is

1 0 3 2
```

Time Complexity: O(n)
Auxiliary Space: O(1)

The problem with above solution is, it may cause overflow.

33. Inplace rotate square matrix by 90 degrees | Set 1

Given an square matrix, turn it by 90 degrees in anti-clockwise direction without using any extra space.

Examples :

```
Input
 1  2  3
 4  5  6
 7  8  9

Output:
 3  6  9
 2  5  8
 1  4  7

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

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

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

An approach that requires extra space is already discussed here.

How to do without extra space?
Below are some important observations.
First row of source –> First column of destination, elements filled in opposite order
Second row of source –> Second column of destination, elements filled in opposite order

so ... on

Last row of source –> Last column of destination, elements filled in opposite order.

An N x N matrix will have floor(N/2) square cycles. For example, a 4 X 4 matrix will have 2 cycles. The first cycle is formed by its 1st row, last column, last row and 1st column. The second cycle is formed by 2nd row, second-last column, second-last row and 2nd column.

The idea is for each square cycle, we swap the elements involved with the corresponding cell in the matrix in anti-clockwise direction i.e. from top to left, left to bottom, bottom to right and from right to top one at a time. We use nothing but a temporary variable to achieve this.

Below steps demonstrate the idea

```
First Cycle (Involves Red Elements)
 1  2  3 4
 5  6  7  8
 9 10 11 12
 13 14 15 16



Moving first group of four elements (First

elements of 1st row, last row, 1st column

and last column) of first cycle in counter

clockwise.

 4  2  3 16
 5  6  7 8

 9 10 11 12

 1 14  15 13


Moving next group of four elements of

first cycle in counter clockwise

 4  8  3 16
 5  6  7  15
 2  10 11 12
 1  14  9 13


Moving final group of four elements of

first cycle in counter clockwise

 4  8 12 16
 3  6  7 15
 2 10 11 14
 1  5  9 13



Second Cycle (Involves Blue Elements)
 4  8 12 16

 3  6 7  15
```

```
 2  10 11 14
 1  5  9 13


Fixing second cycle
 4  8 12 16

 3  7 11 15
 2  6 10 14
 1  5  9 13
```

Below is the implementation of above idea.

```
// C++ program to rotate a matrix by 90 degrees
#include <bits/stdc++.h>
#define N 4
using namespace std;

void displayMatrix(int mat[N][N]);

// An Inplace function to rotate a N x N matrix
// by 90 degrees in anti-clockwise direction
void rotateMatrix(int mat[][N])
{
    // Consider all squares one by one
    for (int x = 0; x < N / 2; x++)
    {
        // Consider elements in group of 4 in
        // current square
        for (int y = x; y < N-x-1; y++)
        {
            // store current cell in temp variable
            int temp = mat[x][y];

            // move values from right to top
            mat[x][y] = mat[y][N-1-x];

            // move values from bottom to right
            mat[y][N-1-x] = mat[N-1-x][N-1-y];

            // move values from left to bottom
            mat[N-1-x][N-1-y] = mat[N-1-y][x];

            // assign temp to left
            mat[N-1-y][x] = temp;
        }
    }
}
```

```c
// Function to print the matrix
void displayMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%2d ", mat[i][j]);

        printf("\n");
    }
    printf("\n");
}


/* Driver program to test above functions */
int main()
{
    // Test Case 1
    int mat[N][N] =
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };


    // Tese Case 2
    /* int mat[N][N] = {
                        {1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}
                    };
     */

    // Tese Case 3
    /*int mat[N][N] = {
                    {1, 2},
                    {4, 5}
                };*/

    //displayMatrix(mat);

    rotateMatrix(mat);

    // Print rotated matrix
    displayMatrix(mat);

    return 0;
```

```
}
```

Output :

```
 4  8 12 16

 3  7 11 15

 2  6 10 14

 1  5  9 13
```

Exercise: Turn 2D matrix by 90 degrees in clockwise direction without using extra space

34. Find position of an element in a sorted array of infinite numbers

Suppose you have a sorted array of infinite numbers, how would you search an element in the array?

Source: Amazon Interview Experience.

Since array is sorted, the first thing clicks into mind is binary search, but the problem here is that we don't know size of array.
If the array is infinite, that means we don't have proper bounds to apply binary search. So in order to find position of key, first we find bounds and then apply binary search algorithm.

Let low be pointing to 1st element and high pointing to 2nd element of array, Now compare key with high index element,
->if it is greater than high index element then copy high index in low index and double the high index.
->if it is smaller, then apply binary search on high and low indices found.

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

Below are implementations of above algorithm

```
// C++ program to demonstrate working of an algorithm that finds
// an element in an array of infinite size
#include<bits/stdc++.h>
using namespace std;

// Simple binary search algorithm
int binarySearch(int arr[], int l, int r, int x)
{
    if (r>=l)
    {
        int mid = l + (r - l)/2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);
```

```
        return binarySearch(arr, mid+1, r, x);
    }
    return -1;
}

// function takes an infinite size array and a key to be
//  searched and returns its position if found else -1.
// We don't know size of arr[] and we can assume size to be
// infinite in this function.
// NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE
// THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING
int findPos(int arr[], int key)
{
    int l = 0, h = 1;
    int val = arr[0];

    // Find h to do binary search
    while (val < key)
    {
        l = h;         // store previous high
        h = 2*h;       // double high index
        val = arr[h]; // update new val
    }

    // at this point we have updated low and
    // high indices, Thus use binary search
    // between them
    return binarySearch(arr, l, h, key);
}

// Driver program
int main()
{
    int arr[] = {3, 5, 7, 9, 10, 90, 100, 130,
                                  140, 160, 170};
    int ans = findPos(arr, 10);
    if (ans==-1)
        cout << "Element not found";
    else
        cout << "Element found at index " << ans;
    return 0;
}
```

Output:

```
Element found at index 4
```

Let p be the position of element to be searched. Number of steps for finding high index ‘h’ is O(Log p). The value of ‘h’ must be less than 2*p. The number

of elements between h/2 and h must be O(p). Therefore, time complexity of Binary Search step is also O(Log p) and overall time complexity is 2*O(Log p) which is O(Log p).