**Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted**

1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.

2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

**Solution:**  
**1) Find the candidate unsorted subarray**   
a) Scan from left to right and find the first element which is greater than the next element. Let *s* be the index of such an element. In the above example 1, *s* is 3 (index of 30).  
b) Scan from right to left and find the first element (first in right to left order) which is smaller than the next element (next in right to left order). Let *e* be the index of such an element. In the above example 1, e is 7 (index of 31).

**2) Check whether sorting the candidate unsorted subarray makes the complete array sorted or not. If not, then include more elements in the subarray.**  
a) Find the minimum and maximum values in *arr[s..e]*. Let minimum and maximum values be *min* and *max*. *min* and *max* for [30, 25, 40, 32, 31] are 25 and 40 respectively.  
b) Find the first element (if there is any) in *arr[0..s-1]* which is greater than *min*, change *s* to index of this element. There is no such element in above example 1.  
c) Find the last element (if there is any) in *arr[e+1..n-1]* which is smaller than max, change *e* to index of this element. In the above example 1, e is changed to 8 (index of 35)

**3) Print *s* and *e*.**

**Find the maximum repeating number in O(n) time and O(1) extra space**

Given an array of size *n*, the array contains numbers in range from 0 to *k-1* where *k* is a positive integer and *k <= n.* Find the maximum repeating number in this array. For example, let *k* be 10 the given array be *arr[]* = {1, 2, 2, 2, 0, 2, 0, 2, 3, 8, 0, 9, 2, 3}, the maximum repeating number would be 2. Expected **time complexity is *O(n)***and **extra space allowed is *O(1)***. Modifications to array are allowed.

* All the elements are in the range 0 to k-1.
* Scan the array once and increment the element by k considering the element as index.
* Find the max. element from the array and print its index. It will be the maximum repeating element of the array.
* Get back the array by taking all the elements %k because we have added k to the element so %k will give us the element back.

If you still didnt get the logic then more explanation and example, see below :

Let us understand the approach with a simple example where *arr[]* = {2, 3, 3, 5, 3, 4, 1, 7}, *k* = 8, *n* = 8 (number of elements in arr[]).

**1)** Iterate though input array *arr[]*, for every element *arr[i]*, increment *arr[arr[i]%k]* by *k* (*arr[]* becomes {2, 11, 11, 29, 11, 12, 1, 15 })

**2)** Find the maximum value in the modified array (maximum value is 29). Index of the maximum value is the maximum repeating element (index of 29 is 3).

**3)** If we want to get the original array back, we can iterate through the array one more time and do *arr[i] = arr[i] % k* where *i* varies from 0 to *n-1*.

**Merge Overlapping Intervals**

Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.   
For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8} }. The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}. Similarly {5, 7} and {6, 8} should be merged and become {5, 8}

Logic:

* Sort the array by starting time.
* Now your work will be done in single linear scan.
* Take first element. Check its finish time with the starting time of the next element in the array.
* If the finish time is greater than the interval is overlapping so we can merge.
* Now check the finish time of second one with starting time of third one.
* When it does not overlap, print the starting time of first interval and finish time of last one.
* Repeat again from the interval when you didnt find the overlapping for the whole array.

TC : O(nlogn) (because of sorting)

SC : O(1) (no space in linear scan.. no considering the space for sorting)

**Rearrange positive and negative numbers in O(n) time and O(1) extra space**

An array contains both positive and negative numbers in random order. Rearrange the array elements so that positive and negative numbers are placed alternatively.

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.

For example, if the input array is [-1, 2, -3, 4, 5, 6, -7, 8, 9], then the output should be [9, -7, 8, -3, 5, -1, 2, 4, 6]

**Soln:**

* separate positive and negative numbers using partition process.
* So all negative numbers are placed before positive numbers.
* we start from the first negative number and first positive number, and swap every alternate negative number with next positive number.
* i.e. Increment index for -ve by 2 and index for +ve by 1 and
* keep swaping the elements if they are of oppposite sign.

**Pancake sorting**

Given an an unsorted array, sort the given array. You are allowed to do only following operation on array.

flip(arr, i): Reverse array from 0 to i

The goal is to sort the sequence in as few reversals as possible.

The idea is to do something similar to [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort).

We one by one place maximum element at the end and reduce the size of current array by one.

Following are the detailed steps. Let given array be arr[] and size of array be n.  
1) Start from current size equal to n and reduce current size by one while it’s greater than 1. Let the current size be curr\_size. Do following for every curr\_size  
……a) Find index of the maximum element in arr[0..curr\_szie-1]. Let the index be ‘mi’  
……b) Call flip(arr, mi)  
……c) Call flip(arr, curr\_size-1)

VIDEO IS THERE IN THE array FOLDER... PLEASE WATCH IT CAREFULLY.. YOU WILL GET WHOLE IDEA IN JUST A MINUTE.

**Counting Sort**

[Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in

the output sequence.

3) Output each object from the input sequence followed by

decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

next data 1 at an index 1 smaller than this index.

e.g.

Now we process input array.

1

Initially

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

Here the count is of 1 is 2. Put 1 at index=(count-1)=(2-1)=1 in output

and decrement count of 1 in count array.

So now

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 1 4 4 5 6 6 7 7 7

Repeat this for all the elements of input array..

Output array will have the sorted sequence.

**NOTES:**

THIS ALGORITHM REDUCES THE TIME COMPLEXITY IF THE RANGE OF THE ELEMENTS IS FIXED

FOR EXAMPLE.. SORTING THE STRING. ONLY 255 IS THE RANGE..

**Time Complexity:** O(n+k) where n is the number of elements in input array and k is the range of input.  
**Auxiliary Space:** O(n+k)

**Points to be noted:**  
**1.** Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.  
**2.** It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.  
**3.** It is often used as a sub-routine to another sorting algorithm like radix sort.  
**4.** Counting sort uses a partial hashing to count the occurrence of the data object in O(1).  
**5.** Counting sort can be extended to work for negative inputs also.

**Exercise:**  
**1.** Modify above code to sort the input data in the range from M to N.  
**2.** Modify above code to sort negative input data.  
**3.** Is counting sort stable and online?  
**4.** Thoughts on parallelizing the counting sort algorithm.

**Block swap algorithm for array rotation**

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

Rotation of the above array by 2 will make array

SOLUTION :

* Reverse arr[0..d-1].
* Reverse arr[d..n-1].
* Reverse arr[0..n-1].

TC : O(n)

SC : O(1)

**Sort a nearly sorted (or K sorted) array**

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

**Solution** :

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.  
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time (See [this GFact](http://www.geeksforgeeks.org/archives/12580))  
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)\*logK)

Another solution but adds some cost to it.

We can also **use a Balanced Binary Search Tree** instead of Heap to store K+1 elements. The [insert](http://www.geeksforgeeks.org/archives/17679) and [delete](http://www.geeksforgeeks.org/archives/18009) operations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn’t need extra space for left and right pointers.

Count smaller elements on right side

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

Examples:

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

Output: countSmaller[] = {6, 1, 1, 1, 0, 1, 0}

(Corner Cases)

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

Output: countSmaller[] = {4, 3, 2, 1, 0}

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

Output: countSmaller[] = {0, 0, 0, 0, 0}

**SOLUTION**:

A Self Balancing Binary Search Tree (AVL, Red Black,.. etc) can be used to get the solution in O(nLogn) time complexity. We can augment these trees so that every node N contains size the subtree rooted with N. We have used AVL tree in the following implementation.

We traverse the array from right to left and insert all elements one by one in an AVL tree.

While inserting a new key in an AVL tree, we first compare the key with root.

If key is greater than root, then it is greater than all the nodes in left subtree of root. So we add the size of left subtree to the count of smaller element for the key being inserted.

We recursively follow the same approach for all nodes down the root.

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

Linked List vs Array

**Difficulty Level:** Rookie

Both Arrays and [Linked List](http://en.wikipedia.org/wiki/Linked_list) can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, .....].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays  
1) Dynamic size  
2) Ease of insertion/deletion

Linked lists have following drawbacks:  
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.  
2) Extra memory space for a pointer is required with each element of the list.  
3) Arrays have better cache locality that can make a pretty big difference in performance.

Please also see [this](http://geeksforgeeks.org/forum/topic/tcs-interview-question-for-software-engineerdeveloper-fresher-about-linked-lists#post-18153) thread.

References:  
<http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>

**Sort elements by frequency**

Print the elements of an array in the decreasing frequency if 2 numbers have same frequency then print the one which came 1st  
E.g. 2 5 2 8 5 6 8 8 output: 8 8 8 2 2 5 5 6.

**SOLUTION:**

**1)** Create a BST and while creating BST maintain the count i,e frequency of each coming element in same BST. This step may take O(nLogn) time if a self balancing BST is used.  
**2)** Do Inorder traversal of BST and store every element and count of each element in an auxiliary array. Let us call the auxiliary array as ‘count[]‘. Note that every element of this array is element and frequency pair. This step takes O(n) time.  
**3)** Sort ‘count[]‘ according to frequency of the elements. This step takes O(nLohn) time if a O(nLogn) sorting algorithm is used.  
**4)** Traverse through the sorted array ‘count[]‘. For each element x, print it ‘freq’ times where ‘freq’ is frequency of x. This step takes O(n) time.

Overall time complexity of the algorithm can be minimum O(nLogn) if we use a O(nLogn) sorting algorithm and use a self balancing BST with O(Logn) insert operation.

**Exercise:**  
The above implementation doesn’t guarantee original order of elements with same frequency (for example, 4 comes before 5 in input, but 4 comes after 5 in output). Extend the implementation to maintain original order. For example, if two elements have same frequency then print the one which came 1st in input array.

**Median in a stream of integers (running integers)**

Given that integers are read from a data stream. Find median of elements read so for in efficient way. For simplicity assume there are no duplicates. For example, let us consider the stream 5, 15, 1, 3 …

After reading 1st element of stream - 5 -> median - 5

After reading 2nd element of stream - 5, 15 -> median - 10

After reading 3rd element of stream - 5, 15, 1 -> median - 5

After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so on...

Making it clear, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick average of middle two elements in sorted stream.

Note that output is *effective median* of integers read from the stream so far. Such an algorithm is called online algorithm. Any algorithm that can guarantee output of *i*-elements after processing *i*-th element, is said to be ***online algorithm***.

**SOLUTION:**

At every node of BST, maintain number of elements in the subtree rooted at that node. We can use a node as root of simple binary tree, whose left child is self balancing BST with elements less than root and right child is self balancing BST with elements greater than root. The root element always holds *effective median*.

If left and right subtrees contain same number of elements, root node holds average of left and right subtree root data. Otherwise, root contains same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.

Self balancing BST is costly in managing balancing factor of BST. However, they provide sorted data which we don’t need. We need median only. The next method make use of Heaps to trace median.

Use of Heaps

Similar to balancing BST above, we can use a max heap on left side to represent elements that are less than *effective median*, and a min heap on right side to represent elements that are greater than *effective median*.

After processing an incoming element, the number of elements in heaps differ utmost by 1 element. When both heaps contain same number of elements, we pick average of heaps root data as *effective median*. When the heaps are not balanced, we select *effective median* from the root of heap containing more elements.

Each time it takes O(logn) time to find median because updation of atleast 1 heap tree and atmost 2 heap trees.

**Time Complexity:** If we omit the way how stream was read, complexity of median finding is ***O(N log N)***, as we need to read the stream, and due to heap insertions/deletions.

k largest(or smallest) elements in an array | added Min Heap method

**Question:** Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.  
……a) If the element is greater than the root then make it root and call [heapify](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSort.htm) for MH  
……b) Else ignore it.  
// The step 2 is O((n-k)\*logk)

3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

**Find the point where a monotonically increasing function becomes positive first time**

Given a function ‘int f(unsigned int x)’ which takes a **non-negative integer** ‘x’ as input and returns an **integer** as output. The function is monotonically increasing with respect to value of x, i.e., the value of f(x+1) is greater than f(x) for every input x. Find the value ‘n’ where f() becomes positive for the first time. Since f() is monotonically increasing, values of f(n+1), f(n+2),… must be positive and values of f(n-2), f(n-3), .. must be negative.  
Find n in O(logn) time, you may assume that f(x) can be evaluated in O(1) time for any input x.

A **simple solution** is to start from i equals to 0 and one by one calculate value of f(i) for 1, 2, 3, 4 .. etc until we find a positive f(i). This works, but takes O(n) time.

**Can we apply Binary Search to find n in O(Logn) time?** We can’t directly apply Binary Search as we don’t have an upper limit or high index. The idea is to do repeated doubling until we find a positive value, i.e., check values of f() for following values until f(i) becomes positive.

f(0)

f(1)

f(2)

f(4)

f(8)

f(16)

f(32)

....

....

f(high)

Let 'high' be the value of i when f() becomes positive for first time.

Can we apply Binary Search to find n after finding ‘high’? We can apply Binary Search now, we can use ‘high/2′ as low and ‘high’ as high indexes in binary search. The result n must lie between ‘high/2′ and ‘high’.

Number of steps for finding ‘high’ is O(Logn). So we can find ‘high’ in O(Logn) time. What about time taken by Binary Search between high/2 and high? The value of ‘high’ must be less than 2\*n. The number of elements between high/2 and high must be O(n). Therefore, time complexity of Binary Search is O(Logn) and overall time complexity is 2\*O(Logn) which is O(Logn).

**Arrange given numbers to form the biggest number**

Given an array of numbers, arrange them in a way that yields the largest value. For example, if the given numbers are {54, 546, 548, 60}, the arrangement 6054854654 gives the largest value. And if the given numbers are {1, 34, 3, 98, 9, 76, 45, 4}, then the arrangement 998764543431 gives the largest value.

**Solution :**

Sort the array but only twist is explained below..

Use any comparison based sorting algorithm.

In the used sorting algorithm, instead of using the default comparison, write a comparison function myCompare() and use it to sort numbers.

Given two numbers X and Y, how should myCompare() decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y).

If XY is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

**Merge two sorted arrays with O(1) extra space**

We are given two sorted array. We need to merge these two arrays such that the initial numbers (after complete sorting) are in the first array and the remaining numbers are in the second array. Extra space allowed in O(1).

Example:

Input: ar1[] = {1, 5, 9, 10, 15, 20};

ar2[] = {2, 3, 8, 13};

Output: ar1[] = {1, 2, 3, 5, 8, 9}

ar2[] = {10, 13, 15, 20}

* The idea is to begin from last element of ar2[] and search it in ar1[].
* If there is a greater element in ar1[], then we move last element of ar1[] to ar2[].
* To keep ar1[] and ar2[] sorted, we need to place last element of ar2[] at correct place in ar1[].
* We can use [Insertion Sort](http://geeksquiz.com/insertion-sort/) type of insertion for this.
* TC : O(m\*n)

# Find if there is a subarray with 0 sum

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

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

Output: true

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

Output: false

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

int sum = 0;

// Traverse through the given array

for (int i = 0; i < arr.length; i++)

{

// Add current element to sum

sum += arr[i];

// Return true in following cases

// a) Current element is 0

// b) sum of elements from 0 to i is 0

// c) sum is already present in hash map

if (arr[i] == 0 || sum == 0 || hM.get(sum) != null)

return true;

// Add sum to hash map

hM.put(sum, i);

}

// We reach here only when there is no subarray with 0 sum

return false;

# Move all zeroes to end of array

Given an array of random numbers, Push all the zero’s of a given array to the end of the array. For example, if the given arrays is {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0}, it should be changed to {1, 9, 8, 4, 2, 7, 6, 0, 0, 0, 0}. The order of all other elements should be same. Expected time complexity is O(n) and extra space is O(1).

Traverse the given array ‘arr’ from left to right.

Maintain two indexes.

One to maintain next non-zero place (next)

second in normal loop. (0 to n-1) (i)

If arr[i] is non-zero then arr[next]=arr[i] and increment next.

After completing whole loop. All non zero elements would be in sequence.

Now set 0 to the rest of the elements.

# Print All Distinct Elements of a given integer array

Input: arr[] = {12, 10, 9, 45, 2, 10, 10, 45}

Output: 12, 10, 9, 45, 2

Sort the array so that all the dup elements would be in sequence.

Now scan the array and print the element first time and skip the duplicates.

TC : O(nlogn + n)

# Count 1’s in a sorted binary array (decreasing sorted)

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

Output: 2

Use binary search and find the last index of 1.

# Find the first repeating element in an array of integers

Given an array of integers, find the first repeating element in it. We need to find the element that occurs more than once and whose index of first occurrence is smallest.

Examples:

Input: arr[] = {10, 5, 3, 4, 3, 5, 6}

Output: 5 [5 is the first element that repeats]

* Make the array of structure which contains the element and index both.
* Sort the array by values.
* Now scan that array and if the element is repeating then record the element and its index(for checking minimum).

# Find the closest pair from two sorted arrays

Given two sorted arrays and a number x, find the pair whose sum is closest to x and **the pair has an element from each array**.

We are given two arrays ar1[0…m-1] and ar2[0..n-1] and a number x, we need to find the pair ar1[i] + ar2[j] such that absolute value of (ar1[i] + ar2[j] – x) is minimum.

Example:

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 32

Output: 1 and 30

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 50

Output: 7 and 40

Use the same logic as finding pair with sum x.

Start and end logic.

1) Initialize a variable diff as infinite (Diff is used to store the

difference between pair and x). We need to find the minimum diff.

2) Initialize two index variables l and r in the given sorted array.

(a) Initialize first to the leftmost index in ar1: l = 0

(b) Initialize second the rightmost index in ar2: r = n-1

3) Loop while l < m and r >= 0

(a) If abs(ar1[l] + ar2[r] - sum) < diff then

update diff and result

(b) Else if(ar1[l] + ar2[r] < sum ) then l++

(c) Else r--

4) Print the result.

TC : O(m+n)

SC : O(1)

# Count pairs with given sum

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

sum = 6

Output : 2

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.

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;

}

TC : O(n)

# Find minimum difference between any two elements

Given an unsorted array, find the minimum difference between any pair in given array.

Input : {1, 5, 3, 19, 18, 25};

Output : 1

Sort the array and find the min diff by comparing neighbor elements.

TC : O(nlogn)

# Sort an array in wave form

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

The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don’t need to worry about odd positioned element. Following are simple steps.  
1) Traverse all even positioned elements of input array, and do following.  
….a) If current element is smaller than previous odd element, swap previous and current.  
….b) If current element is smaller than next odd element, swap next and current.

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=

// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....

void sortInWave(int arr[], int n)

{

    // Traverse all even elements

    for (int i = 0; i < n; i+=2)

    {

        // If current even element is smaller than previous

        if (i>0 && arr[i-1] > arr[i] )

            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next

        if (i<n-1 && arr[i] < arr[i+1] )

            swap(&arr[i], &arr[i + 1]);

    }

}

TC : O(n)

SC : O(1)