# Arrays

### Reverse the array

You are given a string s. You need to reverse the string.

Example 1:

Input:

s = Geeks

Output: skeeG

Example 2:

Input:

s = for

Output: rof

Your Task:

You only need to complete the function reverseWord() that takes s as parameter and returns the reversed string.

Expected Time Complexity: O(|S|).  
Expected Auxiliary Space: O(1).

Constraints:  
1 <= |s| <= 10000

## string reverseWord(string str){

## 

## //Your code here

## 

## int n = str.length();

## 

## for(int i=0; i<n/2; i++){

## int t = str[i];

## str[i] = str[n-i-1];

## str[n-i-1] = t;

## }

## 

## return str;

## }

### Maximum and minimum of an array using minimum number of comparisons

## Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.

## Pair MaxMin(array, array\_size)

## if array\_size = 1

## return element as both max and min

## else if arry\_size = 2

## one comparison to determine max and min

## return that pair

## else /\* array\_size > 2 \*/

## recur for max and min of left half

## recur for max and min of right half

## one comparison determines true max of the two candidates

## one comparison determines true min of the two candidates

## return the pair of max and min

|  |
| --- |
| // C++ program of above implementation#include<iostream>using namespace std;// structure is used to return// two values from minMax()struct Pair{    int min;    int max;};struct Pair getMinMax(int arr[], int low,                                 int high){    struct Pair minmax, mml, mmr;    int mid;    // If there is only one element    if (low == high)    {        minmax.max = arr[low];        minmax.min = arr[low];        return minmax;    }    // If there are two elements    if (high == low + 1)    {        if (arr[low] > arr[high])        {            minmax.max = arr[low];            minmax.min = arr[high];        }        else        {            minmax.max = arr[high];            minmax.min = arr[low];        }        return minmax;    }    // If there are more than 2 elements    mid = (low + high) / 2;    mml = getMinMax(arr, low, mid);    mmr = getMinMax(arr, mid + 1, high);    // Compare minimums of two parts    if (mml.min < mmr.min)        minmax.min = mml.min;    else        minmax.min = mmr.min;    // Compare maximums of two parts    if (mml.max > mmr.max)        minmax.max = mml.max;    else        minmax.max = mmr.max;    return minmax;}// Driver codeint main(){    int arr[] = { 1000, 11, 445,                  1, 330, 3000 };    int arr\_size = 6;    struct Pair minmax = getMinMax(arr, 0,                             arr\_size - 1);    cout << "Minimum element is "         << minmax.min << endl;    cout << "Maximum element is "         << minmax.max;    return 0;}// This code is contributed by nik\_3112 |

## Output:

## Minimum element is 1

## Maximum element is 3000

## Time Complexity: O(n)

## Total number of comparisons: let the number of comparisons be T(n). T(n) can be written as follows:  Algorithmic Paradigm: Divide and Conquer

## 

## T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2

## T(2) = 1

## T(1) = 0

## If n is a power of 2, then we can write T(n) as:

## 

## T(n) = 2T(n/2) + 2

## After solving the above recursion, we get

## T(n) = 3n/2 -2

## Thus, the approach does 3n/2 -2 comparisons if n is a power of 2. And it does more than 3n/2 -2 comparisons if n is not a power of 2.

### Find the Kth max and min element in an array

Given an array arr[] and an integer K where K is smaller than size of array, the task is to find the Kth smallest element in the given array. It is given that all array elements are distinct.

Example 1:

Input:

N = 6

arr[] = 7 10 4 3 20 15

K = 3

Output : 7

Explanation :

3rd smallest element in the given

array is 7.

Example 2:

Input:

N = 5

arr[] = 7 10 4 20 15

K = 4

Output : 15

Explanation :

4th smallest element in the given

array is 15.

Your Task:  
You don't have to read input or print anything. Your task is to complete the function kthSmallest() which takes the array arr[], integers l and r denoting the starting and ending index of the array and an integer K as input and returns the Kth smallest element.

Expected Time Complexity: O(n)

Expected Auxiliary Space: O(1)

Constraints:  
1 <= N <= 105  
1 <= arr[i] <= 105  
1 <= K <= N

## In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the surrounding array. The idea is, not to do complete quicksort, but stop at the point where pivot itself is k’th smallest element. Also, not to recur for both left and right sides of pivot, but recur for one of them according to the position of pivot.

## For Optimization, randomly pick a pivot element. To implement randomized partition, we use a random function, [rand()](http://www.cplusplus.com/reference/cstdlib/rand/) to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

## class Solution{

## public:

## // arr : given array

## // l : starting index of the array i.e 0

## // r : ending index of the array i.e size-1

## // k : find kth smallest element and return using this function

## int kthSmallest(int arr[], int l, int r, int k) {

## 

## int x = partition(arr, l, r);

## 

## if(x==k-1)

## return arr[x];

## else if(x>k-1)

## return kthSmallest(arr, l, x-1, k);

## else

## return kthSmallest(arr, x+1, r, k);

## }

## 

## //Partition is done by selecting a random number

## //from array to minimize the chances of worst case

## int partition(int arr[], int l, int r){

## 

## int random\_index = (rand()%(r-l+1))+l;

## 

## //swap pivot with last element of array

## int temp = arr[random\_index];

## arr[random\_index] = arr[r];

## arr[r] = temp;

## 

## int pivot = arr[r], i = l-1;

## 

## for(int j=l;j<=r;j++){

## if(arr[j]<=pivot){

## i++;

## int t = arr[i];

## arr[i] = arr[j];

## arr[j] = t;

## }

## }

## return i;

## }

## };

### [Given an array which consists of only 0, 1 and 2. Sort the array without using any sorting algo](https://practice.geeksforgeeks.org/problems/sort-an-array-of-0s-1s-and-2s/0)

Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order.

**Example 1:**

**Input:**

N = 5

arr[]= {0 2 1 2 0}

**Output:**

0 0 1 2 2

**Explanation:**

0s 1s and 2s are segregated

into ascending order.

**Example 2:**

**Input:**

N = 3

arr[] = {0 1 0}

**Output:**

0 0 1

**Explanation:**

0s 1s and 2s are segregated

into ascending order.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **sort012()**that takes an array arr and N as input parameters and **sorts the array in-place.**

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= N <= 10^6  
0 <= A[i] <= 2

## void sort012(int a[], int n)

## {

## // coode here

## int zero = 0, one =0, two = 0;

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

## switch(a[i]){

## case 0:

## zero++;

## break;

## case 1:

## one++;

## break;

## case 2:

## two++;

## break;

## }

## }

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

## if(zero!=0){

## a[i] = 0;

## zero--;

## }

## else if(one!=0){

## a[i] = 1;

## one--;

## }

## else{

## a[i] = 2;

## two--;

## }

## }

## }

### [Move all the negative elements to one side of the array](https://www.geeksforgeeks.org/move-negative-numbers-beginning-positive-end-constant-extra-space/)

An array contains both positive and negative numbers in random order. Rearrange the array elements so that all negative numbers appear before all positive numbers.

**Examples :**

**Input:** -12, 11, -13, -5, 6, -7, 5, -3, -6

**Output:** -12 -13 -5 -7 -3 -6 11 6 5

**Note:** Order of elements is not important here.

## **Approach 1:** The idea is to simply applythe [partition process of quicksort](https://www.geeksforgeeks.org/quick-sort/).

// A C++ program to put all negative

// numbers before positive numbers

#include <bits/stdc++.h>

using namespace std;

void rearrange(int arr[], int n)

{

    int j = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] < 0) {

            if (i != j)

                swap(arr[i], arr[j]);

            j++;

        }

    }

}

// A utility function to print an array

void printArray(int arr[], int n)

{

    for (int i = 0; i < n; i++)

        printf("%d ", arr[i]);

}

// Driver code

int main()

{

    int arr[] = { -1, 2, -3, 4, 5, 6, -7, 8, 9 };

    int n = sizeof(arr) / sizeof(arr[0]);

    rearrange(arr, n);

    printArray(arr, n);

    return 0;

}

## **Time complexity:** O(N)  **Auxiliary Space:** O(1)

## Two Pointer Approach: The idea is to solve this problem with constant space and linear time is by using a [two-pointer](https://www.geeksforgeeks.org/two-pointers-technique/)or two-variable approach where we simply take two variables like left and right which hold the 0 and N-1 indexes. Just need to check that :

## Check If the left and right pointer elements are negative then simply increment the left pointer.

## Otherwise, if the left element is positive and the right element is negative then simply swap the elements, and simultaneously increment and decrement the left and right pointers.

## Else if the left element is positive and the right element is also positive then simply decrement the right pointer.

## Repeat the above 3 steps until the left pointer ≤ right pointer.

// C++ program of the above

// approach

#include <iostream>

using namespace std;

// Function to shift all the

// negative elements on left side

void shiftall(int arr[], int left,

              int right)

{

  // Loop to iterate over the

  // array from left to the right

  while (left<=right)

  {

    // Condition to check if the left

    // and the right elements are

    // negative

    if (arr[left] < 0 && arr[right] < 0)

      left+=1;

    // Condition to check if the left

    // pointer element is positive and

    // the right pointer element is negative

    else if (arr[left]>0 && arr[right]<0)

    {

      int temp=arr[left];

      arr[left]=arr[right];

      arr[right]=temp;

      left+=1;

      right-=1;

    }

    // Condition to check if both the

    // elements are positive

    else if (arr[left]>0 && arr[right] >0)

      right-=1;

    else{

      left += 1;

      right -= 1;

    }

  }

}

// Function to print the array

void display(int arr[], int right){

  // Loop to iterate over the element

  // of the given array

  for (int i=0;i<=right;++i){

    cout<<arr[i]<<" ";

  }

  cout<<endl;

}

// Driver Code

int main()

{

  int arr[] = {-12, 11, -13, -5,

               6, -7, 5, -3, 11};

  int arr\_size = sizeof(arr) /

                sizeof(arr[0]);

  // Function Call

  shiftall(arr,0,arr\_size-1);

  display(arr,arr\_size-1);

  return 0;

}

## This is an in-place rearranging algorithm for arranging the positive and negative numbers where the order of elements is not maintained.

## **Time Complexity:** O(N) **Auxiliary Space:**O(1)

### [Find the Union and Intersection of the two arrays. (Elements in arrays may not be distinct but in union all elements should be distinct)](https://practice.geeksforgeeks.org/problems/union-of-two-arrays/0)

Given two arrays **a[]** and **b[]** of size **n** and **m** respectively. The task is to find union between these two arrays.

Union of the two arrays can be defined as the set containing distinct elements from both the arrays. If there are repetitions, then only one occurrence of element should be printed in the union.

**Example 1:**

**Input:**

5 3

1 2 3 4 5

1 2 3

**Output:**

5

**Explanation:**

1, 2, 3, 4 and 5 are the

elements which comes in the union set

of both arrays. So count is 5.

**Example 2:**

**Input:**

6 2

85 25 1 32 54 6

85 2

**Output:**

7

**Explanation:**

85, 25, 1, 32, 54, 6, and

2 are the elements which comes in the

union set of both arrays. So count is 7.

**Your Task:**  
Complete **doUnion**funciton that takes**a, n, b, m as parameters and returns** the count of union elements of the two arrays. The **printing**is done by the **driver**code.

**Constraints:**  
1 ≤ n, m ≤ 105  
0 ≤ a[i], b[i] < 105

**Expected Time Complexity**: O((n+m)log(n+m))  
**Expected Auxilliary Space** : O(1)

## int doUnion(int a[], int n, int b[], int m) {

## sort(a,a+n);

## sort(b,b+m);

## 

## int i=0,j=0,count=0,prev=-1;

## 

## while(i!=n&&j!=m){

## 

## if(a[i]<b[j]){

## if(a[i]!=prev){

## prev = a[i];

## count++;

## }

## i++;

## }

## else if(b[j]<a[i]){

## if(b[j]!=prev){

## prev = b[j];

## count++;

## }

## j++;

## }

## else{

## if(a[i]!=prev){

## prev = a[i];

## count++;

## }

## i++;

## j++;

## }

## }

## 

## if(i==n){

## while(j!=m){

## if(b[j]!=prev){

## prev = b[j];

## count++;

## }

## j++;

## }

## }

## else if(j==m){

## while(i!=n){

## if(a[i]!=prev){

## prev = a[i];

## count++;

## }

## i++;

## }

## }

## return count;

## }

### Rotate an array of n elements by d position

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

METHOD 1 (Using temp array)

Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n =7

1) Store the first 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 :

|  |
| --- |
| // 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[n-1] = 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)

METHOD3

void rotate(vector<int>& arr, int k) {

k=k%arr.size();

reverse(arr.begin(),arr.end());

reverse(arr.begin(),arr.begin()+k);

reverse(arr.begin()+k,arr.end());

}

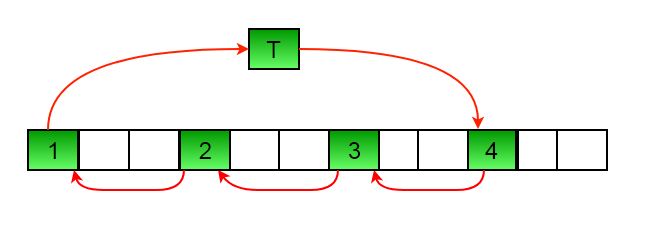
Time complexity : O(n)   
Auxiliary Space : O(1)

METHOD 4 (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;    /\*Function 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)  {      /\* To handle if d >= n \*/      d = d % 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;  } |

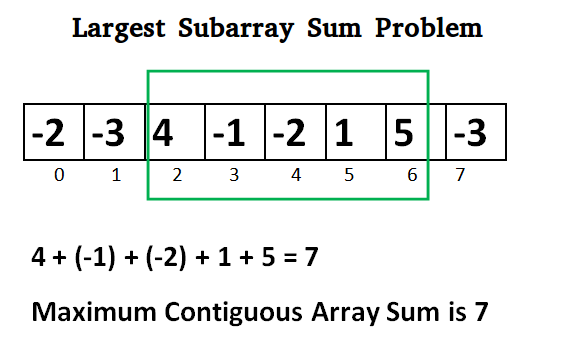
Output :

3 4 5 6 7 1 2

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

### [find Largest sum contiguous Subarray [V. IMP]](https://practice.geeksforgeeks.org/problems/kadanes-algorithm/0)

Write an efficient program to find the sum of contiguous subarray within a one-dimensional array of numbers that has the largest sum.



**Kadane’s Algorithm:**

Initialize:

max\_so\_far = INT\_MIN

max\_ending\_here = 0

Loop for each element of the array

(a) max\_ending\_here = max\_ending\_here + a[i]

(b) if(max\_so\_far < max\_ending\_here)

max\_so\_far = max\_ending\_here

(c) if(max\_ending\_here < 0)

max\_ending\_here = 0

return max\_so\_far

**Explanation:**   
The simple idea of Kadane’s algorithm is to look for all positive contiguous segments of the array (max\_ending\_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max\_so\_far is used for this). Each time we get a positive-sum compare it with max\_so\_far and update max\_so\_far if it is greater than max\_so\_far

// C++ program to print largest contiguous array sum

#include<iostream>

#include<climits>

using namespace std;

int maxSubArraySum(int a[], int size)

{

int max\_so\_far = INT\_MIN, max\_ending\_here = 0;

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

{

max\_ending\_here = max\_ending\_here + a[i];

if (max\_so\_far < max\_ending\_here)

max\_so\_far = max\_ending\_here;

if (max\_ending\_here < 0)

max\_ending\_here = 0;

}

return max\_so\_far;

}

/\*Driver program to test maxSubArraySum\*/

int main()

{

int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};

int n = sizeof(a)/sizeof(a[0]);

int max\_sum = maxSubArraySum(a, n);

cout << "Maximum contiguous sum is " << max\_sum;

return 0;

}

### [Minimise the maximum difference between heights [V.IMP]](https://practice.geeksforgeeks.org/problems/minimize-the-heights3351/1)

Given an array **arr[]** denoting heights of **N** towers and a positive integer **K**, you **have to** modify the height of each tower either by increasing or decreasing them by **K** only **once**. After modifying, height should be a **non-negative** integer.   
Find out what could be the possible minimum difference of the height of shortest and longest towers after you have modified each tower.

A slight modification of the problem can be found [here](https://practice.geeksforgeeks.org/problems/minimize-the-heights-i/1/).

**Example 1:**

**Input:**

K = 2, N = 4

Arr[] = {1, 5, 8, 10}

**Output:**

5

**Explanation:**

The array can be modified as

{3, 3, 6, 8}. The difference between

the largest and the smallest is 8-3 = 5.

**Example 2:**

**Input:**

K = 3, N = 5

Arr[] = {3, 9, 12, 16, 20}

**Output:**

11

**Explanation:**

The array can be modified as

{6, 12, 9, 13, 17}. The difference between

the largest and the smallest is 17-6 = 11.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **getMinDiff()** which takes the **arr[], n** and **k**as input parameters and returns an integer denoting the minimum difference.

**Expected Time Complexity:** O(N\*logN)  
**Expected Auxiliary Space:** O(N)  
  
**Constraints**  
1 ≤ K ≤ 104  
1 ≤ N ≤ 105  
1 ≤ Arr[i] ≤ 105

## int getMinDiff(int arr[], int n, int k) {

## sort(arr,arr+n);

## int ans = arr[n-1]-arr[0];

## /\*ans is intialized with arr[n-1]-arr[0] to handle the condition

## if arr[n-1]-arr[0]<k bcz in this condition (arr[n-1]+k)-(arr[0]+k) or (arr[n-1]-k)-(arr[0]-k) is less than abs((arr[n-1]-k)-(arr[0]+k))\*/

## int smallest = arr[0] + k;

## int largest = arr[n-1] - k;

## int mi,ma;

## /\*Note that if smallest>largest then it will be automatically handled in for loop\*/

## /\*For minimum difference, we always decrease the value of larger value and increase the value of smaller value. Since, we have already sorted the array so arr[i]<arr[i+1].Hence, to minimize the difference, we increase arr[i] by k and decrease the value of arr[i+1] by k.\*/

## for(int i=0;i<n-1;i++){

## mi = min(smallest,arr[i+1]-k);

## ma = max(largest,arr[i]+k);

## if(mi<0) continue;

## ans = min(ans,ma-mi);

## }

## return ans;

## }

### Each element represents max no. of steps which can be taken forward from it. Find minimum no. of Jumps to reach end of an array.

Given an array of N integers arr[] where each element represents the max number of steps that can be made forward from that element. Find the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then you cannot move through that element.  
Note: Return -1 if you can't reach the end of the array.

Example 1:

Input:

N = 11

arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}

Output: 3

Explanation:

First jump from 1st element to 2nd

element with value 3. Now, from here

we jump to 5th element with value 9,

and from here we will jump to last.

Example 2:

Input :

N = 6

arr = {1, 4, 3, 2, 6, 7}

Output: 2

Explanation:

First we jump from the 1st to 2nd element

and then jump to the last element.

Your task:  
You don't need to read input or print anything. Your task is to complete function minJumps() which takes the array arr and it's size N as input parameters and returns the minimum number of jumps. If not possible returns -1.

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

Constraints:  
1 ≤ N ≤ 107  
0 ≤ arri ≤ 107

## Solution:

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then we cannot move through that element. If we can't reach the end, return -1.  
**Examples:**

**Input:**  arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}

**Output:** 3 (1-> 3 -> 8 -> 9)

**Explanation:** Jump from 1st element to

2nd element as there is only 1 step,

now there are three options 5, 8 or 9.

If 8 or 9 is chosen then the end node 9

can be reached. So 3 jumps are made.

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

**Output:** 10

**Explanation:** In every step a jump is

needed so the count of jumps is 10.

In this post, its O(n) solution will be discussed.

**Implementation:**   
Variables to be used:

1. **maxReach** The variable maxReach stores at all time the maximal reachable index in the array.
2. **jump** jump stores the amount of jumps necessary to reach the maximal reachable position. It also indicates the **current jump we are making in the array**.
3. **step** The variable step stores the number of steps we can still take **in the current jump 'jump'** (and is initialized with value at index 0, i.e. initial number of steps)

Given array arr = 1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9

* **maxReach** = arr[0]; // arr[0] = 1, so the maximum index we can reach at the moment is 1.   
  **step** = arr[0]; // arr[0] = 1, the amount of steps we can still take is also 1.   
  **jump** = 1; // we are currently making our first jump.
* Now, starting iteration from index 1, the above values are updated as follows:
  1. First we test whether we have reached the end of the array, in that case we just need to return the jump variable.

if (i == arr.length - 1)

return jump;

2.  Next we update the maxReach. This is equal to the maximum of maxReach and i+arr[i](the number of steps we can take from the current position).

maxReach = Math.max(maxReach, i+arr[i]);

3.  We used up a step to get to the current index, so steps has to be decreased.

step--;

4. If no more steps are remaining (i.e. steps=0, then we must have used a jump. Therefore increase jump. Since we know that it is possible somehow to reach maxReach, we again initialize the steps to the number of steps to reach maxReach from position i. But before re-initializing step, we also check whether a step is becoming zero or negative. In this case, It is not possible to reach further.

if (step == 0) {

jump++;

if(i>=maxReach)

return -1;

step = maxReach - i;

}

// C++ program to count Minimum number

// of jumps to reach end

#include <bits/stdc++.h>

using namespace std;

int max(int x, int y)

{

return (x > y) ? x : y;

}

// Returns minimum number of jumps

// to reach arr[n-1] from arr[0]

int minJumps(int arr[], int n)

{

// The number of jumps needed to

// reach the starting index is 0

if (n <= 1)

return 0;

// Return -1 if not possible to jump

if (arr[0] == 0)

return -1;

// initialization

// stores all time the maximal

// reachable index in the array.

int maxReach = arr[0];

// stores the number of steps

// we can still take

int step = arr[0];

// stores the number of jumps

// necessary to reach that maximal

// reachable position.

int jump = 1;

// Start traversing array

int i = 1;

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

// Check if we have reached the end of the array

if (i == n - 1)

return jump;

// updating maxReach

maxReach = max(maxReach, i + arr[i]);

// we use a step to get to the current index

step--;

// If no further steps left

if (step == 0) {

// we must have used a jump

jump++;

// Check if the current index/position or lesser index

// is the maximum reach point from the previous indexes

if (i >= maxReach)

return -1;

// re-initialize the steps to the amount

// of steps to reach maxReach from position i.

step = maxReach - i;

}

}

return -1;

}

// Driver program to test above function

int main()

{

int arr[] = { 1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9 };

int size = sizeof(arr) / sizeof(int);

// Calling the minJumps function

cout << ("Minimum number of jumps to reach end is %d ",

minJumps(arr, size));

return 0;

}

Output

3

Complexity Analysis:

* Time complexity: O(n).   
  Only one traversal of the array is needed.
* Auxiliary Space: O(1).   
  There is no space required.

## My code:

int minJumps(int arr[], int n){

int current\_pos=0,count=0;

while(current\_pos<n-1){

int temp = arr[current\_pos];

int best\_val=0;

int best\_ind=0;

if(temp+current\_pos>=n-1)//Can reach the eand of array in single jump only now

current\_pos += temp;

else{

for(int i=1;i<=temp;i++){

if(current\_pos+i<=n-1){

if(arr[current\_pos+i]==0) //bcz this can never be best\_ind or best\_val

continue;

int t = i + arr[current\_pos+i];

if(t>best\_val){

best\_val = t;

best\_ind = i;

}

}

}

if(best\_val==0){ // cannot move anymore so can't reach the end of array

count=-1;

break;

}

current\_pos += best\_ind;

}

count++;

}

return count;

}

Complexity Analysis:

* Time complexity: O(n).   
  Only one traversal of the array is needed.
* Auxiliary Space: O(1).   
  There is no space required.

### find duplicate in an array of N+1 Integers which contains elements in range [1,N] only and there is only one repeated element.

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return *this repeated number*.

You must solve the problem without modifying the array nums and uses only constant extra space.

Example 1:

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

Output: 2

Example 2:

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

Output: 3

Example 3:

Input: nums = [1,1]

Output: 1

Example 4:

Input: nums = [1,1,2]

Output: 1

Constraints:

* 1 <= n <= 105
* nums.length == n + 1
* 1 <= nums[i] <= n
* All the integers in nums appear only once except for precisely one integer which appears two or more times.

Follow up:

* How can we prove that at least one duplicate number must exist in nums?

## Proving that at least one duplicate must exist in nums*nums* is an application of the [pigeonhole principle](https://en.wikipedia.org/wiki/Pigeonhole_principle). Here, each number in nums*nums* is a "pigeon" and each distinct number that can appear in nums*nums* is a "pigeonhole." Because there are n+1*n*+1 numbers and n*n* distinct possible numbers, the pigeonhole principle implies that if you were to put each of the n + 1*n*+1 pigeons into n*n* pigeonholes, at least one of the pigeonholes would have 2 or more pigeons.

* Can you solve the problem in linear runtime complexity?

#### Approach : Negative Marking

**Note:** This approach temporarily modifies individual elements and thus does not satisfy the problem constraints. However, this approach is intuitive and utilizes a technique that is useful to know. Furthermore, the underlying concept lends itself to solving similar problems.

**Intuition**

There are n + 1*n*+1 positive numbers in the array (nums*nums*) (all in the range [1, n][1,*n*]). Since the array only contains positive integers, we can track each number (num*num*) that has been seen before by flipping the sign of the number located at index |num|∣*num*∣, where ||∣∣ denotes absolute value.

For example, if the input array is [1, 3, 3, 2][1,3,3,2], then for 11, flip the number at index 11, making the array [1,-3,3,2][1,−3,3,2]. Next, for -3−3 flip the number at index 33, making the array [1,-3,3,-2][1,−3,3,−2]. Finally, when we reach the second 33, we'll notice that nums[3]*nums*[3] is already negative, indicating that 33 has been seen before and hence is the duplicate number.

**Algorithm**

1. Iterate over the array, evaluating each element (let's call the current element cur*cur*).
2. Since we use negative marking, we must ensure that the current element (cur*cur*) is positive (i.e. if cur*cur* is negative, then use its absolute value).
3. Check if nums[cur]*nums*[*cur*] is negative.
   * If it is, then we have already performed this operation for the same number, and hence *cur* is the duplicate number. Store *cur* as the duplicate and exit the loop.
   * Otherwise, flip the sign of *nums*[*cur*] (i.e. make it negative). Move to the next element and repeat step 3.
4. Once we've identified the duplicate, we could just return the duplicate number. However, even though we were not able to meet the problem constraints, we can show that we are mindful of the constraints by restoring the array. This is done by changing all negative numbers to positive.

class Solution {

public:

int findDuplicate(vector<int>& nums) {

int duplicate = -1;

for (int i = 0; i < nums.size(); i++) {

int cur = abs(nums[i]);

if (nums[cur] < 0) {

duplicate = cur;

break;

}

nums[cur] \*= -1;

}

// Restore numbers

for (auto& num : nums)

num = abs(num);

return duplicate;

}

};

**Complexity Analysis**

* Time Complexity: O(n)

Each element is visited at most twice (once in the first loop to find the duplicate and once in the second loop to restore the numbers).

* Space Complexity: *O*(1)

All manipulation is done in place, so no additional storage (barring one variable) is needed.

#### Approach : Floyd's Tortoise and Hare (Cycle Detection)

**Intuition**

The idea is to reduce the problem to [Linked List Cycle II](https://leetcode.com/problems/linked-list-cycle-ii/solution/):

Given a linked list, return the node where the cycle begins.

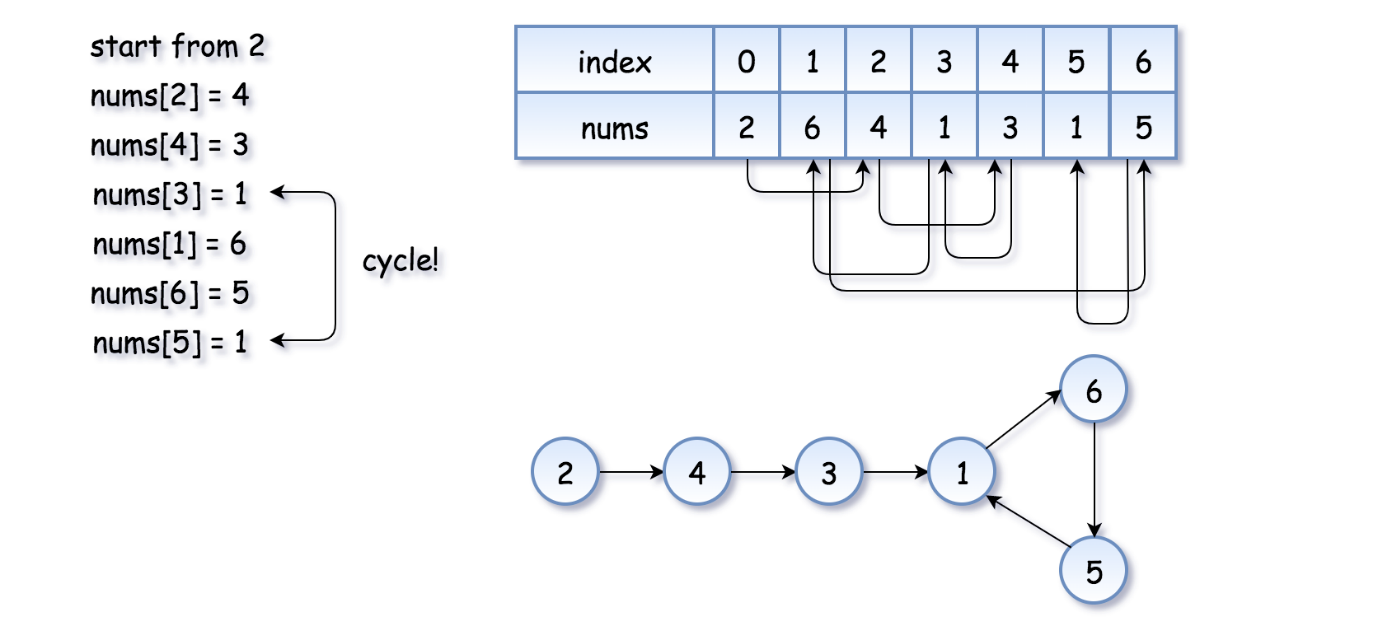
First of all, where does the cycle come from? Let's use the function f(x) = nums[x] to construct the sequence: x, nums[x], nums[nums[x]], nums[nums[nums[x]]], ....

Each new element in the sequence is an element in nums at the index of the previous element.

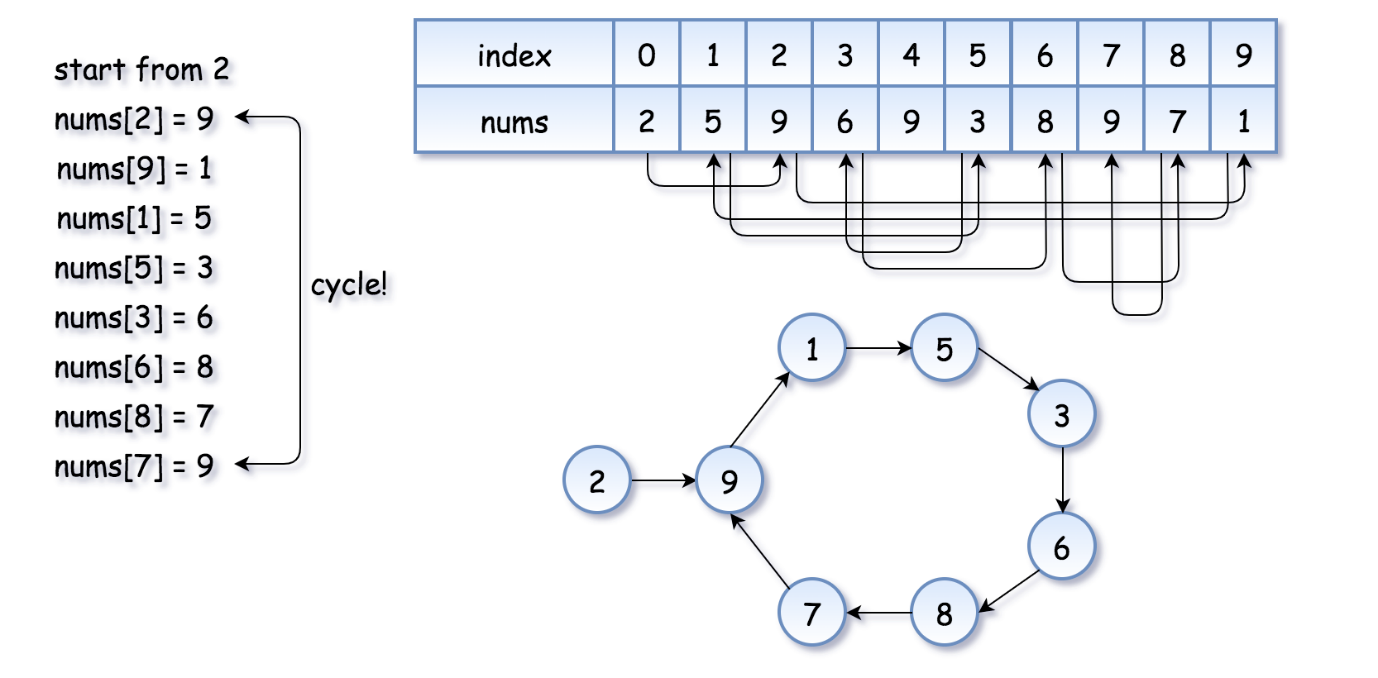
If one starts from x = nums[0], such a sequence will produce a linked list with a cycle.

The cycle appears because nums contains duplicates. The duplicate node is a cycle entrance.

Here is how it works:



The example above is simple because the loop is small. Here is a more interesting example (special thanks to @[sushant\_chaudhari](https://leetcode.com/sushant_chaudhari))



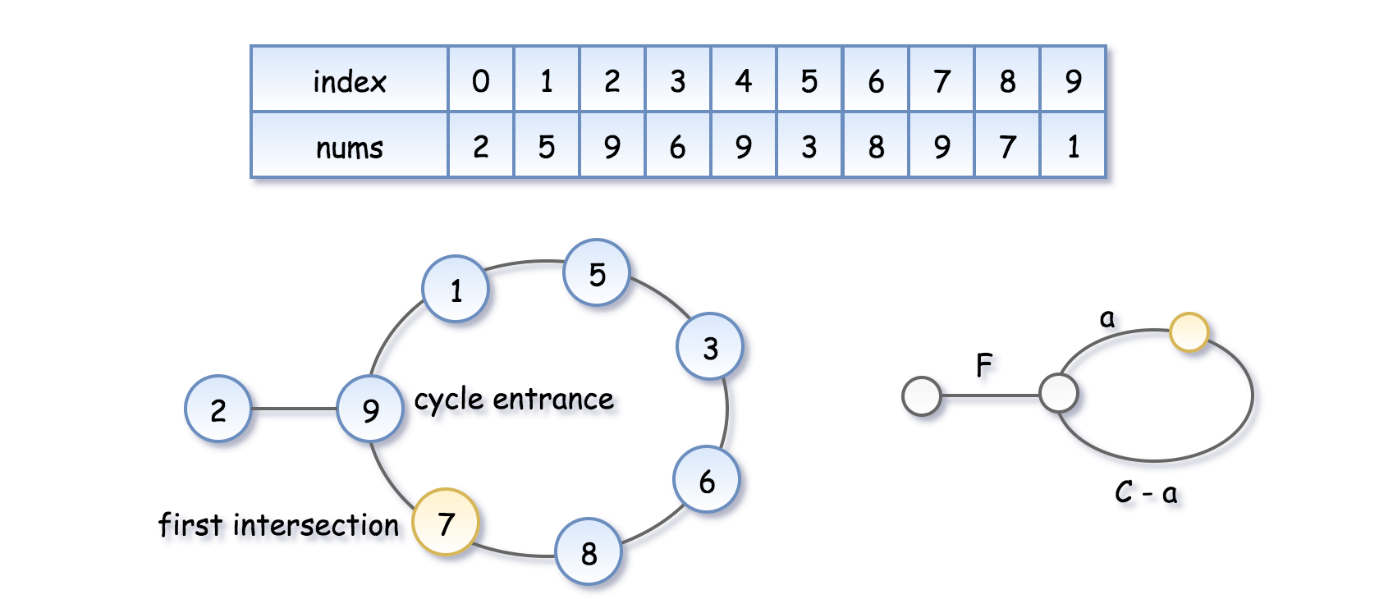
Now the problem is to find the entrance of the cycle.

**Algorithm**

[Floyd's algorithm](https://en.wikipedia.org/wiki/Cycle_detection#Tortoise_and_hare) consists of two phases and uses two pointers, usually called tortoise and hare.

**In phase 1**, hare = nums[nums[hare]] is twice as fast as tortoise = nums[tortoise]. Since the hare goes fast, it would be the first to enter the cycle and run around the cycle. At some point, the tortoise enters the cycle as well, and since it's moving slower the hare catches up to the tortoise at some intersection point. Now phase 1 is over, and the tortoise has lost.

Note that the intersection point is not the cycle entrance in the general case.

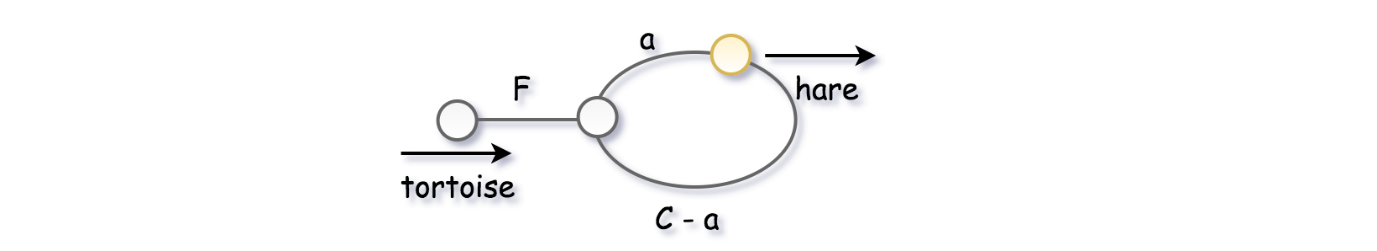


To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, i.e. 2*d*(tortoise)=*d*(hare), implying:

2(*F*+*a*)=*F*+*nC*+*a*, where n*n* is some integer.

Hence the coordinate of the intersection point is *F*+*a*=*nC*.

**In phase 2**, we give the tortoise a second chance by slowing down the hare, so that it now moves at the speed of tortoise: tortoise = nums[tortoise], hare = nums[hare]. The tortoise is back at the starting position, and the hare starts from the intersection point.



Let's show that this time they meet at the cycle entrance after F*F* steps.

* The tortoise started at zero, so its position after F*F* steps is F*F*.
* The hare started at the intersection point F + a = nC*F*+*a*=*nC*, so its position after F steps is nC + F*nC*+*F*, that is the same point as F*F*.
* So the tortoise and the (slowed down) hare will meet at the entrance of the cycle.

class Solution {

public:

int findDuplicate(vector<int>& nums) {

// Find the intersection point of the two runners.

int tortoise = nums[0];

int hare = nums[0];

do {

tortoise = nums[tortoise];

hare = nums[nums[hare]];

} while (tortoise != hare);

// Find the "entrance" to the cycle.

tortoise = nums[0];

while (tortoise != hare) {

tortoise = nums[tortoise];

hare = nums[hare];

}

return hare;

}

};

**Complexity Analysis**

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

### Merge 2 sorted arrays without using Extra space

Given two sorted arrays arr1[] of size N and arr2[] of size M. Each array is sorted in non-decreasing order. Merge the two arrays into one sorted array in non-decreasing order without using any extra space.

Example 1:

Input:

N = 4, M = 5

arr1[] = {1, 3, 5, 7}

arr2[] = {0, 2, 6, 8, 9}

Output: 0 1 2 3 5 6 7 8 9

Explanation: Since you can't use any

extra space, modify the given arrays

to form

arr1[] = {0, 1, 2, 3}

arr2[] = {5, 6, 7, 8, 9}

Example 2:

Input:

N = 2, M = 3

arr1[] = {10, 12}

arr2[] = {5, 18, 20}

Output: 5 10 12 18 20

Explanation: Since you can't use any

extra space, modify the given arrays

to form

arr1[] = {5, 10}

arr2[] = {12, 18, 20}

Your Task:  
You don't need to read input or print anything. Complete the function merge() which takes the two arrays arr1[], arr2[] and their sizes n and m, as input parameters. The function does not return anything. Use the given arrays to sort and merge arr1[] and arr2[] in-place.   
Note: The generated output will print all the elements of arr1[] followed by all the elements of arr2[].

Expected Time Complexity: O((n+m)\*log(n+m))  
Expected Auxiliary Space: O(1)

Constraints:  
1 <= N, M <= 5\*104  
0 <= arr1i, arr2i <= 106

## Approach 1

The idea is: We will traverse the first array and compare it with the first element of the second array. If the first element of the second array is smaller than the first array, we will swap and then sort the second array.

1. First, we have to traverse array1 and then compare it with the first element of array2.   
   If it is less than array1 then we swap both.
2. After swapping we are going to sort the array2 again so that the smallest element of the array2   
   comes at the first position and we can again swap with the array1
3. To sort the array2 we will store the first element of array2 in a variable and left shift all the elements and store  
   the first element in array2 in the last.

#include <bits/stdc++.h>

using namespace std;

void mergeArray(int arr1[], int arr2[],

int n, int m)

{

// Now traverse the array1 and if

// arr2 first element

// is less than arr1 then swap

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

{

if (arr1[i] > arr2[0])

{

// Swap

int temp = arr1[i];

arr1[i] = arr2[0];

arr2[0] = temp;

// After swapping we have to sort the array2

// again so that it can be again swap with

// arr1

// We will store the firstElement of array2

// and left shift all the element and store

// the firstElement in arr2[k-1]

int firstElement = arr2[0];

int k;

for(k = 1;

k < m && arr2[k] < firstElement;

k++)

{

arr2[k - 1] = arr2[k];

}

arr2[k - 1] = firstElement;

}

}

// Read the arr1

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

{

cout << arr1[i] << " ";

}

cout << endl;

// Read the arr2

for(int i = 0; i < m; i++)

{

cout << arr2[i] << " ";

}

}

// Driver Code

int main()

{

int arr1[] = { 1, 3, 5, 7 };

int arr2[] = { 0, 2, 6, 8, 9 };

int n = arr1.length, m = arr2.length;

mergeArray(arr1, arr2, n, m);

}

## Approach 2

The idea: We start comparing elements that are far from each other rather than adjacent.   
For every pass, we calculate the gap and compare the elements towards the right of the gap. Every pass, the gap reduces to the ceiling value of dividing by 2.

Examples:

First example:

a1[] = {3 27 38 43},

a2[] = {9 10 82}

Start with

gap = ceiling of n/2 = 4

[This gap is for whole merged array]

3 27 38 43 9 10 82

3 27 38 43 9 10 82

3 10 38 43 9 27 82

gap = 2:

3 10 38 43 9 27 82

3 10 38 43 9 27 82

3 10 38 43 9 27 82

3 10 9 43 38 27 82

3 10 9 27 38 43 82

gap = 1:

3 10 9 27 38 43 82

3 10 9 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

Output : 3 9 10 27 38 43 82

Second Example:

a1[] = {10 27 38 43 82},

a2[] = {3 9}

Start with gap = ceiling of n/2 (4):

10 27 38 43 82 3 9

10 27 38 43 82 3 9

10 3 38 43 82 27 9

10 3 9 43 82 27 38

gap = 2:

10 3 9 43 82 27 38

9 3 10 43 82 27 38

9 3 10 43 82 27 38

9 3 10 43 82 27 38

9 3 10 27 82 43 38

9 3 10 27 38 43 82

gap = 1

9 3 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

3 9 10 27 38 43 82

Output : 3 9 10 27 38 43 82

 Below is the implementation of the above idea:

// Merging two sorted arrays with O(1)

// extra space

#include <bits/stdc++.h>

using namespace std;

// Function to find next gap.

int nextGap(int gap)

{

if (gap <= 1)

return 0;

return (gap / 2) + (gap % 2);

}

void merge(int\* arr1, int\* arr2, int n, int m)

{

int i, j, gap = n + m;

for (gap = nextGap(gap);

gap > 0; gap = nextGap(gap))

{

// comparing elements in the first array.

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

if (arr1[i] > arr1[i + gap])

swap(arr1[i], arr1[i + gap]);

// comparing elements in both arrays.

for (j = gap > n ? gap - n : 0;

i < n && j < m;

i++, j++)

if (arr1[i] > arr2[j])

swap(arr1[i], arr2[j]);

if (j < m) {

// comparing elements in the second array.

for (j = 0; j + gap < m; j++)

if (arr2[j] > arr2[j + gap])

swap(arr2[j], arr2[j + gap]);

}

}

}

// Driver code

int main()

{

int a1[] = { 10, 27, 38, 43, 82 };

int a2[] = { 3, 9 };

int n = sizeof(a1) / sizeof(int);

int m = sizeof(a2) / sizeof(int);

// Function Call

merge(a1, a2, n, m);

printf("First Array: ");

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

printf("%d ", a1[i]);

printf("\nSecond Array: ");

for (int i = 0; i < m; i++)

printf("%d ", a2[i]);

printf("\n");

return 0;

}

Output

First Array: 3 9 10 27 38

Second Array: 43 82

Time Complexity : O((m+n)log(m+n))

Space Complexity : O(1)

## Approach 3:

#### ****Another method in O(m+n) time complexity:****

Here we use the below technique:

Suppose we have a number A and we want to

convert it to a number B and there is also a

constraint that we can recover number A any

time without using other variable.To achieve

this we chose a number N which is greater

than both numbers and add B\*N in A.

so A --> A+B\*N

To get number B out of (A+B\*N)

we divide (A+B\*N) by N.

so (A+B\*N)/N = B.

To get number A out of (A+B\*N)

we take modulo with N.

so (A+B\*N)%N = A.

-> In short by taking modulo

we get old number back and taking divide

we new number.

We first find the maximum element of both arrays and increment it by one to avoid collision of 0 and maximum element during modulo operation. The idea is to traverse both arrays from starting simultaneously. Let's say an element in a is a[i] and in b is b[j] and k is the position at where the next minimum number will come. Now update value a[k] if k<n else b[k-n] by adding min(a[i],b[j])\*maximum\_element. After updating all the elements divide all the elements by maximum\_element so we get the updated array back.

Below is the implementation of the above idea:

#include <bits/stdc++.h>

using namespace std;

void mergeArray(int a[], int b[], int n, int m)

{

int mx = 0;

// Find maximum element of both array

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

mx = max(mx, a[i]);

}

for (int i = 0; i < m; i++) {

mx = max(mx, b[i]);

}

// increment by one to avoid collision of 0 and maximum

// element of array in modulo operation

mx++;

int i = 0, j = 0, k = 0;

while (i < n && j < m && k < (n + m)) {

// recover back original element to compare

int e1 = a[i] % mx;

int e2 = b[j] % mx;

if (e1 <= e2) {

// update element by adding multiplication

// with new number

if (k < n)

a[k] += (e1 \* mx);

else

b[k - n] += (e1 \* mx);

i++;

k++;

}

else {

// update element by adding multiplication

// with new number

if (k < n)

a[k] += (e2 \* mx);

else

b[k - n] += (e2 \* mx);

j++;

k++;

}

}

// process those elements which are left in array a

while (i < n) {

int el = a[i] % mx;

if (k < n)

a[k] += (el \* mx);

else

b[k - n] += (el \* mx);

i++;

k++;

}

// process those elements which are left in array b

while (j < m) {

int el = b[j] % mx;

if (k < n)

a[k] += (el \* mx);

else

b[k - n] += (el \* mx);

j++;

k++;

}

// finally update elements by dividing

// with maximum element

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

a[i] = a[i] / mx;

// finally update elements by dividing

// with maximum element

for (int i = 0; i < m; i++)

b[i] = b[i] / mx;

return;

}

// Driver Code

int main()

{

int a[] = { 3, 5, 6, 8, 12 };

int b[] = { 1, 4, 9, 13 };

int n = sizeof(a) / sizeof(int); // Length of a

int m = sizeof(b) / sizeof(int); // length of b

// Function Call

mergeArray(a, b, n, m);

cout << "First array : ";

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

cout << a[i] << " ";

cout << endl;

cout << "Second array : ";

for (int i = 0; i < m; i++)

cout << b[i] << " ";

cout << endl;

return 0;

}

Output

First array : 1 3 4 5 6

Second array : 8 9 12 13

Time Complexity: O(m+n)

### Kadane’s algo

Same as Question 8. (Largest sum contiguous subarray).

### Merge Intervals

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Example 1:

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

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

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

Example 2:

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

Output: [[1,5]]

Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Constraints:

* 1 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= starti <= endi <= 104

## Solution:

Intuition

If we sort the intervals by their start value, then each set of intervals that can be merged will appear as a contiguous "run" in the sorted list.

Algorithm

First, we sort the list as described. Then, we insert the first interval into our merged list and continue considering each interval in turn as follows: If the current interval begins *after* the previous interval ends, then they do not overlap and we can append the current interval to merged. Otherwise, they do overlap, and we merge them by updating the end of the previous interval if it is less than the end of the current interval.

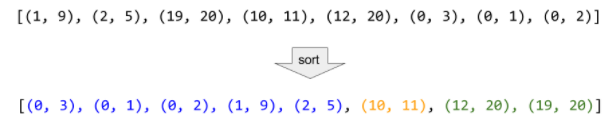
A simple proof by contradiction shows that this algorithm always produces the correct answer. First, suppose that the algorithm at some point fails to merge two intervals that should be merged. This would imply that there exists some triple of indices i*i*, j*j*, and k*k* in a list of intervals ints such that *i*<*j*<*k* and ints[i], ints[k]) can be merged, but neither (ints[i], ints[j]) nor (ints[j], ints[k]) can be merged. From this scenario follow several inequalities:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \\ \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \\ \end{aligned}ints[i].end<ints[j].startints[j].end<ints[k].startints[i].end≥ints[k].start​

We can chain these inequalities (along with the following inequality, implied by the well-formedness of the intervals: \text{ints[j].start} \leq \text{ints[j].end}ints[j].start≤ints[j].end) to demonstrate a contradiction:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \leq \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \end{aligned}ints[i].end<ints[j].start≤ints[j].end<ints[k].startints[i].end≥ints[k].start​

Therefore, all mergeable intervals must occur in a contiguous run of the sorted list.



Consider the example above, where the intervals are sorted, and then all mergeable intervals form contiguous blocks.

class Solution {

public:

vector<vector<int>> merge(vector<vector<int>>& intervals) {

sort(intervals.begin(), intervals.end());

vector<vector<int>> merged;

for (auto interval : intervals) {

// if the list of merged intervals is empty or if the current

// interval does not overlap with the previous, simply append it.

if (merged.empty() || merged.back()[1] < interval[0]) {

merged.push\_back(interval);

}

// otherwise, there is overlap, so we merge the current and previous

// intervals.

else {

merged.back()[1] = max(merged.back()[1], interval[1]);

}

}

return merged;

}

};

**Complexity Analysis**

* Time complexity : *O*(*n*log*n*)

Other than the sort invocation, we do a simple linear scan of the list, so the runtime is dominated by the *O*(*n*log*n*) complexity of sorting.

* Space complexity : *O*(log*N*) (or *O*(*n*))

If we can sort intervals in place, we do not need more than constant additional space, although the sorting itself takes *O*(log*n*) space. Otherwise, we must allocate linear space to store a copy of intervals and sort that.

### Next Permutation

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** nums = [3,2,1]

**Output:** [1,2,3]

**Example 3:**

**Input:** nums = [1,1,5]

**Output:** [1,5,1]

**Example 4:**

**Input:** nums = [1]

**Output:** [1]

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

#### Single Pass Approach

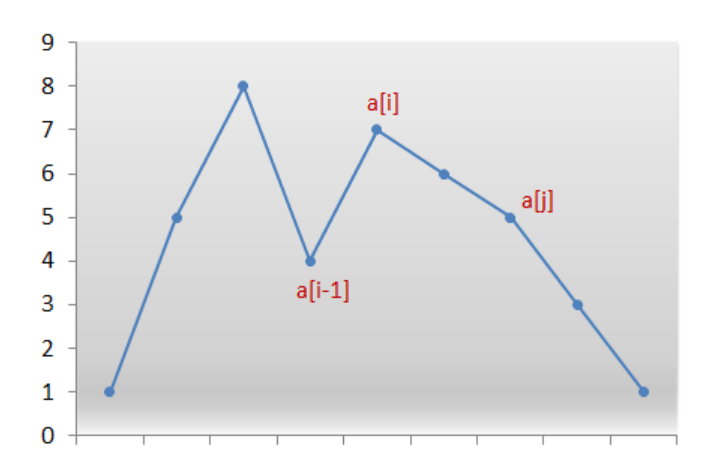
**Algorithm**

First, we observe that for any given sequence that is in descending order, no next larger permutation is possible. For example, no next permutation is possible for the following array:

[9, 5, 4, 3, 1]

We need to find the first pair of two successive numbers *a*[*i*] and *a*[*i*−1], from the right, which satisfy a[i] > a[i-1]. Now, no rearrangements to the right of *a*[*i*−1] can create a larger permutation since that subarray consists of numbers in descending order. Thus, we need to rearrange the numbers to the right of a[i-1] including itself.

Now, what kind of rearrangement will produce the next larger number? We want to create the permutation just larger than the current one. Therefore, we need to replace the number a[i-1] with the number which is just larger than itself among the numbers lying to its right section, say *a*[*j*].



We swap the numbers *a*[*i*−1] and *a*[*j*]. We now have the correct number at index i-1. But still the current permutation isn't the permutation that we are looking for. We need the smallest permutation that can be formed by using the numbers only to the right of a[i-1]. Therefore, we need to place those numbers in ascending order to get their smallest permutation.

But, recall that while scanning the numbers from the right, we simply kept decrementing the index until we found the pair a[i]*a*[*i*] and a[i-1] where, a[i] > a[i-1]. Thus, all numbers to the right of a[i-1] were already sorted in descending order. Furthermore, swapping a[i-1] and *a*[*j*] didn't change that order. Therefore, we simply need to reverse the numbers following *a*[*i*−1] to get the next smallest lexicographic permutation.

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int i = nums.size()-2;

while(i>=0 && nums[i+1]<=nums[i]){

i--;

}

if(i>=0){

int j = nums.size()-1;

while(nums[j]<=nums[i]){

j--;

}

swap(nums,i,j);

}

reverse(nums,i+1);

}

void reverse(vector<int> &nums, int start){

int i=start,j=nums.size()-1;

while(i<j){

swap(nums,i,j);

i++;

j--;

}

}

void swap(vector<int> &nums,int i,int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

};

### Count Inversions

*Inversion Count*for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in the reverse order, the inversion count is the maximum.   
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j   
Example:

Input: arr[] = {8, 4, 2, 1}

Output: 6

Explanation: Given array has six inversions:

(8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).

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

Output: 2

Explanation: Given array has two inversions:

(3, 1), (3, 2)

## Solution:

METHOD 1 (Simple)

* Approach: Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all index in the array and print the sum.
* Algorithm:
  1. Traverse through the array from start to end
  2. For every element, find the count of elements smaller than the current number up to that index using another loop.
  3. Sum up the count of inversion for every index.
  4. Print the count of inversions.
* Implementation:

// C++ program to Count Inversions

// in an array

#include <bits/stdc++.h>

using namespace std;

int getInvCount(int arr[], int n)

{

int inv\_count = 0;

for (int i = 0; i < n - 1; i++)

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

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

inv\_count++;

return inv\_count;

}

// Driver Code

int main()

{

int arr[] = { 1, 20, 6, 4, 5 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << " Number of inversions are "

<< getInvCount(arr, n);

return 0;

}

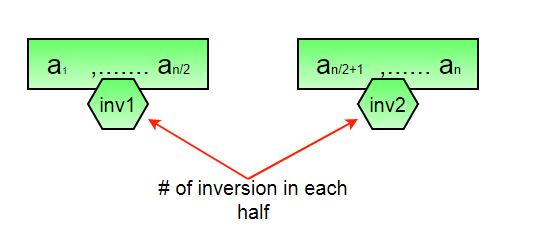
Output

Number of inversions are 5

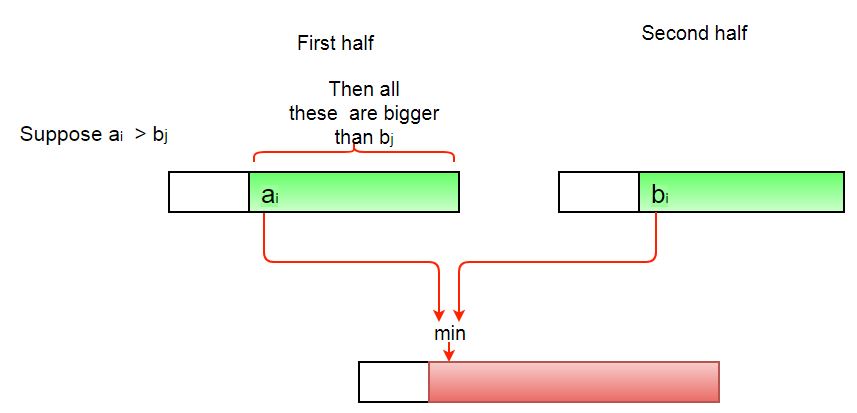
* Complexity Analysis:
  + Time Complexity: O(n^2), Two nested loops are needed to traverse the array from start to end, so the Time complexity is O(n^2)
  + Space Complexity:O(1), No extra space is required.

METHOD 2(Enhance Merge Sort)

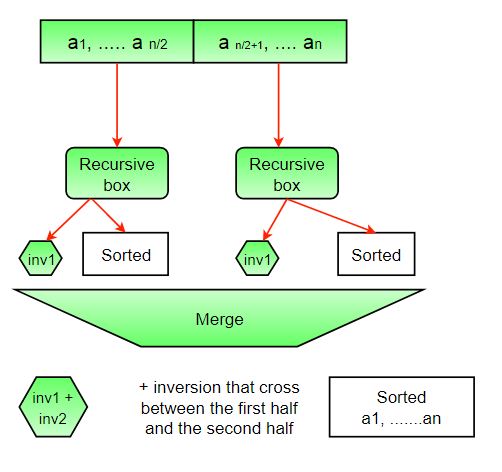
* Approach:   
  Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that needs to be added are the number of inversions in the left subarray, right subarray, and merge().



* How to get the number of inversions in merge()?   
  In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j]



* The complete picture:



* Algorithm:
  1. The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
  2. Create a function merge that counts the number of inversions when two halves of the array are merged, create two indices i and j, i is the index for the first half, and j is an index of the second half. if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
  3. Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, the number of inversion in the second half and the number of inversions by merging the two.
  4. The base case of recursion is when there is only one element in the given half.
  5. Print the answer
* Implementation:

// C++ program to Count

// Inversions in an array

// using Merge Sort

#include <bits/stdc++.h>

using namespace std;

int \_mergeSort(int arr[], int temp[], int left, int right);

int merge(int arr[], int temp[], int left, int mid,

int right);

/\* This function sorts the

input array and returns the

number of inversions in the array \*/

int mergeSort(int arr[], int array\_size)

{

int temp[array\_size];

return \_mergeSort(arr, temp, 0, array\_size - 1);

}

/\* An auxiliary recursive function

that sorts the input array and

returns the number of inversions in the array. \*/

int \_mergeSort(int arr[], int temp[], int left, int right)

{

int mid, inv\_count = 0;

if (right > left) {

/\* Divide the array into two parts and

call \_mergeSortAndCountInv()

for each of the parts \*/

mid = (right + left) / 2;

/\* Inversion count will be sum of

inversions in left-part, right-part

and number of inversions in merging \*/

inv\_count += \_mergeSort(arr, temp, left, mid);

inv\_count += \_mergeSort(arr, temp, mid + 1, right);

/\*Merge the two parts\*/

inv\_count += merge(arr, temp, left, mid + 1, right);

}

return inv\_count;

}

/\* This funt merges two sorted arrays

and returns inversion count in the arrays.\*/

int merge(int arr[], int temp[], int left, int mid,

int right)

{

int i, j, k;

int inv\_count = 0;

i = left; /\* i is index for left subarray\*/

j = mid; /\* j is index for right subarray\*/

k = left; /\* k is index for resultant merged subarray\*/

while ((i <= mid - 1) && (j <= right)) {

if (arr[i] <= arr[j]) {

temp[k++] = arr[i++];

}

else {

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

/\* this is tricky -- see above

explanation/diagram for merge()\*/

inv\_count = inv\_count + (mid - i);

}

}

/\* Copy the remaining elements of left subarray

(if there are any) to temp\*/

while (i <= mid - 1)

temp[k++] = arr[i++];

/\* Copy the remaining elements of right subarray

(if there are any) to temp\*/

while (j <= right)

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

/\*Copy back the merged elements to original array\*/

for (i = left; i <= right; i++)

arr[i] = temp[i];

return inv\_count;

}

// Driver code

int main()

{

int arr[] = { 1, 20, 6, 4, 5 };

int n = sizeof(arr) / sizeof(arr[0]);

int ans = mergeSort(arr, n);

cout << " Number of inversions are " << ans;

return 0;

}

Output:

Number of inversions are 5

Complexity Analysis: 

* Time Complexity: O(n log n), The algorithm used is divide and conquer, So in each level, one full array traversal is needed, and there are log n levels, so the time complexity is O(n log n).
* Space Complexity: O(n), Temporary array.

Note that the above code modifies (or sorts) the input array. If we want to count only inversions, we need to create a copy of the original array and call mergeSort() on the copy to preserve the original array’s order.

### Best time to buy and Sell stock

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

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

Output: 5

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

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

Example 2:

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

Output: 0

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

Constraints:

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104

## Solution:

We need to find out the maximum difference (which will be the maximum profit) between two numbers in the given array. Also, the second number (selling price) must be larger than the first one (buying price).

In formal terms, we need to find max(prices[j]−prices[i]), for every i and j such that j > i.

#### Approach 1: Brute Force

public class Solution {

public int maxProfit(int prices[]) {

int maxprofit = 0;

for (int i = 0; i < prices.length - 1; i++) {

for (int j = i + 1; j < prices.length; j++) {

int profit = prices[j] - prices[i];

if (profit > maxprofit)

maxprofit = profit;

}

}

return maxprofit;

}

}

**Complexity Analysis**

* Time complexity : O(n^2). Loop runs n(n-1)/2​ times.
* Space complexity : O(1). Only two variables - maxprofit and profit are used.

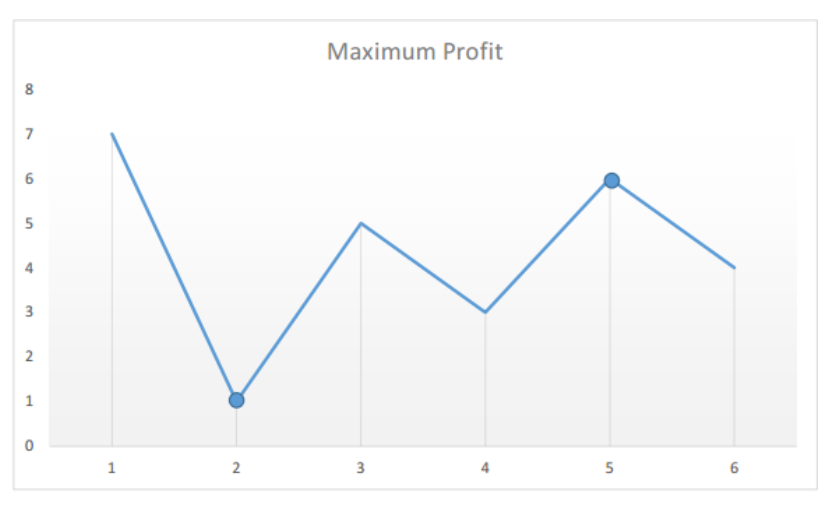
#### Approach 2: One Pass

**Algorithm**

Say the given array is:

[7, 1, 5, 3, 6, 4]

If we plot the numbers of the given array on a graph, we get:



The points of interest are the peaks and valleys in the given graph. We need to find the largest peak following the smallest valley. We can maintain two variables - minprice and maxprofit corresponding to the smallest valley and maximum profit (maximum difference between selling price and minprice) obtained so far respectively.

class Solution {

public:

int maxProfit(vector<int>& prices) {

int min = prices[0], max = 0, n = prices.size();

for(int i=1;i<n;i++){

if(prices[i]<min)

min = prices[i];

if(prices[i]-min>max)

max = prices[i] - min;

}

return max;

}

};

**Complexity Analysis**

* Time complexity : O(n)*.* Only a single pass is needed.
* Space complexity : *O*(1). Only two variables are used.

### find all pairs on integer array whose sum is equal to given number

Given an array of integers, and a number ‘sum’, find the number of pairs of integers in the array whose sum is equal to ‘sum’.

Examples:

Input : arr[] = {1, 5, 7, -1},

sum = 6

Output : 2

Pairs with sum 6 are (1, 5) and (7, -1)

Input : arr[] = {1, 5, 7, -1, 5},

sum = 6

Output : 3

Pairs with sum 6 are (1, 5), (7, -1) &

(1, 5)

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

sum = 2

Output : 6

There are 3! pairs with sum 2.

Input : arr[] = {10, 12, 10, 15, -1, 7, 6,

5, 4, 2, 1, 1, 1},

sum = 11

Output : 9

Expected time complexity O(n)

**Naive Solution –**A **simple solution** is to traverse each element and check if there’s another number in the array which can be added to it to give sum.

// C++ implementation of simple method to find count of

// pairs with given sum.

#include <bits/stdc++.h>

using namespace std;

// Returns number of pairs in arr[0..n-1] with sum equal

// to 'sum'

int getPairsCount(int arr[], int n, int sum)

{

int count = 0; // Initialize result

// Consider all possible pairs and check their sums

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

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

if (arr[i] + arr[j] == sum)

count++;

return count;

}

// Driver function to test the above function

int main()

{

int arr[] = { 1, 5, 7, -1, 5 };

int n = sizeof(arr) / sizeof(arr[0]);

int sum = 6;

cout << "Count of pairs is "

<< getPairsCount(arr, n, sum);

return 0;

}

Output

Count of pairs is 3

Time Complexity: O(n2)   
Auxiliary Space: O(1)

Efficient solution –   
A better solution is possible in O(n) time. Below is the Algorithm –

1. Create a map to store frequency of each number in the array. (Single traversal is required)
2. In the next traversal, for every element check if it can be combined with any other element (other than itself!) to give the desired sum. Increment the counter accordingly.
3. After completion of second traversal, we’d have twice the required value stored in counter because every pair is counted two times. Hence divide count by 2 and return.

Below is the implementation of above idea :

// C++ implementation of simple method to find count of

// pairs with given sum.

#include <bits/stdc++.h>

using namespace std;

// Returns number of pairs in arr[0..n-1] with sum equal

// to 'sum'

int getPairsCount(int arr[], int n, int sum)

{

unordered\_map<int, int> m;

// Store counts of all elements in map m

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

m[arr[i]]++;

int twice\_count = 0;

// iterate through each element and increment the

// count (Notice that every pair is counted twice)

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

twice\_count += m[sum - arr[i]];

// if (arr[i], arr[i]) pair satisfies the condition,

// then we need to ensure that the count is

// decreased by one such that the (arr[i], arr[i])

// pair is not considered

if (sum - arr[i] == arr[i])

twice\_count--;

}

// return the half of twice\_count

return twice\_count / 2;

}

// Driver function to test the above function

int main()

{

int arr[] = { 1, 5, 7, -1, 5 };

int n = sizeof(arr) / sizeof(arr[0]);

int sum = 6;

cout << "Count of pairs is "

<< getPairsCount(arr, n, sum);

return 0;

}

Output

Count of pairs is 3

More efficient solution in one loop:-

// C++ implementation of simple method to find count of

// pairs with given sum.

#include <bits/stdc++.h>

using namespace std;

// Returns number of pairs in arr[0..n-1] with sum equal

// to 'sum'

int getPairsCount(int arr[], int n, int k)

{

unordered\_map<int, int> m;

int count = 0;

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

if (m.find(k - arr[i]) != m.end()) {

count += m[k - arr[i]];

}

m[arr[i]]++;

}

return count;

}

// Driver function to test the above function

int main()

{

int arr[] = { 1, 5, 7, -1, 5};

int n = sizeof(arr) / sizeof(arr[0]);

int sum = 6;

cout << "Count of pairs is "

<< getPairsCount(arr, n, sum);

return 0;

}

Output

Count of pairs is 3

### find common elements In 3 sorted arrays

Given three arrays sorted in increasing order. Find the elements that are common in all three arrays.  
Note: can you take care of the duplicates without using any additional Data Structure?

Example 1:

Input:

n1 = 6; A = {1, 5, 10, 20, 40, 80}

n2 = 5; B = {6, 7, 20, 80, 100}

n3 = 8; C = {3, 4, 15, 20, 30, 70, 80, 120}

Output: 20 80

Explanation: 20 and 80 are the only

common elements in A, B and C.

Your Task:    
You don't need to read input or print anything. Your task is to complete the function commonElements() which take the 3 arrays A[], B[], C[] and their respective sizes n1, n2 and n3 as inputs and returns an array containing the common element present in all the 3 arrays in sorted order.   
If there are no such elements return an empty array. In this case the output will be printed as -1.

Expected Time Complexity: O(n1 + n2 + n3)  
Expected Auxiliary Space: O(n1 + n2 + n3)

Constraints:  
1 <= n1, n2, n3 <= 10^5  
The array elements can be both positive or negative integers.

## Solution:

vector <int> commonElements (int A[], int B[], int C[], int n1, int n2, int n3)

{

vector<int> common;

int i=0,j=0,k=0;

while(i<n1 && j<n2 && k<n3){

if(A[i]==B[j] && B[j]==C[k]){

if(common.empty() || common.back()!=A[i])

common.push\_back(A[i]);

i++;

j++;

k++;

}

else if(A[i]<=B[j]){

A[i]<=C[k] ? i++ : k++;

}

else{

B[j]<=C[k] ? j++ : k++;

}

}

return common;

}

Time Complexity of above approach is O(n1+n2+n3).

Space Complexity is O(n) for vector used to store duplicate elements.

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

## Solution:

Duplicate entries can be handled without using any additional data structure by keeping the track of the previous element. Since the elements inside the array are arranged in sorted manner there is no possibility for the repeated elements to occur at random positions.

Let’s consider the current element traversed in ar1[] be x, in ar2[] be y and in ar3[] be z and let the variables prev1, prev2, prev3 for keeping the track of last encountered element in each array and initialize them with INT\_MIN. Hence for every element we visit across each array, we check for the following.

* If x = prev1, move ahead in ar1[] and repeat the procedure until x != prev1. Similarly, apply the same for the ar2[] and ar3[].
* If x, y, and z are same, we can simply print any of them as common element, update prev1, prev2, and prev3 and move ahead in all three arrays.
* Else If (x < y), we update prev1 and move ahead in ar1[] as x cannot be a common element.
* Else If (y < z), we update prev2 and move ahead in ar2[] as y cannot be a common element.
* Else If (x > z and y > z), we update prev3 and we move ahead in ar3[] as z cannot be a common element.

Below is the implementation of the above approach:

// C++ program to print common

// elements in three arrays

#include <bits/stdc++.h>

using namespace std;

// This function prints

// common elements in ar1

void findCommon(int ar1[], int ar2[], int ar3[], int n1,

int n2, int n3)

{

// Initialize starting indexes

// for ar1[], ar2[] and

// ar3[]

int i = 0, j = 0, k = 0;

// Declare three variables prev1,

// prev2, prev3 to track

// previous element

int prev1, prev2, prev3;

// Initialize prev1, prev2,

// prev3 with INT\_MIN

prev1 = prev2 = prev3 = INT\_MIN;

// Iterate through three arrays

// while all arrays have

// elements

while (i < n1 && j < n2 && k < n3) {

// If ar1[i] = prev1 and i < n1,

// keep incrementing i

while (ar1[i] == prev1 && i < n1)

i++;

// If ar2[j] = prev2 and j < n2,

// keep incrementing j

while (ar2[j] == prev2 && j < n2)

j++;

// If ar3[k] = prev3 and k < n3,

// keep incrementing k

while (ar3[k] == prev3 && k < n3)

k++;

// If x = y and y = z, print

// any of them, update

// prev1 prev2, prev3 and move

//ahead in each array

if (ar1[i] == ar2[j] && ar2[j] == ar3[k]) {

cout << ar1[i] << " ";

prev1 = ar1[i];

prev2 = ar2[j];

prev3 = ar3[k];

i++;

j++;

k++;

}

// If x < y, update prev1

// and increment i

else if (ar1[i] < ar2[j]) {

prev1 = ar1[i];

i++;

}

// If y < z, update prev2

// and increment j

else if (ar2[j] < ar3[k]) {

prev2 = ar2[j];

j++;

}

// We reach here when x > y

// and z < y, i.e., z is

// smallest update prev3

// and imcrement k

else {

prev3 = ar3[k];

k++;

}

}

}

// Driver code

int main()

{

int ar1[] = { 1, 5, 10, 20, 40, 80, 80 };

int ar2[] = { 6, 7, 20, 80, 80, 100 };

int ar3[] = { 3, 4, 15, 20, 30, 70, 80, 80, 120 };

int n1 = sizeof(ar1) / sizeof(ar1[0]);

int n2 = sizeof(ar2) / sizeof(ar2[0]);

int n3 = sizeof(ar3) / sizeof(ar3[0]);

cout << "Common Elements are ";

findCommon(ar1, ar2, ar3, n1, n2, n3);

return 0;

}

Output

Common Elements are 20 80

Time Complexity for the above approach still remains O(n1 + n2 + n3) and space complexity also remains O(1) and no extra space and data structure is required to handle the duplicate array entries.

### Rearrange the array in alternating positive and negative items with O(1) extra space

Given an array of positive and negative numbers, arrange them in an alternate fashion such that every positive number is followed by negative and vice-versa maintaining the order of appearance.   
Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.

Examples :

Input: arr[] = {1, 2, 3, -4, -1, 4}

Output: arr[] = {-4, 1, -1, 2, 3, 4}

Input: arr[] = {-5, -2, 5, 2, 4, 7, 1, 8, 0, -8}

output: arr[] = {-5, 5, -2, 2, -8, 4, 7, 1, 8, 0}

Naive Approach :   
The above problem can be easily solved if O(n) extra space is allowed. It becomes interesting due to the limitations that O(1) extra space and order of appearances.   
The idea is to process array from left to right. While processing, find the first out of place element in the remaining unprocessed array. An element is out of place if it is negative and at odd index, or it is positive and at even index. Once we find an out of place element, we find the first element after it with opposite sign. We right rotate the subarray between these two elements (including these two).

Following is the implementation of above idea.

/\* C++ program to rearrange

positive and negative integers

in alternate fashion while keeping

the order of positive and negative numbers. \*/

#include <assert.h>

#include <iostream>

using namespace std;

// Utility function to right rotate all elements between

// [outofplace, cur]

void rightrotate(int arr[], int n, int outofplace, int cur)

{

char tmp = arr[cur];

for (int i = cur; i > outofplace; i--)

arr[i] = arr[i - 1];

arr[outofplace] = tmp;

}

void rearrange(int arr[], int n)

{

int outofplace = -1;

for (int index = 0; index < n; index++)

{

if (outofplace >= 0)

{

// find the item which must be moved into the

// out-of-place entry if out-of-place entry is

// positive and current entry is negative OR if

// out-of-place entry is negative and current

// entry is negative then right rotate

//

// [...-3, -4, -5, 6...] --> [...6, -3, -4,

// -5...]

// ^ ^

// | |

// outofplace --> outofplace

//

if (((arr[index] >= 0) && (arr[outofplace] < 0))

|| ((arr[index] < 0)

&& (arr[outofplace] >= 0)))

{

rightrotate(arr, n, outofplace, index);

// the new out-of-place entry is now 2 steps

// ahead

if (index - outofplace >= 2)

outofplace = outofplace + 2;

else

outofplace = -1;

}

}

// if no entry has been flagged out-of-place

if (outofplace == -1) {

// check if current entry is out-of-place

if (((arr[index] >= 0) && (!(index & 0x01)))

|| ((arr[index] < 0) && (index & 0x01))) {

outofplace = index;

}

}

}

}

// A utility function to print an array 'arr[]' of size 'n'

void printArray(int arr[], int n)

{

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

cout << arr[i] << " ";

cout << endl;

}

// Driver code

int main()

{

int arr[] = { -5, -2, 5, 2,

4, 7, 1, 8, 0, -8 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Given array is \n";

printArray(arr, n);

rearrange(arr, n);

cout << "Rearranged array is \n";

printArray(arr, n);

return 0;

}

Output

Given array is

-5 -2 5 2 4 7 1 8 0 -8

Rearranged array is

-5 5 -2 2 -8 4 7 1 8 0

Time Complexity : O(N^2)

Space Complexity : O(1)

Given an array of positive and negative numbers, arrange them in an alternate fashion such that every positive number is followed by negative and vice-versa. Order of elements in output doesn’t matter. Extra positive or negative elements should be moved to end.

Examples:

Input :

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

Output :

arr[] = {-2, 3, -1, 4} OR {-1, 3, -2, 4} OR ..

Input :

arr[] = {-2, 3, 1}

Output :

arr[] = {-2, 3, 1} OR {-2, 1, 3}

Input :

arr[] = {-5, 3, 4, 5, -6, -2, 8, 9, -1, -4}

Output :

arr[] = {-5, 3, -2, 5, -6, 4, -4, 9, -1, 8}

OR ..

Approach 1:

1. First, sort the array in non-increasing order. Then we will count the number of positive and negative integers.
2. Then swap the one negative and one positive number in the odd positions till we reach our condition.
3. This will rearrange the array elements because we are sorting the array and accessing the element from left to right according to our need.

Below is the implementation of the above approach:

// Below is the implementation of the above approach

#include <bits/stdc++.h>

using namespace std;

// Function which works in the condition

// when number of negative numbers are

// lesser or equal than positive numbers

void fill1(int a[], int neg, int pos)

{

if (neg % 2 == 1)

{

for(int i = 1; i < neg; i += 2)

{

int c = a[i];

int d = a[i + neg];

int temp = c;

a[i] = d;

a[i + neg] = temp;

}

}

else

{

for(int i = 1; i <= neg; i += 2)

{

int c = a[i];

int d = a[i + neg - 1];

int temp = c;

a[i] = d;

a[i + neg - 1] = temp;

}

}

}

// Function which works in the condition

// when number of negative numbers are

// greater than positive numbers

void fill2(int a[], int neg, int pos)

{

if (pos % 2 == 1)

{

for(int i = 1; i < pos; i += 2)

{

int c = a[i];

int d = a[i + pos];

int temp = c;

a[i] = d;

a[i + pos] = temp;

}

}

else

{

for(int i = 1; i <= pos; i += 2)

{

int c = a[i];

int d = a[i + pos - 1];

int temp = c;

a[i] = d;

a[i + pos - 1] = temp;

}

}

}

// Reverse the array

void reverse(int a[], int n)

{

int i, k, t;

for(i = 0; i < n / 2; i++)

{

t = a[i];

a[i] = a[n - i - 1];

a[n - i - 1] = t;

}

}

// Print the array

void print(int a[], int n)

{

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

cout << a[i] << " ";

cout << endl;

}

// Driver code

int main()

{

int arr[] = { 2, 3, -4, -1, 6, -9 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Given array is ";

print(arr, n);

int neg = 0, pos = 0;

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

{

if (arr[i] < 0)

neg++;

else

pos++;

}

// Sort the array

sort(arr, arr + n);

if (neg <= pos)

{

fill1(arr, neg, pos);

}

else

{

// Reverse the array in this condition

reverse(arr, n);

fill2(arr, neg, pos);

}

cout << "Rearranged array is ";

print(arr, n);

}

**Output:**

Given array is

2 3 -4 -1 6 -9

Rearranged array is

-9 3 -1 2 -4 6

**Time Complexity:**O(N\*logN)

**Space Complexity:** O(1)

**Efficient Approach :**  
We have already discussed a O(n2) solution that maintains the order of appearance in the array [here](https://www.geeksforgeeks.org/rearrange-array-alternating-positive-negative-items-o1-extra-space/). If we are allowed to change order of appearance, we can solve this problem in O(n) time and O(1) space.  
The idea is to process the array and shift all negative values to the end in O(n) time. After all negative values are shifted to the end, we can easily rearrange array in alternating positive & negative items. We basically swap next positive element at even position from next negative element in this step.

Following is the implementation of above idea.

// C++ program to rearrange

// array in alternating

// positive & negative items

// with O(1) extra space

#include <bits/stdc++.h>

using namespace std;

// Function to rearrange positive and negative

// integers in alternate fashion. The below

// solution doesn't maintain original order of

// elements

void rearrange(int arr[], int n)

{

int i = 0, j = n-1;

// shift all negative values to the end

while (i < j) {

while (i <= n - 1 and arr[i] > 0)

i += 1;

while (j >= 0 and arr[j] < 0)

j -= 1;

if (i < j )

swap(arr[i], arr[j]);

}

// i has index of leftmost

// negative element

if (i == 0 || i == n)

return;

// start with first positive

// element at index 0

// Rearrange array in alternating

// positive &

// negative items

int k = 0;

while (k < n && i < n ) {

// swap next positive

// element at even position

// from next negative element.

swap(arr[k], arr[i]);

i = i + 1;

k = k + 2;

}

}

// Utility function to print an array

void printArray(int arr[], int n)

{

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

cout << arr[i] << " ";

cout << endl;

}

// Driver code

int main()

{

int arr[] = { 2, 3, -4, -1, 6, -9 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Given array is \n";

printArray(arr, n);

rearrange(arr, n);

cout << "Rearranged array is \n";

printArray(arr, n);

return 0;

}

Output:

Given array is

2 3 -4 -1 6 -9

Rearranged array is

-1 3 -4 2 -9 6

Time Complexity : O(N)

Space Complexity : O(1)

### [Find if there is any subarray with sum equal to 0](https://practice.geeksforgeeks.org/problems/subarray-with-0-sum/0)

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

Example 1:

Input:

5

4 2 -3 1 6

Output:

Yes

Explanation:

2, -3, 1 is the subarray

with sum 0.

Example 2:

Input:

5

4 2 0 1 6

Output:

Yes

Explanation:

0 is one of the element

in the array so there exist a

subarray with sum 0.

Your Task:  
You only need to complete the function subArrayExists() that takes array and n as parameters and returns true or false depending upon whether there is a subarray present with 0-sum or not. Printing will be taken care by the drivers code.

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

Constraints:  
1 <= n <= 104  
-105 <= a[i] <= 105

## 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](https://www.cdn.geeksforgeeks.org/find-subarray-with-given-sum/)for implementation). The 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 the sum of elements from 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.   
Space Complexity: O(n) .Here we required extra space for unordered\_set to insert array elements.

### Find factorial of a large number

Given an integer N, find its factorial.  
  
Example 1:

Input: N = 5

Output: 120

Explanation : 5! = 1\*2\*3\*4\*5 = 120

Example 2:

Input: N = 10

Output: 3628800

Explanation :

10! = 1\*2\*3\*4\*5\*6\*7\*8\*9\*10 = 3628800

Your Task:  
You don't need to read input or print anything. Complete the function*factorial()*that takes integer N as input parameter and returns a list of integers denoting the digits that make up the factorial of N.

Expected Time Complexity : O(N2)  
Expected Auxilliary Space : O(1)

Constraints:  
1 ≤ N ≤ 1000

## Solution:

How to compute factorial of 100 using a C/C++ program?   
Factorial of 100 has 158 digits. It is not possible to store these many digits even if we use long long int.

Examples :

Input : 100

Output : 933262154439441526816992388562667004-

907159682643816214685929638952175999-

932299156089414639761565182862536979-

208272237582511852109168640000000000-

00000000000000

Input :50

Output : 3041409320171337804361260816606476884-

4377641568960512000000000000

Following is a simple solution where we use an array to store individual digits of the result. The idea is to use basic mathematics for multiplication.

The following is a detailed algorithm for finding factorial.  
*factorial(n)*   
1) Create a vector 'res[]'.   
2) Initialize value it with one element 1.   
3) Do following for all numbers from x = 2 to n.   
......a) Multiply x with res[] and update res[] and res\_size to store the multiplication result.

*How to multiply a number 'x' with the number stored in res[]?*   
The idea is to use simple school mathematics. We one by one multiply x with every digit of res[]. The important point to note here is digits are multiplied from rightmost digit to leftmost digit. If we store digits in same order in res[], then it becomes difficult to update res[] without extra space. That is why res[] is maintained in reverse way, i.e., digits from right to left are stored.

*multiply(res[], x)*   
1) Initialize carry as 0.   
2) Do following for i = 0 to res\_size - 1   
....a) Find value of res[i] \* x + carry. Let this value be prod.   
....b) Update res[i] by storing last digit of prod in it.   
....c) Update carry by storing remaining digits in carry.   
3) Put all digits of carry in res[] and increase res\_size by number of digits in carry.

Example to show working of multiply(res[], x)

A number 5189 is stored in res[] as following.

res[] = {9, 8, 1, 5}

x = 10

Initialize carry = 0;

i = 0, prod = res[0]\*x + carry = 9\*10 + 0 = 90.

res[0] = 0, carry = 9

i = 1, prod = res[1]\*x + carry = 8\*10 + 9 = 89

res[1] = 9, carry = 8

i = 2, prod = res[2]\*x + carry = 1\*10 + 8 = 18

res[2] = 8, carry = 1

i = 3, prod = res[3]\*x + carry = 5\*10 + 1 = 51

res[3] = 1, carry = 5

res[4] = carry = 5

res[] = {0, 9, 8, 1, 5}

Below is the implementation of the above algorithm.

vector<int> factorial(int N){

vector<int> res;

res.push\_back(1);

for(int x=2;x<=N;x++){

int vec\_size = res.size(), carry=0;

for(int i=0;i<vec\_size;i++){

int temp = x\*res[i]+carry;

res[i] = temp%10;

carry = temp/10;

}

while(carry!=0){

res.push\_back(carry%10);

carry /= 10;

}

}

reverse(res.begin(),res.end());

return res;

}

### find maximum product subarray

Given an array that contains both positive and negative integers, find the product of the maximum product subarray. Expected Time complexity is O(n) and only O(1) extra space can be used.

Examples:

Input: arr[] = {6, -3, -10, 0, 2}

Output: 180 // The subarray is {6, -3, -10}

Input: arr[] = {-1, -3, -10, 0, 60}

Output: 60 // The subarray is {60}

Input: arr[] = {-2, -40, 0, -2, -3}

Output: 80 // The subarray is {-2, -40}

## Naive Solution:

The idea is to traverse over every contiguous subarrays, find the product of each of these subarrays and return the maximum product from these results.

Below is the implementation of the above approach.

// C++ program to find Maximum Product Subarray

#include <bits/stdc++.h>

using namespace std;

/\* Returns the product of max product subarray.\*/

int maxSubarrayProduct(int arr[], int n)

{

// Initializing result

int result = arr[0];

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

{

int mul = arr[i];

// traversing in current subarray

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

{

// updating result every time

// to keep an eye over the maximum product

result = max(result, mul);

mul \*= arr[j];

}

// updating the result for (n-1)th index.

result = max(result, mul);

}

return result;

}

// Driver code

int main()

{

int arr[] = { 1, -2, -3, 0, 7, -8, -2 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Maximum Sub array product is "

<< maxSubarrayProduct(arr, n);

return 0;

}

Output:

Maximum Sub array product is 112

Time Complexity: O(N2)  
Auxiliary Space: O(1)

## Using two traversals

In this post an interesting solution is discussed. The idea is based on the fact that overall maximum product is maximum of following two: 

1. Maximum product in left to right traversal.
2. Maximum product in right to left traversal

For example, consider the above third sample input {-1, -2, -3, 4}. If we traverse the array only in forward direction (considering -1 as part of output), maximum product will be 2. If we traverse the array in backward direction (considering 4 as part of output), maximum product will be 24 i.e; { -2, -3, 4}.   
One important thing is to handle 0’s. We need to compute fresh forward (or backward) sum whenever we see 0.  
Below is the implementation of above idea : 

// C++ program to find maximum product subarray

#include<bits/stdc++.h>

using namespace std;

// Function for maximum product

int max\_product(int arr[], int n)

{

// Initialize maximum products in forward and

// backward directions

int max\_fwd = INT\_MIN, max\_bkd = INT\_MIN;

// Initialize current product

int max\_till\_now = 1;

//check if zero is present in an array or not

bool isZero=false;

// max\_fwd for maximum contiguous product in

// forward direction

// max\_bkd for maximum contiguous product in

// backward direction

// iterating within forward direction in array

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

{

// if arr[i]==0, it is breaking condition

// for contiguous subarray

max\_till\_now = max\_till\_now\*arr[i];

if (max\_till\_now == 0)

{

isZero=true;

max\_till\_now = 1;

continue;

}

if (max\_fwd < max\_till\_now) // update max\_fwd

max\_fwd = max\_till\_now;

}

max\_till\_now = 1;

// iterating within backward direction in array

for (int i=n-1; i>=0; i--)

{

max\_till\_now = max\_till\_now \* arr[i];

if (max\_till\_now == 0)

{

isZero=true;

max\_till\_now = 1;

continue;

}

// update max\_bkd

if (max\_bkd < max\_till\_now)

max\_bkd = max\_till\_now;

}

// return max of max\_fwd and max\_bkd

int res = max(max\_fwd, max\_bkd);

// Product should not be nagative.

// (Product of an empty subarray is

// considered as 0)

if(isZero)

return max(res, 0);

return res;

}

// Driver Program to test above function

int main()

{

int arr[] = {-1, -2, -3, 4};

int n = sizeof(arr)/sizeof(arr[0]);

cout << max\_product(arr, n) << endl;

return 0;

}

**Output:**

24

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

## Using single traversal

The idea is to traverse array from left to right keeping two variables minVal and maxVal which represents the minimum and maximum product value till the ith index of the array. Now, if the ith element of the array is negative that means now the values of minVal and maxVal will be swapped as value of maxVal will become minimum by multiplying it with a negative number. Now, compare the minVal and maxVal.   
The value of minVal and maxVal depends on the current index element or the product of the current index element and the previous minVal and maxVal respectively.  
Below is the implementation of above approach:

// C++ program to find maximum product subarray

#include <bits/stdc++.h>

using namespace std;

// Function to find maximum product subarray

int maxProduct(int\* arr, int n)

{

// Variables to store maximum and minimum

// product till ith index.

int minVal = arr[0];

int maxVal = arr[0];

int maxProduct = arr[0];

for (int i = 1; i < n; i++) {

// When multiplied by -ve number,

// maxVal becomes minVal

// and minVal becomes maxVal.

if (arr[i] < 0)

swap(maxVal, minVal);

// maxVal and minVal stores the

// product of subarray ending at arr[i].

maxVal = max(arr[i], maxVal \* arr[i]);

minVal = min(arr[i], minVal \* arr[i]);

// Max Product of array.

maxProduct = max(maxProduct, maxVal);

}

// Return maximum product found in array.

return maxProduct;

}

// Driver Code

int main()

{

int arr[] = { -1, -3, -10, 0, 60 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Maximum Subarray product is "

<< maxProduct(arr, n) << endl;

return 0;

}

Output:

Maximum Sub array product is 60

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

### 29. Find longest consecutive subsequence

Given an array of integers, find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.

Examples:

Input: arr[] = {1, 9, 3, 10, 4, 20, 2}

Output: 4

Explanation:

The subsequence 1, 3, 4, 2 is the longest

subsequence of consecutive elements

Input: arr[] = {36, 41, 56, 35, 44, 33, 34, 92, 43, 32, 42}

Output: 5

Explanation:

The subsequence 36, 35, 33, 34, 32 is the longest

subsequence of consecutive elements.

## Solution:

**Naive Approach:** The idea is to first sort the array and find the longest subarray with consecutive elements.   
After sorting the array and removing the multiple occurrences of elements, run a loop and keep a count and max (both initially zero). Run a loop from start to end and if the current element is not equal to the previous (element+1) then set the count to 1 else increase the count. Update max with a maximum of count and max.

// C++ program to find longest

// contiguous subsequence

#include <bits/stdc++.h>

using namespace std;

// Returns length of the longest

// contiguous subsequence

int findLongestConseqSubseq(int arr[], int n)

{

int ans = 0, count = 0;

// sort the array

sort(arr, arr + n);

vector<int> v;

v.push\_back(arr[0]);

//insert repeated elements only once in the vector

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

{

if (arr[i] != arr[i - 1])

v.push\_back(arr[i]);

}

// find the maximum length

// by traversing the array

for (int i = 0; i < v.size(); i++)

{

// Check if the current element is equal

// to previous element +1

if (i > 0 && v[i] == v[i - 1] + 1)

count++;

// reset the count

else

count = 1;

// update the maximum

ans = max(ans, count);

}

return ans;

}

// Driver code

int main()

{

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

int n = sizeof arr / sizeof arr[0];

cout << "Length of the Longest contiguous subsequence "

"is "

<< findLongestConseqSubseq(arr, n);

return 0;

}

Output

Length of the Longest contiguous subsequence is 3

Complexity Analysis:

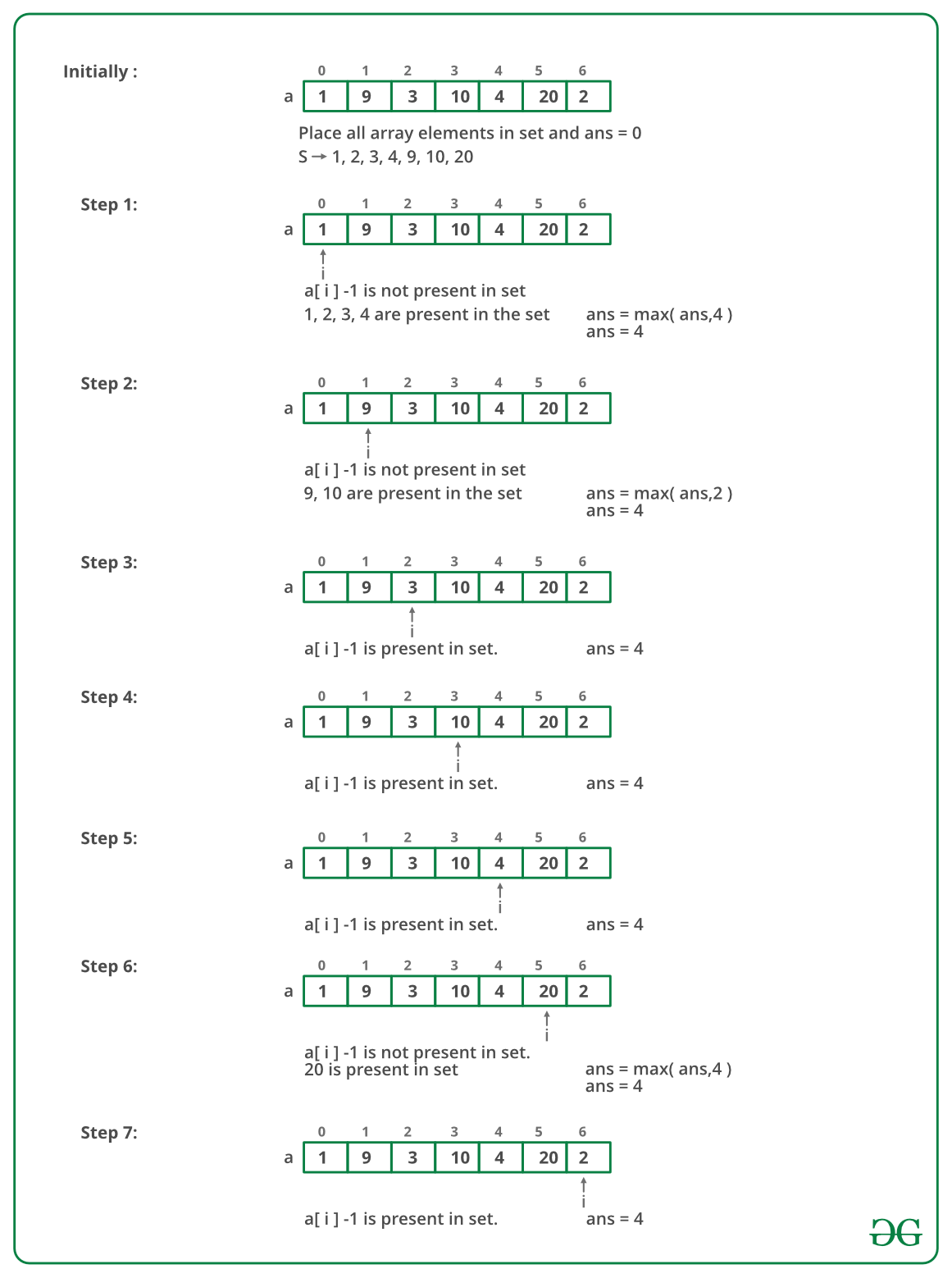
* Time complexity: O(nLogn).   
  Time to sort the array is O(nlogn).
* Auxiliary space : O(1).   
  As no extra space is needed.

**Efficient solution:**   
This problem can be solved in O(n) time using an **Efficient Solution**. The idea is to use [Hashing](http://geeksquiz.com/hashing-set-1-introduction/). We first insert all elements in a [Set](https://www.geeksforgeeks.org/set-in-cpp-stl/). Then check all the possible starts of consecutive subsequences.

**Algorithm:**

1. Create an empty hash.
2. Insert all array elements to hash.
3. Do following for every element arr[i]
4. Check if this element is the starting point of a subsequence. To check this, simply look for arr[i] – 1 in the hash, if not found, then this is the first element a subsequence.
5. If this element is the first element, then count the number of elements in the consecutive starting with this element. Iterate from arr[i] + 1 till the last element that can be found.
6. If the count is more than the previous longest subsequence found, then update this.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// C++ program to find longest

// contiguous subsequence

#include <bits/stdc++.h>

using namespace std;

// Returns length of the longest

// contiguous subsequence

int findLongestConseqSubseq(int arr[], int n)

{

unordered\_set<int> S;

int ans = 0;

// Hash all the array elements

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

S.insert(arr[i]);

// check each possible sequence from

// the start then update optimal length

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

{

// if current element is the starting

// element of a sequence

if (S.find(arr[i] - 1) == S.end())

{

// Then check for next elements

// in the sequence

int j = arr[i];

while (S.find(j) != S.end())

j++;

// update optimal length if

// this length is more

ans = max(ans, j - arr[i]);

}

}

return ans;

}

// Driver code

int main()

{

int arr[] = { 1, 9, 3, 10, 4, 20, 2 };

int n = sizeof arr / sizeof arr[0];

cout << "Length of the Longest contiguous subsequence "

"is "

<< findLongestConseqSubseq(arr, n);

return 0;

}

Output

Length of the Longest contiguous subsequence is 4

 Complexity Analysis:

* Time complexity: O(n).   
  Only one traversal is needed and the time complexity is O(n) under the assumption that hash insert and search take O(1) time.
* Auxiliary space: O(n).   
  To store every element in hashmap O(n) space is needed

### 30. Given an array of size n and a number k, find all elements that appear more than " n/k " times.

Given an array arr[] of size N and an element k. The task is to find all elements in array that appear more than n/k times.

Example 1:

Input:

N = 8

arr[] = {3,1,2,2,1,2,3,3}

k = 4

Output: 2

Explanation: In the given array, 3 and

2 are the only elements that appears

more than n/k times.

Example 2:

Input:

N = 4

arr[] = {2,3,3,2}

k = 3

Output: 2

Explanation: In the given array, 3 and 2

are the only elements that appears more

than n/k times. So the count of elements

are 2.

## Solution:

## Approach 1:

A **simple method** is to pick all elements one by one. For every picked element, count its occurrences by traversing the array, if count becomes more than n/k, then print the element. Time Complexity of this method would be O(n2).

Approach 2:  
A better solution is to **use sorting**. First, sort all elements using a O(nLogn) algorithm. Once the array is sorted, we can find all required elements in a linear scan of array. So overall time complexity of this method is O(nLogn) + O(n).

## Approach 3:

Following is an interesting **O(nk) solution:**

We can solve the above problem in O(nk) time using O(k-1) extra space. Note that there can never be more than k-1 elements in output (Why?). There are mainly three steps in this algorithm.

**1)** Create a temporary array of size (k-1) to store elements and their counts (The output elements are going to be among these k-1 elements). Following is structure of temporary array elements.

struct eleCount {

int element;

int count;

};

struct eleCount temp[];

This step takes O(k) time.   
**2)**Traverse through the input array and update temp[] (add/remove an element or increase/decrease count) for every traversed element. The array temp[] stores potential (k-1) candidates at every step. This step takes O(nk) time.

**3)** Iterate through final (k-1) potential candidates (stored in temp[]). or every element, check if it actually has count more than n/k. This step takes O(nk) time.

The main step is step 2, how to maintain (k-1) potential candidates at every point? The steps used in step 2 are like famous game: [Tetris](http://en.wikipedia.org/wiki/Tetris). We treat each number as a piece in Tetris, which falls down in our temporary array temp[]. Our task is to try to keep the same number stacked on the same column (count in temporary array is incremented).

Consider k = 4, n = 9

Given array: 3 1 2 2 2 1 4 3 3

i = 0

3 \_ \_

temp[] has one element, 3 with count 1

i = 1

3 1 \_

temp[] has two elements, 3 and 1 with

counts 1 and 1 respectively

i = 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 1 respectively.

i = 3

- - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 2 respectively.

i = 4

- - 2

- - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 3 respectively.

i = 5

- - 2

- 1 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 2 and 3 respectively.

Now the question arises, what to do when temp[] is full and we see a new element – we remove the bottom row from stacks of elements, i.e., we decrease count of every element by 1 in temp[]. We ignore the current element.

i = 6

- - 2

- 1 2

temp[] has two elements, 1 and 2 with

counts as 1 and 2 respectively.

i = 7

- 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 1, 1 and 2 respectively.

i = 8

3 - 2

3 1 2

temp[] has three elements, 3, 1 and 2 with

counts as 2, 1 and 2 respectively.

Finally, we have at most k-1 numbers in temp[]. The elements in temp are {3, 1, 2}. Note that the counts in temp[] are useless now, the counts were needed only in step 2. Now we need to check whether the actual counts of elements in temp[] are more than n/k (9/4) or not. The elements 3 and 2 have counts more than 9/4. So we print 3 and 2.

Note that the algorithm doesn’t miss any output element. There can be two possibilities, many occurrences are together or spread across the array. If occurrences are together, then count will be high and won’t become 0. If occurrences are spread, then the element would come again in temp[]. Following is the implementation of the above algorithm.

# A Python3 program to print elements with

# count more than n/k

# Prints elements with more than n/k

# occurrences in arrof size n. If

# there are no such elements, then

# it prints nothing.

def moreThanNdK(arr, n, k):

# k must be greater than 1

# to get some output

if (k < 2):

return

# Step 1: Create a temporary array

# (contains element and count) of

# size k-1. Initialize count of all

# elements as 0

temp = [[0 for i in range(2)]

for i in range(k)]

for i in range(k - 1):

temp[i][0] = 0

# Step 2: Process all elements

# of input array

for i in range(n):

j = 0

# If arr[i] is already present in

# the element count array, then

# increment its count

while j < k - 1:

if (temp[j][1] == arr[i]):

temp[j][0] += 1

break

j += 1

# If arr[i] is not present in temp

if (j == k - 1):

l = 0

# If there is position available

# in temp[], then place arr[i]

# in the first available position

# and set count as 1\*/

while l < k - 1:

if (temp[l][0] == 0):

temp[l][1] = arr[i]

temp[l][0] = 1

break

l += 1

# If all the position in the

# tempare filled, then decrease

# count of every element by 1

if (l == k - 1):

while l < k:

temp[l][0] -= 1

l += 1

# Step 3: Check actual counts

# of potential candidates in temp[]

for i in range(k - 1):

# Calculate actual count of elements

ac = 0 # Actual count

for j in range(n):

if (arr[j] == temp[i][1]):

ac += 1

# If actual count is more

# than n/k, then prit

if (ac > n // k):

print("Number:",

temp[i][1],

" Count:", ac)

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

print("First Test")

arr1 = [4, 5, 6, 7, 8, 4, 4]

size = len(arr1)

k = 3

moreThanNdK(arr1, size, k)

print("Second Test")

arr2 = [4, 2, 2, 7]

size = len(arr2)

k = 3

moreThanNdK(arr2, size, k)

print("Third Test")

arr3 = [2, 7, 2]

size = len(arr3)

k = 2

moreThanNdK(arr3, size, k)

print("Fourth Test")

arr4 = [2, 3, 3, 2]

size = len(arr4)

k = 3

moreThanNdK(arr4, size, k)

Output

First Test

Number:4 Count:3

Second Test

Number:2 Count:2

Third Test

Number:2 Count:2

Fourth Test

Number:2 Count:2

Number:3 Count:2

Time Complexity: O(nk)   
Auxiliary Space: O(k)  
Generally asked variations of this problem are, find all elements that appear n/3 times or n/4 times in O(n) time complexity and O(1) extra space.

## Approach 4:

Hashing can also be an efficient solution. With a good hash function, we can solve the above problem in O(n) time on average. Extra space required hashing would be higher than O(k). Also, hashing cannot be used to solve the above variations with O(1) extra space.

Below is the implementation of the above idea:

// C++ code to find elements whose

// frequency yis more than n/k

#include<bits/stdc++.h>

using namespace std;

void morethanNbyK(int arr[], int n, int k)

{

int x = n / k;

// unordered\_map initialization

unordered\_map<int, int> freq;

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

{

freq[arr[i]]++;

}

// Traversing the map

for(auto i : freq)

{

// Checking if value of a key-value pair

// is greater than x (where x=n/k)

if (i.second > x)

{

// Print the key of whose value

// is greater than x

cout << i.first << endl;

}

}

}

// Driver Code

int main()

{

int arr[] = { 1, 1, 2, 2, 3, 5,

4, 2, 2, 3, 1, 1, 1 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 4;

morethanNbyK(arr, n, k);

return 0;

}

### 26. Maximum profit by buying and selling a share atmost twice

In daily share trading, a buyer buys shares in the morning and sells them on the same day. If the trader is allowed to make at most 2 transactions in a day, whereas the second transaction can only start after the first one is complete (Buy->sell->Buy->sell). Given stock prices throughout the day, find out the maximum profit that a share trader could have made.

Examples:

Input:   price[] = {10, 22, 5, 75, 65, 80}  
Output:  87  
Trader earns 87 as sum of 12, 75   
Buy at 10, sell at 22,   
Buy at 5 and sell at 80  
Input:   price[] = {2, 30, 15, 10, 8, 25, 80}  
Output:  100  
Trader earns 100 as sum of 28 and 72  
Buy at price 2, sell at 30, buy at 8 and sell at 80  
Input:   price[] = {100, 30, 15, 10, 8, 25, 80};  
Output:  72  
Buy at price 8 and sell at 80.  
Input:   price[] = {90, 80, 70, 60, 50}  
Output:  0  
Not possible to earn.

## Solution:

## Approach 1

A Simple Solution is to consider every index ‘i’ and do the following

Max profit with at most two transactions =

MAX {max profit with one transaction and subarray price[0..i] +

max profit with one transaction and subarray price[i+1..n-1] }

i varies from 0 to n-1.

Maximum possible using one transaction can be calculated using O(n) algorithm we discussed in question 17.

The time complexity of the above simple solution is O(n2).

## Approach 2:

We can do this O(n) using the following Efficient Solution. The idea is to store the maximum possible profit of every subarray and solve the problem in the following two phases.

1) Create a table profit[0..n-1] and initialize all values in it 0.  
2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]  
3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].  
4) Return profit[n-1]

To do step 2, we need to keep track of the maximum price from right to left side, and to do step 3, we need to keep track of the minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in the third step, we use the same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After iteration i, the array profit[0..i] contains the maximum profit with 2 transactions, and profit[i+1..n-1] contains profit with two transactions.

Below are the implementations of the above idea.

// C++ program to find maximum

// possible profit with at most

// two transactions

#include <bits/stdc++.h>

using namespace std;

// Returns maximum profit with

// two transactions on a given

// list of stock prices, price[0..n-1]

int maxProfit(int price[], int n)

{

// Create profit array and

// initialize it as 0

int\* profit = new int[n];

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

profit[i] = 0;

/\* Get the maximum profit with

only one transaction

allowed. After this loop,

profit[i] contains maximum

profit from price[i..n-1]

using at most one trans. \*/

int max\_price = price[n - 1];

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

// max\_price has maximum

// of price[i..n-1]

if (price[i] > max\_price)

max\_price = price[i];

// we can get profit[i] by taking maximum of:

// a) previous maximum, i.e., profit[i+1]

// b) profit by buying at price[i] and selling at

// max\_price

profit[i]

= max(profit[i + 1], max\_price - price[i]);

}

/\* Get the maximum profit with two transactions allowed

After this loop, profit[n-1] contains the result \*/

int min\_price = price[0];

for (int i = 1; i < n; i++) {

// min\_price is minimum price in price[0..i]

if (price[i] < min\_price)

min\_price = price[i];

// Maximum profit is maximum of:

// a) previous maximum, i.e., profit[i-1]

// b) (Buy, Sell) at (min\_price, price[i]) and add

// profit of other trans. stored in profit[i]

profit[i] = max(profit[i - 1],

profit[i] + (price[i] - min\_price));

}

int result = profit[n - 1];

delete[] profit; // To avoid memory leak

return result;

}

// Driver code

int main()

{

int price[] = { 2, 30, 15, 10, 8, 25, 80 };

int n = sizeof(price) / sizeof(price[0]);

cout << "Maximum Profit = " << maxProfit(price, n);

return 0;

}

Output

Maximum Profit = 100

The time complexity of the above solution is O(n).

## Approach 3:

Algorithmic Paradigm: Dynamic Programming

There is one more approach for calculating this problem using the Valley-Peak approach i.e. take a variable profit and initialize it with zero and then traverse through the array of price[] from (i+1)th position whenever the initial position value is greater than the previous value add it to variable profit.

But this approach does not work in this problem, that you can only buy and sell a stock two times. For example, **if price [] = {2, 4, 2, 4, 2, 4}** then this particular approach will give result **6** while in this given problem you can only do two transactions so answer should be **4**. Hence **this approach only works when we have a chance to do infinite transaction.**

#include <iostream>

using namespace std;

int main()

{

int price[] = { 2, 30, 15, 10, 8, 25, 80 };

int n = 7;

// adding array

int profit = 0;

// Initializing variable

// valley-peak approach

/\*

80

/

30 /

/ \ 25

/ 15 /

/ \ /

2 10 /

\ /

8

\*/

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

{

// traversing through array from (i+1)th

// position

int sub = price[i] - price[i - 1];

if (sub > 0)

profit += sub;

}

cout << "Maximum Profit=" << profit;

return 0;

}

Output

Maximum Profit=100

The time and space complexity is O(n) and O(1) respectively.

## Approach 4:

Initialize four variables for taking care of the first buy, first sell, second buy, second sell. Set first buy and second buy as INT\_MIN and first and second sell as 0. This is to ensure to get profit from transactions. Iterate through the array and return the second buy as it will store maximum profit.

TIME COMPLEXITY : O(N)

SPACE COMPLEXITY : O(1)

#include <iostream>

#include<climits>

using namespace std;

int maxtwobuysell(int arr[],int size) {

int first\_buy = INT\_MIN;

int first\_sell = 0;

int second\_buy = INT\_MIN;

int second\_sell = 0;

for(int i=0;i<size;i++) {

first\_buy = max(first\_buy,-arr[i]);

first\_sell = max(first\_sell,first\_buy+arr[i]);

second\_buy = max(second\_buy,first\_sell-arr[i]);

second\_sell = max(second\_sell,second\_buy+arr[i]);

}

return second\_sell;

}

int main() {

int arr[] = {2, 30, 15, 10, 8, 25, 80};

int size = sizeof(arr)/sizeof(arr[0]);

cout<<maxtwobuysell(arr,size);

return 0;

}

Output

100

### 27. Find whether an array is a subset of another array

Given two arrays: a1[0..n-1] of size n and a2[0..m-1] of size m. Task is to check whether a2[] is a subset of a1[] or not. Both the arrays can be sorted or unsorted. It may be assumed that elements in both array are distinct.

Example 1:

Input:

a1[] = {11, 1, 13, 21, 3, 7}

a2[] = {11, 3, 7, 1}

Output:

Yes

Explanation:

a2[] is a subset of a1[]

Example 2:

Input:

a1[] = {1, 2, 3, 4, 5, 6}

a2[] = {1, 2, 4}

Output:

Yes

Explanation:

a2[] is a subset of a1[]

Example 3:

Input:

a1[] = {10, 5, 2, 23, 19}

a2[] = {19, 5, 3}

Output:

No

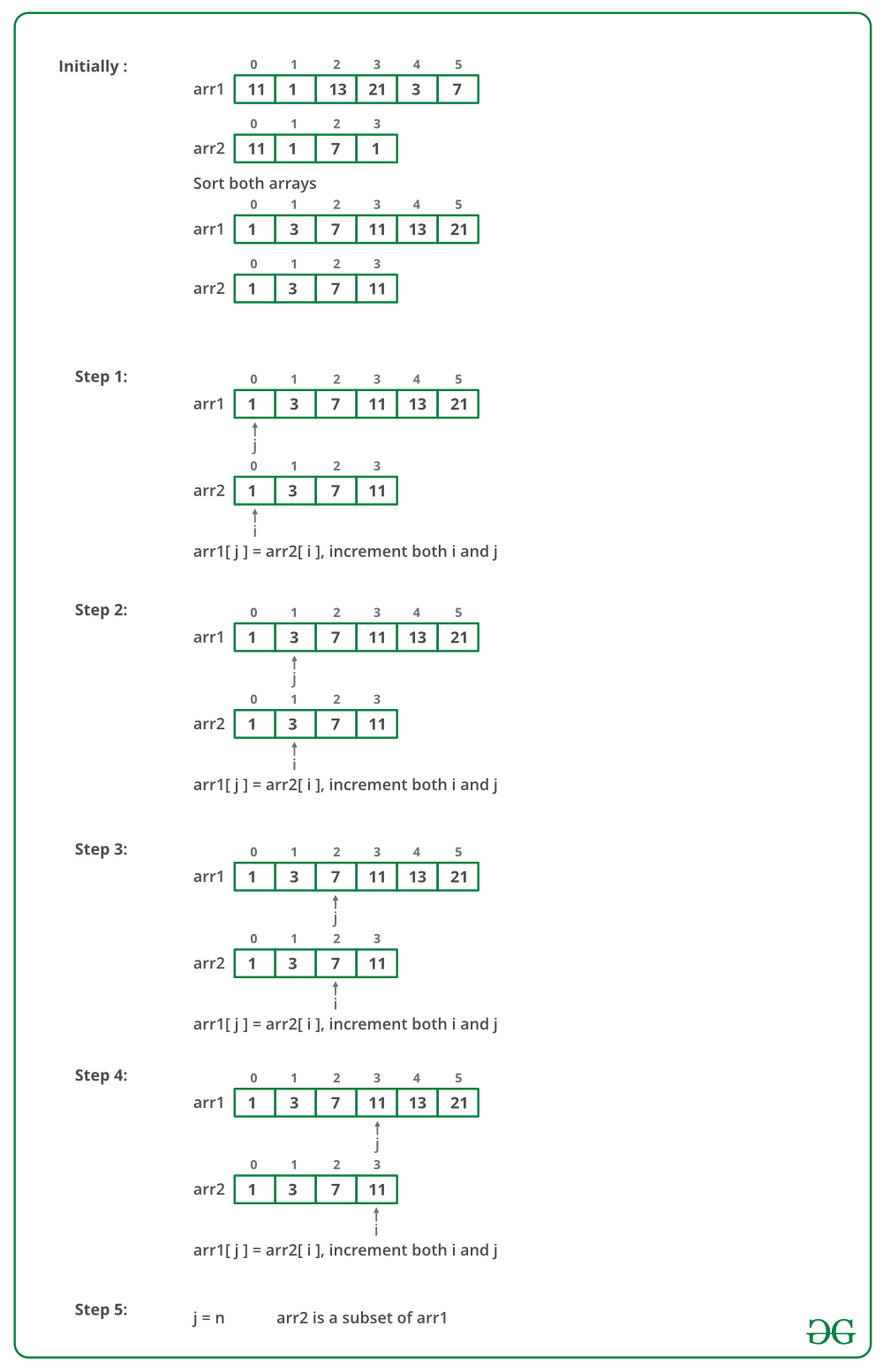
Explanation:

a2[] is not a subset of a1[]

Your Task:    
You don't need to read input or print anything. Your task is to complete the function isSubset() which takes the array a1[], a2[], its size n and m as inputs and return "Yes" if arr2 is subset of arr1 else return "No" if arr2 is not subset of arr1.

Expected Time Complexity: O(n)  
Expected Auxiliary Space: O(n)  
  
  
Constraints:  
1 <= n,m <= 105  
1 <= a1[i], a2[j] <= 105

## Approach1 (Sorting and Merging):

* Sort both arrays: arr1[] and arr2[] which takes O(mLogm + nLogn)
* Use Merge type of process to see if all elements of sorted arr2[] are present in sorted arr1[].  
  
* Below is the implementation of the above approach:

// C++ program to find whether an array

// is subset of another array

#include <bits/stdc++.h>

using namespace std;

/\* Return 1 if arr2[] is a subset of arr1[] \*/

bool isSubset(int arr1[], int arr2[],

int m, int n)

{

int i = 0, j = 0;

if (m < n)

return 0;

// Sort both the arrays

sort(arr1, arr1 + m);

sort(arr2, arr2 + n);

// Iterate till they donot exceed their sizes

while (i < n && j < m)

{

// If the element is smaller than

// Move aheaad in the first array

if (arr1[j] < arr2[i])

j++;

// If both are equal, then move

// both of them forward

else if (arr1[j] == arr2[i])

{

j++;

i++;

}

// If we donot have a element smaller

// or equal to the second array then break

else if (arr1[j] > arr2[i])

return 0;

}

return (i < n) ? false : true;

}

// Driver Code

int main()

{

int arr1[] = { 11, 1, 13, 21, 3, 7 };

int arr2[] = { 11, 3, 7, 1 };

int m = sizeof(arr1) / sizeof(arr1[0]);

int n = sizeof(arr2) / sizeof(arr2[0]);

if (isSubset(arr1, arr2, m, n))

printf("arr2[] is subset of arr1[] ");

else

printf("arr2[] is not a subset of arr1[] ");

return 0;

}

Output

arr2[] is subset of arr1[]

Time Complexity: O(mLogm + nLogn) which is better than method 2. Please note that this will be the complexity if an nLogn algorithm is used for sorting both arrays which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is O(n^2)

## Approach (Use Frequency table)

* Create a Frequency Table for all the elements of arr1[].
* Traverse arr2[] and search for each element of arr2[] in the Frequency Table. if element is found decrease the frequency, If element frequency is not found then return 0.
* If all elements are found then return 1.

Below is the implementation of the above approach:

// C++ program to find whether an array

// is subset of another array

#include <bits/stdc++.h>

using namespace std;

/\* Return true if arr2[] is a subset of arr1[] \*/

bool isSubset(int arr1[], int m,

int arr2[], int n)

{

// Create a Frequency Table using STL

map<int, int> frequency;

// Increase the frequency of each element

// in the frequency table.

for (int i = 0; i < m; i++)

{

frequency[arr1[i]]++;

}

// Decrease the frequency if the

// element was found in the frequency

// table with the frequency more than 0.

// else return 0 and if loop is

// completed return 1.

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

{

if (frequency[arr2[i]] > 0)

frequency[arr2[i]]--;

else

{

return false;

}

}

return true;

}

// Driver Code

int main()

{

int arr1[] = { 11, 1, 13, 21, 3, 7 };

int arr2[] = { 11, 3, 7, 1 };

int m = sizeof(arr1) / sizeof(arr1[0]);

int n = sizeof(arr2) / sizeof(arr2[0]);

if (isSubset(arr1, m, arr2, n))

cout << "arr2[] is subset of arr1[] "

<< "\n";

else

cout << "arr2[] is not a subset of arr1[] "

<< "\n";

return 0;

}

Output

arr2[] is subset of arr1[]

**Time Complexity:**O(m+n)

### 28. Find the 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.

Examples: 

*Input: array = {12, 3, 4, 1, 6, 9}, sum = 24;   
Output: 12, 3, 9   
Explanation: There is a triplet (12, 3 and 9) present   
in the array whose sum is 24.   
Input: array = {1, 2, 3, 4, 5}, sum = 9   
Output: 5, 3, 1   
Explanation: There is a triplet (5, 3 and 1) present   
in the array whose sum is 9.*

## Solution:

Method 1: This is the naive approach towards solving the above problem.

* Approach: 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.
* Algorithm:
  1. Given an array of length *n* and a sum *s*
  2. Create three nested loop first loop runs from start to end (loop counter i), second loop runs from i+1 to end (loop counter j) and third loop runs from j+1 to end (loop counter k)
  3. The counter of these loops represents the index of 3 elements of the triplets.
  4. Find the sum of ith, jth and kth element. If the sum is equal to given sum. Print the triplet and break.
  5. If there is no triplet, then print that no triplet exist.
* Implementation:

#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;

}

**Output**

Triplet is 4, 10, 8

* **Complexity Analysis:**
  + **Time Complexity:** O(n3).   
    There are three nested loops traversing the array, so the time complexity is O(n^3)
  + **Space Complexity:**O(1).   
    As no extra space is required.

**Method 2:** This method uses sorting to increase the efficiency of the code.

* **Approach:** By Sorting the array the efficiency of the algorithm can be improved. This efficient approach uses the [two-pointer technique](https://www.geeksforgeeks.org/given-an-array-a-and-a-number-x-check-for-pair-in-a-with-sum-as-x/). Traverse the array and fix the first element of the triplet. Now use the Two Pointers algorithm to find if there is a pair whose sum is equal to x – array[i]. Two pointers algorithm take linear time so it is better than a nested loop.
* **Algorithm :**
  1. Sort the given array.
  2. Loop over the array and fix the first element of the possible triplet, arr[i].
  3. Then fix two pointers, one at i + 1 and the other at n – 1. And look at the sum,
     1. If the sum is smaller than the required sum, increment the first pointer.
     2. Else, If the sum is bigger, Decrease the end pointer to reduce the sum.
     3. Else, if the sum of elements at two-pointer is equal to given sum then print the triplet and break.
* **Implementation:**

// 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

* Complexity Analysis:
  + Time complexity: O(N^2).   
    There are only two nested loops traversing the array, so time complexity is O(n^2). Two pointers algorithm takes O(n) time and the first element can be fixed using another nested traversal.
  + Space Complexity: O(1).   
    As no extra space is required.

### 29. Trapping Rain water problem

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.

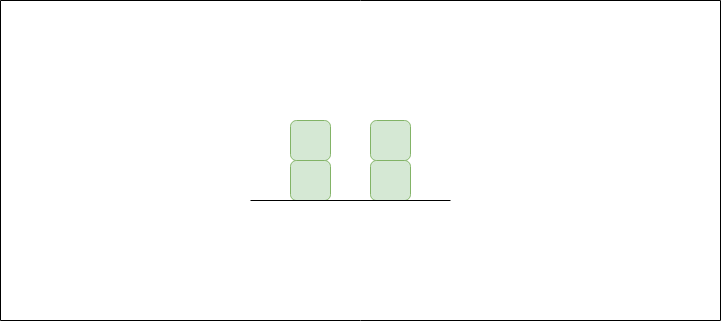
Examples:

Input: arr[] = {2, 0, 2}

Output: 2

Explanation:

The structure is like below



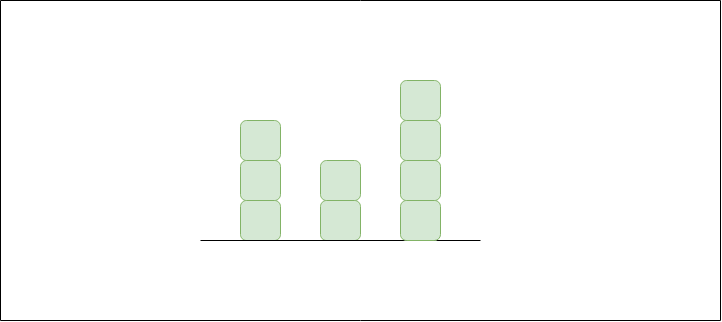
We can trap 2 units of water in the middle gap.

Input: arr[] = {3, 0, 2, 0, 4}

Output: 7

Explanation:

Structure is like below



We can trap "3 units" of water between 3 and 2,

"1 unit" on top of bar 2 and "3 units" between 2

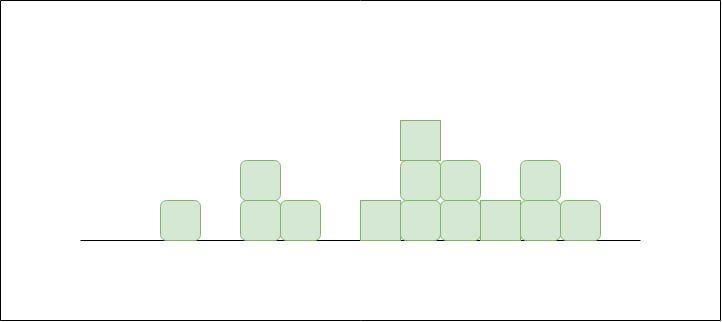
and 4. See below diagram also.

Input: arr[] = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

Output: 6

Explanation:

The structure is like below



Trap "1 unit" between first 1 and 2, "4 units" between

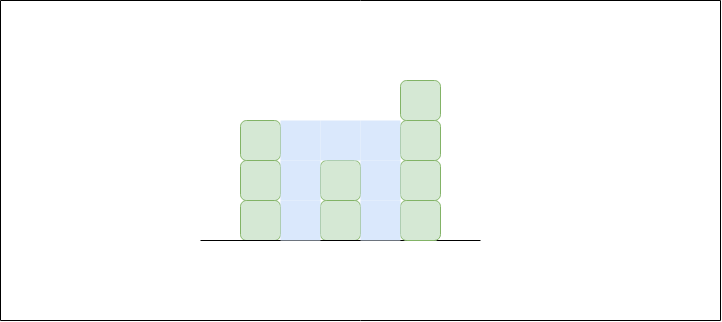
first 2 and 3 and "1 unit" between second last 1 and last 2

## Solution:

Basic Insight:   
An element of the array can store water if there are higher bars on the left and right. The amount of water to be stored in every element can be found out by finding the heights of bars on the left and right sides. The idea is to compute the amount of water that can be stored in every element of the array.

Example   
Consider the array {3, 0, 2, 0, 4}, three units of water can be stored three indexes 1 and 2, and one unit of water at index 3, and three units of water at index 4.

*For Array[] = {3, 0, 2, 0, 4}   
Water stored = 0 + 3 + 1 + 3 + 0 = 7*

**

Method 1 : This is a simple solution to the above problem.

* Approach: The idea is to traverse every array element and find the highest bars on the 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.
* Algorithm:
  1. Traverse the array from start to end.
  2. For every element, traverse the array from start to that index and find the maximum height *(a)* and traverse the array from the current index to end, and find the maximum height *(b)*.
  3. The amount of water that will be stored in this column is *min(a,b) – array[i]*, add this value to the total amount of water stored
  4. Print the total amount of water stored.
* Implementation:

// 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

* Complexity Analysis:
  + Time Complexity: O(n2).   
    There are two nested loops traversing the array, So time Complexity is O(n2).
  + Space Complexity: O(1).   
    No extra space is required.

Method 2: This is an efficient solution to the above problem.

* Approach: In the previous solution, to find the highest bar on the left and right, array traversal is needed which reduces the efficiency of the solution. To make this efficient one must pre-compute the highest bar on the left and right of every bar in linear time. Then use these pre-computed values to find the amount of water in every array element.
* Algorithm:
  1. Create two arrays *left* and *right* of size n. create a variable *max\_ = INT\_MIN*.
  2. Run one loop from start to end. In each iteration update max\_ as *max\_ = max(max\_, arr[i])* and also assign *left[i] = max\_*
  3. Update max\_ = INT\_MIN.
  4. Run another loop from end to start. In each iteration update max\_ as *max\_ = max(max\_, arr[i])* and also assign *right[i] = max\_*
  5. Traverse the array from start to end.
  6. The amount of water that will be stored in this column is *min(a,b) – array[i]*,(where a = left[i] and b = right[i]) add this value to total amount of water stored
  7. Print the total amount of water stored.
* Implementation:

// 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

* Complexity Analysis:
  + Time Complexity: O(n).   
    Only one traversal of the array is needed, So time Complexity is O(n).
  + Space Complexity: O(n).   
    Two extra arrays are needed each of size n.

Space Optimization for the above Solution:

Instead of maintaining two arrays of size n for storing the left and a right max of each element, maintain two variables to store the maximum till that point. Since water trapped at any element = *min(max\_left, max\_right) – arr[i]*. Calculate water trapped on smaller elements out of A[lo] and A[hi] first and move the pointers till *lo* doesn’t cross *hi*.

* Implementation:

// 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

* Complexity Analysis:
  + Time Complexity: O(n).   
    Only one traversal of the array is needed.
  + Auxiliary Space: O(1).   
    As no extra space is required.

Method 3: Here another efficient solution has been shown.

* Approach: The concept here is that if there is a larger wall to the right then the water can be retained with a 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. Let’s take an example of the heights are {….,3, 2, 1, 4,….}, So here 3 and 4 are boundaries the heights 2 and 1 are submerged and cannot act as boundaries. So at any point or index knowing the previous boundary is sufficient if there is a higher or equal length boundary in the remaining part of the array. If not then traverse the array backward and now must be a larger wall to the left.
* Algorithm:
  + 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 the 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 the 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 backward).
* Implementation:

// C++ implementation of the approach

#include<iostream>

using namespace std;

// Function to return the maximum

// water that can be stored

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 variable

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

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;

}

// This code is contributed by Debidutta Rath

Output

6

* Complexity Analysis:
  + Time Complexity: O(n).   
    As only one traversal of the array is needed.
  + Auxiliary Space: O(1).   
    As no extra space is required.

Method 4 (Using Stack):

We can use a Stack to track the bars that are bounded by the longer left and right bars. This can be done using only one iteration using Stacks.

Approach:

1. Loop through the indices of the bar array.

2. For each bar, we can do the following:

* While the Stack is not empty and the current bar has a height greater than the top bar of the stack,
* Store the index of the top bar in pop\_height and pop it from the Stack.
* Find the distance between the left bar(current top) of the popped bar and the current bar.
* Find the minimum height between the top bar and the current bar.
* The maximum water that can be trapped in distance \* min\_height.
* The water trapped including the popped bar is (distance \* min\_height) – height[pop\_height].
* Add that to the fans.

3. Final answer will the ans.

// 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 height[], int n)

{

// Stores the indices of the bars

stack <int> st;

// Stores the final result

int ans = 0;

// Loop through the each bar

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

{

// Remove bars from the stack

// until the condition holds

while ((!st.empty()) &&

(height[st.top()] < height[i]))

{

// Store the height of the top

// and pop it.

int pop\_height = height[st.top()];

st.pop();

// If the stack does not have any

// bars or the the popped bar

// has no left boundary

if (st.empty())

break;

// Get the distance between the

// left and right boundary of

// popped bar

int distance = i - st.top() - 1;

// Calculate the min. height

int min\_height = min(height[st.top()],

height[i]) -

pop\_height;

ans += distance \* min\_height;

}

// If the stack is either empty or

// height of the current bar is less than

// or equal to the top bar of stack

st.push(i);

}

return ans;

}

// 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;

}

// The code is contributed by Soumitri Chattopadhyay

Output:

6

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

### 30. Chocolate Distribution problem

Given an array A[ ] of positive integers of size N, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are M students, the task is to distribute chocolate packets among M students such that :  
1. Each student gets exactly one packet.  
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

Example 1:

Input:

N = 8, M = 5

A = {3, 4, 1, 9, 56, 7, 9, 12}

Output: 6

Explanation: The minimum difference between

maximum chocolates and minimum chocolates

is 9 - 3 = 6 by choosing following M packets :

{3, 4, 9, 7, 9}.

Example 2:

Input:

N = 7, M = 3

A = {7, 3, 2, 4, 9, 12, 56}

Output: 2

Explanation: The minimum difference between

maximum chocolates and minimum chocolates

is 4 - 2 = 2 by choosing following M packets :

{3, 2, 4}.

Your Task:  
You don't need to take any input or print anything. Your task is to complete the function findMinDiff() which takes array A[ ], N and M as input parameters and returns the minimum possible difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student.

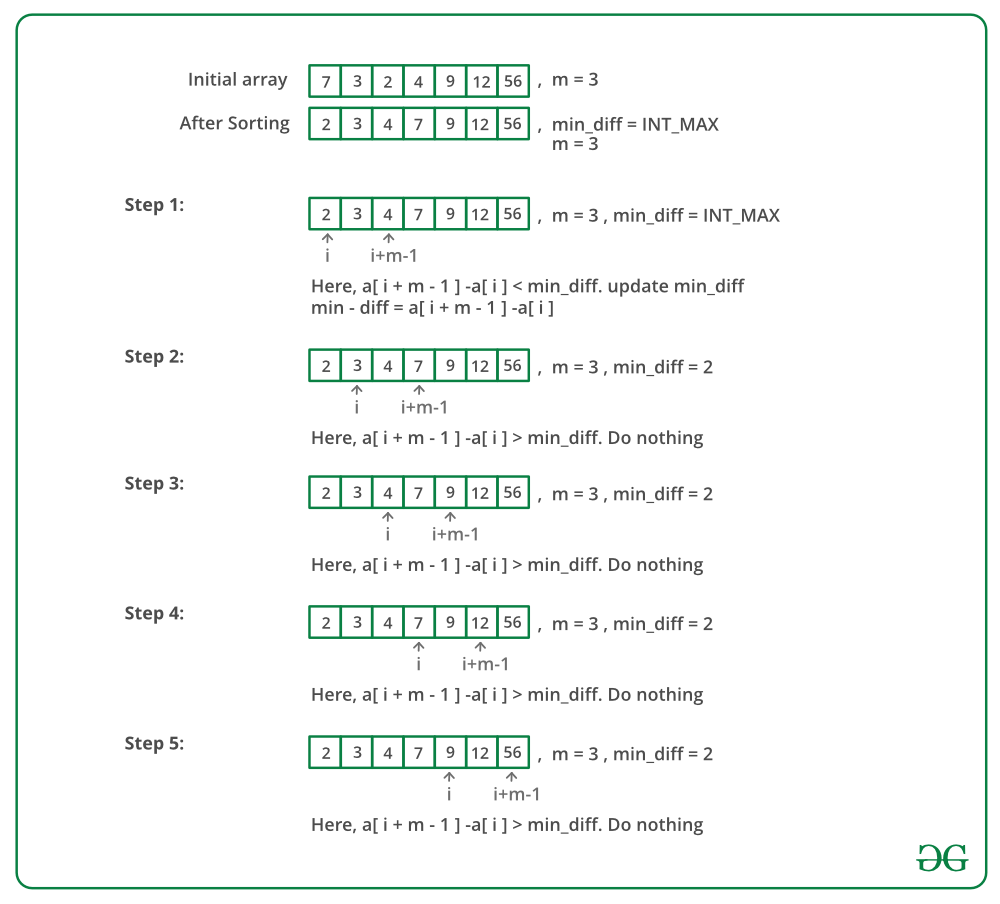
Expected Time Complexity: O(N\*Log(N))  
Expected Auxiliary Space: O(1)

Constraints:  
1 ≤ T ≤ 100  
1 ≤ N ≤ 105  
1 ≤ Ai ≤ 109  
1 ≤ M ≤ N

## Solution:

A simple solution is to generate all subsets of size m of arr[0..n-1]. For every subset, find the difference between the maximum and minimum elements in it. Finally, return the minimum difference.  
An efficient solution is based on the observation that to minimize the difference, we must choose consecutive elements from a sorted packet. We first sort the array arr[0..n-1], then find the subarray of size m with the minimum difference between the last and first elements.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// C++ program to solve chocolate distribution

// problem

#include <bits/stdc++.h>

using namespace std;

// arr[0..n-1] represents sizes of packets

// m is number of students.

// Returns minimum difference between maximum

// and minimum values of distribution.

int findMinDiff(int arr[], int n, int m)

{

// if there are no chocolates or number

// of students is 0

if (m == 0 || n == 0)

return 0;

// Sort the given packets

sort(arr, arr + n);

// Number of students cannot be more than

// number of packets

if (n < m)

return -1;

// Largest number of chocolates

int min\_diff = INT\_MAX;

// Find the subarray of size m such that

// difference between last (maximum in case

// of sorted) and first (minimum in case of

// sorted) elements of subarray is minimum.

for (int i = 0; i + m - 1 < n; i++) {

int diff = arr[i + m - 1] - arr[i];

if (diff < min\_diff)

min\_diff = diff;

}

return min\_diff;

}

int main()

{

int arr[] = { 12, 4, 7, 9, 2, 23, 25, 41, 30,

40, 28, 42, 30, 44, 48, 43, 50 };

int m = 7; // Number of students

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Minimum difference is "

<< findMinDiff(arr, n, m);

return 0;

}

Output

Minimum difference is 10

Time Complexity: O(n Log n) as we apply sorting before subarray search.

### 31. Smallest Subarray with sum greater than a given value

Given an array of integers (A[])  and a number x, find the smallest subarray with sum greater than the given value.

Note: The answer always exists. It is guaranteed that x doesn't exceed the summation of a[i] (from 1 to N).

Example 1:

Input:

A[] = {1, 4, 45, 6, 0, 19}

x = 51

Output: 3

Explanation:

Minimum length subarray is

{4, 45, 6}

Example 2:

Input:

A[] = {1, 10, 5, 2, 7}

x = 9

Output: 1

Explanation:

Minimum length subarray is {10}

Your Task:    
You don't need to read input or print anything. Your task is to complete the function smallestSubWithSum() which takes the array A[], its size N and an integer X as inputs and returns the required ouput.

Expected Time Complexity: O(N)  
Expected Auxiliary Space: O(1)  
  
Constraints:  
1 ≤ N, x ≤ 105  
1 ≤ A[] ≤ 104

## Solution:

A **simple solution**is to use two nested loops. The outer loop picks a starting element, the inner loop considers all elements (on right side of current start) as ending element. Whenever sum of elements between current start and end becomes more than the given number, update the result if current length is smaller than the smallest length so far.   
Following is the implementation of simple approach.

# include <iostream>

using namespace std;

// Returns length of smallest subarray with sum greater than x.

// If there is no subarray with given sum, then returns n+1

int smallestSubWithSum(int arr[], int n, int x)

{

// Initialize length of smallest subarray as n+1

int min\_len = n + 1;

// Pick every element as starting point

for (int start=0; start<n; start++)

{

// Initialize sum starting with current start

int curr\_sum = arr[start];

// If first element itself is greater

if (curr\_sum > x) return 1;

// Try different ending points for curremt start

for (int end=start+1; end<n; end++)

{

// add last element to current sum

curr\_sum += arr[end];

// If sum becomes more than x and length of

// this subarray is smaller than current smallest

// length, update the smallest length (or result)

if (curr\_sum > x && (end - start + 1) < min\_len)

min\_len = (end - start + 1);

}

}

return min\_len;

}

/\* Driver program to test above function \*/

int main()

{

int arr1[] = {1, 4, 45, 6, 10, 19};

int x = 51;

int n1 = sizeof(arr1)/sizeof(arr1[0]);

int res1 = smallestSubWithSum(arr1, n1, x);

(res1 == n1+1)? cout << "Not possible\n" :

cout << res1 << endl;

int arr2[] = {1, 10, 5, 2, 7};

int n2 = sizeof(arr2)/sizeof(arr2[0]);

x = 9;

int res2 = smallestSubWithSum(arr2, n2, x);

(res2 == n2+1)? cout << "Not possible\n" :

cout << res2 << endl;

int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};

int n3 = sizeof(arr3)/sizeof(arr3[0]);

x = 280;

int res3 = smallestSubWithSum(arr3, n3, x);

(res3 == n3+1)? cout << "Not possible\n" :

cout << res3 << endl;

return 0;

}

**Output:**

3

1

4

**Time Complexity:**Time complexity of the above approach is clearly O(n2).

**Efficient Solution:** This problem can be solved in **O(n) time** using the idea used in [this](https://www.cdn.geeksforgeeks.org/find-subarray-with-given-sum/)post. Thanks to Ankit and Nitin for suggesting this optimized solution.

// O(n) solution for finding smallest subarray with sum

// greater than x

#include <iostream>

using namespace std;

// Returns length of smallest subarray with sum greater than

// x. If there is no subarray with given sum, then returns

// n+1

int smallestSubWithSum(int arr[], int n, int x)

{

// Initialize current sum and minimum length

int curr\_sum = 0, min\_len = n + 1;

// Initialize starting and ending indexes

int start = 0, end = 0;

while (end < n) {

// Keep adding array elements while current sum

// is smaller than or equal to x

while (curr\_sum <= x && end < n)

curr\_sum += arr[end++];

// If current sum becomes greater than x.

while (curr\_sum > x && start < n) {

// Update minimum length if needed

if (end - start < min\_len)

min\_len = end - start;

// remove starting elements

curr\_sum -= arr[start++];

}

}

return min\_len;

}

/\* Driver program to test above function \*/

int main()

{

int arr1[] = { 1, 4, 45, 6, 10, 19 };

int x = 51;

int n1 = sizeof(arr1) / sizeof(arr1[0]);

int res1 = smallestSubWithSum(arr1, n1, x);

(res1 == n1 + 1) ? cout << "Not possible\n"

: cout << res1 << endl;

int arr2[] = { 1, 10, 5, 2, 7 };

int n2 = sizeof(arr2) / sizeof(arr2[0]);

x = 9;

int res2 = smallestSubWithSum(arr2, n2, x);

(res2 == n2 + 1) ? cout << "Not possible\n"

: cout << res2 << endl;

int arr3[] = { 1, 11, 100, 1, 0, 200, 3, 2, 1, 250 };

int n3 = sizeof(arr3) / sizeof(arr3[0]);

x = 280;

int res3 = smallestSubWithSum(arr3, n3, x);

(res3 == n3 + 1) ? cout << "Not possible\n"

: cout << res3 << endl;

return 0;

}

Output

3

1

4

### 32. Three way partitioning of an array around a given value

Given an array and a range [lowVal, highVal], partition the array around the range such that array is divided in three parts.   
1) All elements smaller than lowVal come first.   
2) All elements in range lowVal to highVVal come next.   
3) All elements greater than highVVal appear in the end.   
The individual elements of three sets can appear in any order.

Examples:

Input: arr[] = {1, 14, 5, 20, 4, 2, 54, 20, 87, 98, 3, 1, 32}

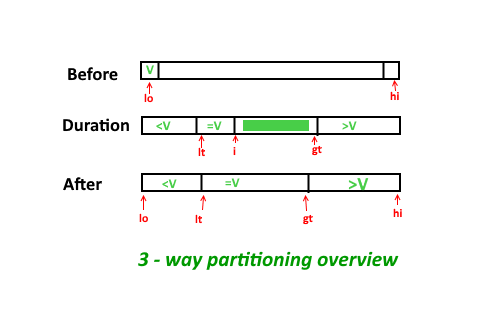
lowVal = 14, highVal = 20

Output: arr[] = {1, 5, 4, 2, 1, 3, 14, 20, 20, 98, 87, 32, 54}

Input: arr[] = {1, 14, 5, 20, 4, 2, 54, 20, 87, 98, 3, 1, 32}

lowVal = 20, highVal = 20

Output: arr[] = {1, 14, 5, 4, 2, 1, 3, 20, 20, 98, 87, 32, 54}



## Solution:

A **simple solution**is to sort the array. This solution does a lot of extra rearrangements and requires O(n Log n) time.  
An **efficient solution** is based on [Dutch National Flag based QuickSort](https://www.geeksforgeeks.org/3-way-quicksort-dutch-national-flag/). We traverse given array elements from left. We keep track of two pointers, first (called start in below code) to store next position of smaller element (smaller than range) from beginning; and second (called end in below code) to store next position of greater element from end.

// C++ program to implement three way partitioning

// around a given range.

#include<iostream>

using namespace std;

// Partitions arr[0..n-1] around [lowVal..highVal]

void threeWayPartition(int arr[], int n,

int lowVal, int highVal)

{

// Initialize ext available positions for

// smaller (than range) and greater lements

int start = 0, end = n-1;

// Traverse elements from left

for (int i=0; i<=end;)

{

// If current element is smaller than

// range, put it on next available smaller

// position.

if (arr[i] < lowVal)

swap(arr[i++], arr[start++]);

// If current element is greater than

// range, put it on next available greater

// position.

else if (arr[i] > highVal)

swap(arr[i], arr[end--]);

else

i++;

}

}

// Driver code

int main()

{

int arr[] = {1, 14, 5, 20, 4, 2, 54, 20, 87,

98, 3, 1, 32};

int n = sizeof(arr)/sizeof(arr[0]);

threeWayPartition(arr, n, 10, 20);

cout << "Modified array \n";

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

cout << arr[i] << " ";

}

**Output:**

Modified array

1 5 4 2 1 3 14 20 20 98 87 32 54

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

### 33. Minimum swaps required bring elements less equal K together

Given an array of n positive integers and a number **k**. Find the **minimum** number of swaps required to bring all the numbers less than or equal to **k** together.

Input: arr[] = {2, 1, 5, 6, 3}, k = 3

Output: 1

Explanation:

To bring elements 2, 1, 3 together, swap

element '5' with '3' such that final array

will be-

arr[] = {2, 1, 3, 6, 5}

Input: arr[] = {2, 7, 9, 5, 8, 7, 4}, k = 5

Output: 2

## Solution:

A **simple solution** is to first count all elements less than or equals to **k**(say ‘good’). Now traverse for every sub-array and swap those elements whose value is greater than **k**. Time complexity of this approach is O(n2)  
A **simple approach** is to use [two pointer technique](https://www.geeksforgeeks.org/two-pointers-technique/) and [sliding window](https://www.geeksforgeeks.org/window-sliding-technique/). 

1. Find count of all elements which are less than or equals to ‘k’. Let’s say the count is ‘cnt’
2. Using two pointer technique for window of length ‘cnt’, each time keep track of how many elements in this range are greater than ‘k’. Let’s say the total count is ‘bad’.
3. Repeat step 2, for every window of length ‘cnt’ and take minimum of count ‘bad’ among them. This will be the final answer.

// C++ program to find minimum swaps required

// to club all elements less than or equals

// to k together

#include <iostream>

using namespace std;

// Utility function to find minimum swaps

// required to club all elements less than

// or equals to k together

int minSwap(int \*arr, int n, int k) {

// Find count of elements which are

// less than equals to k

int count = 0;

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

if (arr[i] <= k)

++count;

// Find unwanted elements in current

// window of size 'count'

int bad = 0;

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

if (arr[i] > k)

++bad;

// Initialize answer with 'bad' value of

// current window

int ans = bad;

for (int i = 0, j = count; j < n; ++i, ++j) {

// Decrement count of previous window

if (arr[i] > k)

--bad;

// Increment count of current window

if (arr[j] > k)

++bad;

// Update ans if count of 'bad'

// is less in current window

ans = min(ans, bad);

}

return ans;

}

// Driver code

int main() {

int arr[] = {2, 1, 5, 6, 3};

int n = sizeof(arr) / sizeof(arr[0]);

int k = 3;

cout << minSwap(arr, n, k) << "\n";

int arr1[] = {2, 7, 9, 5, 8, 7, 4};

n = sizeof(arr1) / sizeof(arr1[0]);

k = 5;

cout << minSwap(arr1, n, k);

return 0;

}

Time Complexity: O(n)

Space Complexity: O(1)

### 34. Minimum no. of operations required to make an array palindrome

# Find minimum number of merge operations to make an array palindrome

Given an array of positive integers. We need to make the given array a ‘Palindrome’. The only allowed operation is”merging” (of two adjacent elements). Merging two adjacent elements means replacing them with their sum. The task is to find the minimum number of merge operations required to make the given array a ‘Palindrome’.  
To make any array a palindrome, we can simply apply merge operation n-1 times where n is the size of the array (because a single-element array is always palindromic, similar to single-character string). In that case, the size of array will be reduced to 1. But in this problem, we are asked to do it in the minimum number of operations.

**Example :**

Input : arr[] = {15, 4, 15}

Output : 0

Array is already a palindrome. So we

do not need any merge operation.

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

Output : 1

We can make given array palindrome with

minimum one merging (merging 4 and 5 to

make 9)

Input : arr[] = {11, 14, 15, 99}

Output : 3

We need to merge all elements to make

a palindrome.

The expected time complexity is O(n).

## Solution:

Let f(i, j) be minimum merging operations to make subarray arr[i..j] a palindrome. If i == j answer is 0. We start i from 0 and j from n-1.

1. If arr[i] == arr[j], then there is no need to do any merging operations at index i or index j. Our answer in this case will be f(i+1, j-1).
2. Else, we need to do merging operations. Following cases arise.
   * If arr[i] > arr[j], then we should do merging operation at index j. We merge index j-1 and j, and update arr[j-1] = arr[j-1] + arr[j]. Our answer in this case will be 1 + f(i, j-1).
   * For the case when arr[i] < arr[j], update arr[i+1] = arr[i+1] + arr[i]. Our answer in this case will be 1 + f(i+1, j).
3. Our answer will be f(0, n-1), where n is the size of array arr[].

Therefore this problem can be solved iteratively using two pointers (first pointer pointing to start of the array and second pointer pointing to the last element of the array) method and keeping count of total merging operations done till now.  
Below is an implementation of the above idea.

 // C++ program to find number of operations

// to make an array palindrome

#include <bits/stdc++.h>

using namespace std;

// Returns minimum number of count operations

// required to make arr[] palindrome

int findMinOps(int arr[], int n)

{

int ans = 0; // Initialize result

// Start from two corners

for (int i=0,j=n-1; i<=j;)

{

// If corner elements are same,

// problem reduces arr[i+1..j-1]

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

{

i++;

j--;

}

// If left element is greater, then

// we merge right two elements

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

{

// need to merge from tail.

j--;

arr[j] += arr[j+1] ;

ans++;

}

// Else we merge left two elements

else

{

i++;

arr[i] += arr[i-1];

ans++;

}

}

return ans;

}

// Driver program to test above

int main()

{

int arr[] = {1, 4, 5, 9, 1};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Count of minimum operations is "

<< findMinOps(arr, n) << endl;

return 0;

}

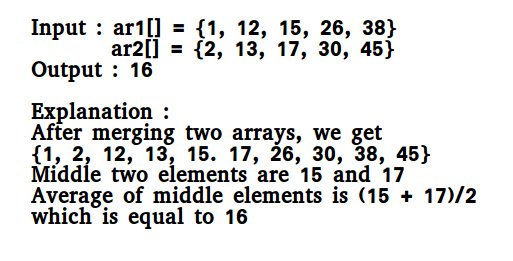
**Output :**

Count of minimum operations is 1

The time complexity for the given program is: O(n)

### 35. Median of 2 sorted arrays of equal size

There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n). The complexity should be O(log(n)).



## Solution:

Note: Since the size of the set for which we are looking for the median is even (2n), we need to take the average of the middle two numbers and return the floor of the average.

**Method 1 (Simply count while Merging)**   
Use the merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array.

See the below implementation.

// A Simple Merge based O(n)

// solution to find median of

// two sorted arrays

#include <bits/stdc++.h>

using namespace std;

/\* This function returns

median of ar1[] and ar2[].

Assumptions in this function:

Both ar1[] and ar2[]

are sorted arrays

Both have n elements \*/

int getMedian(int ar1[],

int ar2[], int n)

{

int i = 0; /\* Current index of

i/p array ar1[] \*/

int j = 0; /\* Current index of

i/p array ar2[] \*/

int count;

int m1 = -1, m2 = -1;

/\* Since there are 2n elements,

median will be average of elements

at index n-1 and n in the array

obtained after merging ar1 and ar2 \*/

for (count = 0; count <= n; count++)

{

/\* Below is to handle case where

all elements of ar1[] are

smaller than smallest(or first)

element of ar2[]\*/

if (i == n)

{

m1 = m2;

m2 = ar2[0];

break;

}

/\*Below is to handle case where

all elements of ar2[] are

smaller than smallest(or first)

element of ar1[]\*/

else if (j == n)

{

m1 = m2;

m2 = ar1[0];

break;

}

/\* equals sign because if two

arrays have some common elements \*/

if (ar1[i] <= ar2[j])

{

/\* Store the prev median \*/

m1 = m2;

m2 = ar1[i];

i++;

}

else

{

/\* Store the prev median \*/

m1 = m2;

m2 = ar2[j];

j++;

}

}

return (m1 + m2)/2;

}

// Driver Code

int main()

{

int ar1[] = {1, 12, 15, 26, 38};

int ar2[] = {2, 13, 17, 30, 45};

int n1 = sizeof(ar1) / sizeof(ar1[0]);

int n2 = sizeof(ar2) / sizeof(ar2[0]);

if (n1 == n2)

cout << "Median is "

<< getMedian(ar1, ar2, n1) ;

else

cout << "Doesn't work for arrays"

<< " of unequal size" ;

getchar();

return 0;

}

**Output :**

Median is 16

**Time Complexity :** O(n)

**Auxiliary Space:** O(1)

Method 2 (By comparing the medians of two arrays)   
This method works by first getting medians of the two sorted arrays and then comparing them.  
Let ar1 and ar2 be the input arrays.   
Algorithm :

1) Calculate the medians m1 and m2 of the input arrays ar1[]

and ar2[] respectively.

2) If m1 and m2 both are equal then we are done.

return m1 (or m2)

3) If m1 is greater than m2, then median is present in one

of the below two subarrays.

a) From first element of ar1 to m1 (ar1[0...|\_n/2\_|])

b) From m2 to last element of ar2 (ar2[|\_n/2\_|...n-1])

4) If m2 is greater than m1, then median is present in one

of the below two subarrays.

a) From m1 to last element of ar1 (ar1[|\_n/2\_|...n-1])

b) From first element of ar2 to m2 (ar2[0...|\_n/2\_|])

5) Repeat the above process until size of both the subarrays

becomes 2.

6) If size of the two arrays is 2 then use below formula to get

the median.

Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

Examples :

ar1[] = {1, 12, 15, 26, 38}

ar2[] = {2, 13, 17, 30, 45}

For above two arrays m1 = 15 and m2 = 17  
For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

[15, 26, 38] and [2, 13, 17]

Let us repeat the process for above two subarrays:

m1 = 26 m2 = 13.

m1 is greater than m2. So the subarrays become

[15, 26] and [13, 17]

Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

= (max(15, 13) + min(26, 17))/2

= (15 + 17)/2

= 16

Implementation :

// A divide and conquer based

// efficient solution to find

// median of two sorted arrays

// of same size.

#include<bits/stdc++.h>

using namespace std;

/\* to get median of a

sorted array \*/

int median(int [], int);

/\* This function returns median

of ar1[] and ar2[].

Assumptions in this function:

Both ar1[] and ar2[] are

sorted arrays

Both have n elements \*/

int getMedian(int ar1[],

int ar2[], int n)

{

/\* return -1 for

invalid input \*/

if (n <= 0)

return -1;

if (n == 1)

return (ar1[0] +

ar2[0]) / 2;

if (n == 2)

return (max(ar1[0], ar2[0]) +

min(ar1[1], ar2[1])) / 2;

/\* get the median of

the first array \*/

int m1 = median(ar1, n);

/\* get the median of

the second array \*/

int m2 = median(ar2, n);

/\* If medians are equal then

return either m1 or m2 \*/

if (m1 == m2)

return m1;

/\* if m1 < m2 then median must

exist in ar1[m1....] and

ar2[....m2] \*/

if (m1 < m2)

{

if (n % 2 == 0)

return getMedian(ar1 + n / 2 - 1,

ar2, n - n / 2 + 1);

return getMedian(ar1 + n / 2,

ar2, n - n / 2);

}

/\* if m1 > m2 then median must

exist in ar1[....m1] and

ar2[m2...] \*/

if (n % 2 == 0)

return getMedian(ar2 + n / 2 - 1,

ar1, n - n / 2 + 1);

return getMedian(ar2 + n / 2,

ar1, n - n / 2);

}

/\* Function to get median

of a sorted array \*/

int median(int arr[], int n)

{

if (n % 2 == 0)

return (arr[n / 2] +

arr[n / 2 - 1]) / 2;

else

return arr[n / 2];

}

// Driver code

int main()

{

int ar1[] = {1, 2, 3, 6};

int ar2[] = {4, 6, 8, 10};

int n1 = sizeof(ar1) /

sizeof(ar1[0]);

int n2 = sizeof(ar2) /

sizeof(ar2[0]);

if (n1 == n2)

cout << "Median is "

<< getMedian(ar1, ar2, n1);

else

cout << "Doesn't work for arrays "

<< "of unequal size";

return 0;

}

**Output :**

Median is 5

**Time Complexity :** O(logn)

**Space Complexity :** O(logn), used in stack space.  
Algorithmic Paradigm: Divide and Conquer

**Method 3 (By taking union w/o extra space)**

This method works by taking the union of two arrays without extra space and then sorting them.

**Algorithm :**

1) Take the union of the input arrays ar1[] and ar2[].

2) Sort ar1[] and ar2[] respectively.

3) The median will be the last element of ar1[] + the first

element of ar2[] divided by 2. [(ar1[n-1] + ar2[0])/2].

**Implementation :**

// CPP program for the above approach

#include <bits/stdc++.h>

using namespace std;

/\* This function returns

median of ar1[] and ar2[].

Assumptions in this function:

Both ar1[] and ar2[]

are sorted arrays

Both have n elements \*/

int getMedian(int ar1[], int ar2[], int n)

{

int j = 0;

int i = n - 1;

while (ar1[i] > ar2[j] && j < n && i > -1)

swap(ar1[i--], ar2[j++]);

sort(ar1, ar1 + n);

sort(ar2, ar2 + n);

return (ar1[n - 1] + ar2[0]) / 2;

}

// Driver Code

int main()

{

int ar1[] = { 1, 12, 15, 26, 38 };

int ar2[] = { 2, 13, 17, 30, 45 };

int n1 = sizeof(ar1) / sizeof(ar1[0]);

int n2 = sizeof(ar2) / sizeof(ar2[0]);

if (n1 == n2)

cout << "Median is " << getMedian(ar1, ar2, n1);

else

cout << "Doesn't work for arrays"

<< " of unequal size";

getchar();

return 0;

}

Output

Median is 16

Time Complexity: O(nlogn)

### 36. Median of 2 sorted arrays of different size

Given two sorted arrays, a[] and b[], the task is to find the median of these sorted arrays, in O(log n + log m) time complexity, when n is the number of elements in the first array, and m is the number of elements in the second array.  
This is an extension of [median of two sorted arrays of equal size](https://www.geeksforgeeks.org/median-of-two-sorted-arrays/) problem. Here we handle arrays of unequal size also.

**Example:**

**Input:** ar1[] = {-5, 3, 6, 12, 15}

ar2[] = {-12, -10, -6, -3, 4, 10}

**Output :** The median is 3.

**Explanation :** The merged array is :

ar3[] = {-12, -10, -6, -5 , -3,

3, 4, 6, 10, 12, 15},

So the median of the merged array is 3

**Input:** ar1[] = {2, 3, 5, 8}

ar2[] = {10, 12, 14, 16, 18, 20}

**Output :** The median is 11.

**Explanation :** The merged array is :

ar3[] = {2, 3, 5, 8, 10, 12, 14, 16, 18, 20}

if the number of the elements are even,

so there are two middle elements,

take the average between the two :

(10 + 12) / 2 = 11.

## Solution:

**Method 1:** This method uses a linear and simpler approach.

**Approach:** The given arrays are sorted, so [merge the sorted arrays in an efficient way](https://www.geeksforgeeks.org/merge-two-sorted-arrays/) and keep the count of elements inserted in the output array or printed form. So when the elements in the output array are half the original size of the given array print the element as a median element. There are two cases:

1. **Case 1:** m+n is odd, the median is at (m+n)/2 th index in the array obtained after merging both the arrays.
2. **Case 2:** m+n is even, the median will be average of elements at index ((m+n)/2 – 1) and (m+n)/2 in the array obtained after merging both the arrays

**Algorithm:**

1. Given two arrays are sorted. So they can be merged in O(m+n) time. Create a variable count to have a count of elements in the output array.
2. If the value of (m+n) is odd then there is only one median else the median is the average of elements at index (m+n)/2 and ((m+n)/2 – 1).
3. To merge the both arrays, keep two indices i and j initially assigned to 0. Compare the ith index of 1st array and jth index of second, increase the index of the smallest element and increase the count.
4. Check if the count reached (m+n) / 2 if (m+n) is odd and store the element, if even store the average of (m+n)/2 th and (m+n)/2 -1 th element and print it.

****Implementation:****

// A Simple Merge based O(n) solution to find

// median of two sorted arrays

#include <bits/stdc++.h>

using namespace std;

/\* This function returns median of ar1[] and ar2[].

Assumption in this function:

Both ar1[] and ar2[] are sorted arrays \*/

int getMedian(int ar1[], int ar2[], int n, int m)

{

int i = 0; /\* Current index of input array ar1[] \*/

int j = 0; /\* Current index of input array ar2[] \*/

int count;

int m1 = -1, m2 = -1;

// Since there are (n+m) elements,

// There are following two cases

// if n+m is odd then the middle

//index is median i.e. (m+n)/2

if((m + n) % 2 == 1)

{

for (count = 0; count <= (n + m)/2; count++)

{

if(i != n && j != m)

{

m1 = (ar1[i] > ar2[j]) ? ar2[j++] : ar1[i++];

}

else if(i < n)

{

m1 = ar1[i++];

}

// for case when j<m,

else

{

m1 = ar2[j++];

}

}

return m1;

}

// median will be average of elements

// at index ((m+n)/2 - 1) and (m+n)/2

// in the array obtained after merging ar1 and ar2

else

{

for (count = 0; count <= (n + m)/2; count++)

{

m2 = m1;

if(i != n && j != m)

{

m1 = (ar1[i] > ar2[j]) ? ar2[j++] : ar1[i++];

}

else if(i < n)

{

m1 = ar1[i++];

}

// for case when j<m,

else

{

m1 = ar2[j++];

}

}

return (m1 + m2)/2;

}

}

/\* Driver code \*/

int main()

{

int ar1[] = {900};

int ar2[] = {5, 8, 10, 20};

int n1 = sizeof(ar1)/sizeof(ar1[0]);

int n2 = sizeof(ar2)/sizeof(ar2[0]);

cout << getMedian(ar1, ar2, n1, n2);

return 0;

}

Output

10

Complexity Analysis:

* Time Complexity: O(m + n).   
  To merge both the arrays O(m+n) time is needed.
* Space Complexity: O(1).   
  No extra space is required.

***Solution 2 :*Simple Mathematical Approach**

**Approach**: The given two arrays are sorted, so we need to merge them into a third array using the method System.arraycopy(src, srcPos, dest, destPos, length) and then sort the third array using Arrays.sort(array) method.

1. Case 1: If the length of the third array is odd, then the median is at (length)/2 th index in the array obtained after merging both the arrays.

2. Case 2: If the length of the third array is even, then the median will be the average of elements at index ((length)/2 ) and ((length)/2 – 1) in the array obtained after merging both the arrays.

#### ****Algorithm:****

1. Merge the two given arrays into one array.

2. Then sort the third(merged) array

3. If the length of the third array is even then :

divide the length of array by 2

return arr[value] + arr[value - 1] / 2

4. If the length of the third array is odd then :

divide the length of array by 2

round that value

return the arr[value]

#### ****Implementation:****

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

int Solution(int arr[], int n)

{

// If length of array is even

if (n % 2 == 0)

{

int z = n / 2;

int e = arr[z];

int q = arr[z - 1];

int ans = (e + q) / 2;

return ans;

}

// If length if array is odd

else

{

int z = round(n / 2);

return arr[z];

}

}

// Driver Code

int main() {

// TODO Auto-generated method stub

int arr1[] = { -5, 3, 6, 12, 15 };

int arr2[] = { -12, -10, -6, -3, 4, 10 };

int i = sizeof(arr1) / sizeof(arr1[0]);

int j = sizeof(arr2) / sizeof(arr2[0]);

int arr3[i+j];

int l = i+j;

// Merge two array into one array

for(int k=0;k<i;k++)

{

arr3[k]=arr1[k];

}

int a=0;

for(int k=i;k<l;k++)

{

arr3[k]=arr2[a++];

}

// Sort the merged array

sort(arr3,arr3+l);

// calling the method

cout<<"Median = " << Solution(arr3, l);

}

Output

Median = 3

Complexity Analysis :

* Time Complexity: O(n Log n)
* Space Complexity: O(i+j). Since we are creating a new array of size i+j.

*****Solution 3***** : ****Binary Search****

Approach: The given two arrays are sorted, so we can utilize the ability of Binary Search to divide the array and find the median.

Median means the point at which the whole array is divide into two parts. Hence since the two arrays are not merged so to get the median we require merging which is costly. Hence instead of merging we will use below given algorithm to efficiently find median.

Algorithm:

1. Lets assume that there are two arrays A and B with array A having the minimum number of elements.

If this is not the case than swap A and B to make A having small size.

2. The edge cases like one array is empty or both are empty will be handled.

3. let n be the size of A and m be the size of B.

Now think of an idea that if we have to find the median than we have to divide the whole merged array into two parts

namely left and right parts.

Now since we are given the size of left part (i.e (n+m+1)/2), Now look at below given example.

A-> 1,2,3,4,5 n = 5

B-> 1,2,3,4,5,6 m = 6

Here merged array will look like :- 1,1,2,2,3,3,4,4,5,5,6 and median then is 3

Now we can see our left part which is underlined. We divide A and B into two parts such that the

sum of left part of both A and B will result in left part of merged array.

A-> 1,2,3,4,5 // pointers l =0 and r = n-1 hence mid = (l+r)/2;

B -> 1,2,3,4,5,6

we can see that left part of A is given as n/2 and since total length of left part in merged array

is (m+n+1)/2, so left part of B = (m+n+1)/2-n/2;

Now we just have to confirm if our left and right partitions in A and B are correct or not.

4. Now we have 4 variables indicating four values two from array A and two from array B.

leftA -> Rightmost element in left part of A = 2

leftb -> Rightmost element in left part of B = 4

rightA -> Leftmost element in right part of A = 3

rightB -> Leftmost element in right part of B = 5

Hence to confirm that partition is correct we have to check the following conditions.

leftA<=rightB and leftB<=rightA // This is the case when the sum of two parts of A and B results in left part of merged array

if our partition not works that means we have to find other mid point in A and then left part in B

This is seen when

leftA > rightB //means we have to dec size of A's partition

so do r = mid-1;

else

do l =mid+1;

Hence repeat the above steps with new partitions till we get the answers.

5. If leftA<=rightB and leftB<=rightA

then we get correct partition and our answer depends on the total size of merged array (i.e. m+n)

If (m+n)%2==0

ans is max(leftA,leftB)+min(rightA,rightB)/2; // max of left part is nearest to median and min of right part is nearest to medain

else

ans is max(leftA,leftB);

Hence the above algorithm can be coded as:

#include <bits/stdc++.h>

using namespace std;

// Method to find median

double Median(vector<int>& A, vector<int>& B)

{

int n = A.size();

int m = B.size();

if (n > m)

return Median(B, A); // Swapping to make A smaller

int start = 0;

int end = n;

int realmidinmergedarray = (n + m + 1) / 2;

while (start <= end) {

int mid = (start + end) / 2;

int leftAsize = mid;

int leftBsize = realmidinmergedarray - mid;

int leftA

= (leftAsize > 0)

? A[leftAsize - 1]

: INT\_MIN; // checking overflow of indices

int leftB

= (leftBsize > 0) ? B[leftBsize - 1] : INT\_MIN;

int rightA

= (leftAsize < n) ? A[leftAsize] : INT\_MAX;

int rightB

= (leftBsize < m) ? B[leftBsize] : INT\_MAX;

// if correct partition is done

if (leftA <= rightB and leftB <= rightA) {

if ((m + n) % 2 == 0)

return (max(leftA, leftB)

+ min(rightA, rightB))

/ 2.0;

return max(leftA, leftB);

}

else if (leftA > rightB) {

end = mid - 1;

}

else

start = mid + 1;

}

return 0.0;

}

// Driver code

int main()

{

vector<int> arr1 = { -5, 3, 6, 12, 15 };

vector<int> arr2 = { -12, -10, -6, -3, 4, 10 };

cout << "Median of the two arrays are" << endl;

cout << Median(arr1, arr2);

return 0;

}

Output

Median of the two arrays are

3

Time Complexity: O(min(log m, log n)) : Since binary search is being applied on the smaller of the 2 arrays  
Auxiliary Space: O(1)