Contributor

Santosh Kumar Mishra

Software Engineer at Microsoft

LinkedIn: @iamsantoshmishra

Email: 93mishra@gmail.com

Arrays & Matrices

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure."

In computer science, an array data structure, or simply an array, is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.

Arrays & Matrices

Table of Contents

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

http://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/

METHOD 1 (Use Sorting)

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

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

METHOD 2 (Use Hash Map)

```
This method works in O(n) time if range of numbers is known.
Let sum be the given sum and A[] be the array in which we need to find pair.
1) Initialize Binary Hash Map M[] = {0, 0, ...}
2) Do following for each element A[i] in A[]
   (a)        If M[x - A[i]] is set then print the pair (A[i], x - A[i])
   (b)        Set M[A[i]]
```

2 Largest Sum Contiguous Subarray

http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Kadane's Algorithm:
Simple idea of the 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

```
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;
}
```

Above program can be optimized further, if we compare max_so_far with max_ending_here only if max_ending_here is greater than 0.

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = 0, max_ending_here = 0;
   for (int i = 0; i < size; i++){
       max_ending_here = max_ending_here + a[i];
       if (max_ending_here < 0)
           max_ending_here = 0;
        /* Do not compare for all elements. Compare only
          when  max_ending_here > 0 */
       else if (max_so_far < max_ending_here)
           max_so_far = max_ending_here;
   }
   return max_so_far;
}
```

Time Complexity: O(n)

Algorithmic Paradigm: Dynamic Programming

The implementation handles the case when all numbers in array are negative.

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = a[0];
   int curr_max = a[0];

   for (int i = 1; i < size; i++)
   {
        curr_max = max(a[i], curr_max+a[i]);
        max_so_far = max(max_so_far, curr_max);
   }
   return max_so_far;
}
```

To print the subarray with the maximum sum, we maintain indices whenever we get the maximum sum.

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0,
       start =0, end = 0, s=0;

    for (int i=0; i< size; i++ )
    {   max_ending_here += a[i];
         if (max_so_far < max_ending_here)
        {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }
         if (max_ending_here < 0)
        {
            max_ending_here = 0;
            s = i+1;
        }
    }
    cout << "Maximum contiguous sum is "
        << max_so_far << endl;
    cout << "Starting index "<< start
        << endl << "Ending index "<< end << endl;
}
```

Now try below question

Given an array of integers (possibly some of the elements negative), write a C program to find out the *maximum product* possible by adding 'n' consecutive integers in the array, n <= ARRAY_SIZE. Also give where in the array this sequence of n integers starts.

3 Majority Element

http://www.geeksforgeeks.org/majority-element/

Majority Element: A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).
Write a function which takes an array and emits the majority element (if it exists), otherwise prints NONE as follows:

```
I/P : 3 3 4 2 4 4 2 4 4
O/P : 4

I/P : 3 3 4 2 4 4 2 4
O/P : NONE
```

METHOD 1 (Basic)

The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than n/2 then break the loops and return the element having maximum count. If maximum count doesn't become more than n/2 then majority element doesn't exist.
Time Complexity: O(n*n).
Auxiliary Space : O(1).

METHOD 2 (Using Binary Search Tree)

```
struct tree
{
  int element;
  int count;
}BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than n/2 then return.
The method works well for the cases where n/2+1 occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 1, 2, 3, 4}.
Time Complexity: If a binary search tree is used then time complexity will be O(n^2). If a self-balancing-binary-search tree is used then O(nlogn)
Auxiliary Space: O(n)

METHOD 3 (Using Moore's Voting Algorithm)

This is a two step process.

1. Get an element occurring most of the time in the array. This phase will make sure that if there is a majority element then it will return that only.

2. Check if the element obtained from above step is majority element.

1. **Finding a Candidate:**

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

```
/* Function to find the candidate for Majority */
int findCandidate(int a[], int size)
{   int maj_index = 0, count = 1;
    int i;
    for (i = 1; i < size; i++)
    {
        if (a[maj_index] == a[i])
            count++;
        else
            count--;
        if (count == 0)
        {
            maj_index = i;
            count = 1;
        }}
    return a[maj_index];
}
 /* Function to check if the candidate occurs more than n/2 times */
bool isMajority(int a[], int size, int cand)
{
    int i, count = 0;
    for (i = 0; i < size; i++)
      if (a[i] == cand)
         count++;
    if (count > size/2)
       return 1;
    else
       return 0;
}
```

4 Find the Number Occurring Odd Number of Times

http://www.geeksforgeeks.org/find-the-number-occurring-odd-number-of-times/

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

Example:

I/P = [1, 2, 3, 2, 3, 1, 3]

O/P = 3

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is $O(n^2)$.

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

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

```
int getOddOccurrence(int ar[], int ar_size)
{
     int i;
     int res = 0;
     for (i=0; i < ar_size; i++)
        res = res ^ ar[i];

     return res;
}
```

5 Find the Missing Number

http://www.geeksforgeeks.org/find-the-missing-number/

You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Example:

I/P [1, 2, 4, ,6, 3, 7, 8]

O/P 5

METHOD 1(Use sum formula)

Algorithm:
1. Get the sum of numbers
   total = n*(n+1)/2
2 Subtract all the numbers from sum and
   you will get the missing number.

Time Complexity: O(n)
There can be overflow if n is large. In order to avoid Integer Overflow, we can pick one number from known numbers and subtract one number from given numbers. This way we wont have Integer Overflow ever. Thanks to Sahil Rally for suggesting this improvement.

METHOD 2(Use XOR)

1) XOR all the array elements, let the result of XOR be X1.
2) XOR all numbers from 1 to n, let XOR be X2.
3) XOR of X1 and X2 gives the missing number.

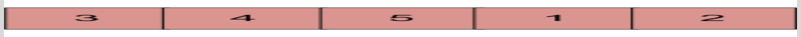
```
int getMissingNo(int a[], int n)
{
    int i;
    int x1 = a[0]; /* For xor of all the elements in array */
    int x2 = 1; /* For xor of all the elements from 1 to n+1 */

    for (i = 1; i< n; i++)
        x1 = x1^a[i];

    for ( i = 2; i <= n+1; i++)
        x2 = x2^i;

    return (x1^x2);
}
```

Time Complexity: O(n)

6 Search an element in a sorted and rotated array

http://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/

An element in a sorted array can be found in O(log n) time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.



Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
         key = 3
Output : Found at index 8

```
Input  : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
        key = 30
Output : Not found
Input : arr[] = {30, 40, 50, 10, 20}
       key = 10
Output : Found at index 3
```

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.
The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only only element for which next element to it is smaller than it.
Using above criteria and binary search methodology we can get pivot element in O(logn) time

```
int pivotedBinarySearch(int arr[], int n, int key)
{
   int pivot = findPivot(arr, 0, n-1);

   // If we didn't find a pivot, then array is not rotated at all
   if (pivot == -1)
       return binarySearch(arr, 0, n-1, key);
    // If we found a pivot, then first compare with pivot and then
   // search in two subarrays around pivot
   if (arr[pivot] == key)
       return pivot;
   if (arr[0] <= key)
       return binarySearch(arr, 0, pivot-1, key);
   return binarySearch(arr, pivot+1, n-1, key);
}
```

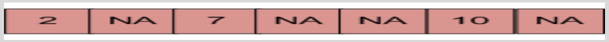
```
/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
   3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
   // base cases
   if (high < low)  return -1;
   if (high == low) return low;

   int mid = (low + high)/2;   /*low + (high - low)/2;*/
   if (mid < high && arr[mid] > arr[mid + 1])
       return mid;
   if (mid > low && arr[mid] < arr[mid - 1])
       return (mid-1);
   if (arr[low] >= arr[mid])
       return findPivot(arr, low, mid-1);
   return findPivot(arr, mid + 1, high);
}
```

7 Merge an array of size n into another array of size m+n

http://www.geeksforgeeks.org/merge-one-array-of-size-n-into-another-one-of-size-mn/

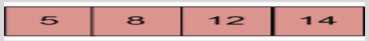
There are two sorted arrays. First one is of size m+n containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m+n such that the output is sorted.

Input: array with m+n elements (mPlusN[]).

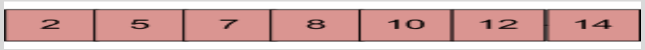


NA => Value is not filled/available in array mPlusN[]. There should be n such array blocks.

Input: array with n elements (N[]).



Output: N[] merged into mPlusN[] (Modified mPlusN[])



Algorithm:

Let first array be mPlusN[] and other array be N[]

1) Move m elements of mPlusN[] to end.

2) Start from nth element of mPlusN[] and 0th element of N[] and merge them into mPlusN[].

```
int merge(int mPlusN[], int N[], int m, int n)
{
  int i = n;  /* Current index of i/p part of mPlusN[]*/
  int j = 0; /* Current index of N[]*/
  int k = 0; /* Current index of of output mPlusN[]*/
  while (k < (m+n))
  {
    /* Take an element from mPlusN[] if
       a) value of the picked element is smaller and we have
          not reached end of it
       b) We have reached end of N[] */
    if ((i < (m+n) && mPlusN[i] <= N[j]) || (j == n))
    {
      mPlusN[k] = mPlusN[i];
      k++;
      i++;
    }
    else  // Otherwise take element from N[]
    { mPlusN[k] = N[j];
      k++;
      j++;
    }
  }
}
```

8 Median of two sorted arrays

http://www.geeksforgeeks.org/median-of-two-sorted-arrays/

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))

Method 1 (Simply count while Merging)

Use 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.

Time Complexity: O(n)

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
Example:
  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

```
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;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}
```

9 Reversal algorithm for array rotation

http://www.geeksforgeeks.org/program-for-array-rotation-continued-reversal-algorithm/
http://www.geeksforgeeks.org/block-swap-algorithm-for-array-rotation/

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.
Input: arr[] = [1, 2, 3, 4, 5, 6, 7]
d = 2
Output: arr[] = [3, 4, 5, 6, 7, 1, 2]

Rotation of the above array by 2 will make array

Method 4(The Reversal Algorithm)

Algorithm:

```
rotate(arr[], d, n)
reverse(arr[], 1, d) ;
reverse(arr[], d + 1, n);
reverse(arr[], l, n);
```

Time Complexity: O(n)

10 Maximum sum such that no two elements are adjacent

http://www.geeksforgeeks.org/maximum-sum-such-that-no-two-elements-are-adjacent/

Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7).Answer the question in most efficient way.

Examples :
Input : arr[] = {5, 5, 10, 100, 10, 5}
Output : 110
Input : arr[] = {1, 2, 3}
Output : 4
Input : arr[] = {1, 20, 3}
Output : 20

Algorithm:
Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.
Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

```
/*Function to return max sum such that no two elements
 are adjacent */
int FindMaxSum(int arr[], int n)
{
  int incl = arr[0];
  int excl = 0;
  int excl_new,i;
   for (i = 1; i < n; i++)
  {
     /* current max excluding i */
     excl_new = (incl > excl)? incl: excl;

     /* current max including i */
     incl = excl + arr[i];
     excl = excl_new;
  }

   /* return max of incl and excl */
   return ((incl > excl)? incl : excl);
}
```

Time Complexity: O(n)
Now try the same problem for array with negative numbers also.

11 Leaders in an array

http://www.geeksforgeeks.org/leaders-in-an-array/
Write a program to print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. For example int the array {16, 17, 4, 3, 5, 2}, leaders are 17, 5 and 2.
Let the input array be arr[] and size of the array be *size*.

Method 1 (Simple)
Use two loops. The outer loop runs from 0 to size – 1 and one by one picks all elements from left to right. The inner loop compares the picked element to all the elements to its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader

Time Complexity: O(n*n)

Method 2 (Scan from right)

Scan all the elements from right to left in array and keep track of maximum till now. When maximum changes it's value, print it.

```
/* C++ Function to print leaders in an array */

void printLeaders(int arr[], int size)
{
    int max_from_right =  arr[size-1];

    /* Rightmost element is always leader */
    cout << max_from_right << " ";

    for (int i = size-2; i >= 0; i--)
    {
        if (max_from_right < arr[i])
        {
            max_from_right = arr[i];
            cout << max_from_right << " ";
        }
    }
}
```

Time Complexity: O(n)

12 Sort elements by frequency

http://www.geeksforgeeks.org/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 first.

Examples:
Input: arr[] = {2, 5, 2, 8, 5, 6, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 5, 6}

Input: arr[] = {2, 5, 2, 6, -1, 9999999, 5, 8, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 5, 6, -1, 9999999}

METHOD 1 (Use Sorting)

1) Use a sorting algorithm to sort the elements O(nlogn)
2) Scan the sorted array and construct a 2D array of element and count O(n).
3) Sort the 2D array according to count O(nlogn).

Example:

```
Input 2 5 2 8 5 6 8 8

After sorting we get
2 2 5 5 6 8 8 8

Now construct the 2D array as
2, 2
5, 2
6, 1
8, 3
Sort by count
8, 3
2, 2
5, 2
6, 1
```

How to maintain order of elements if frequency is same?

The above approach doesn't make sure order of elements if frequency is same. To handle this, we should use indexes in step 3, if two counts are same then we should first process(or print) the element with lower index. In step 1, we should store the indexes instead of elements.

```
Input 5  2  2  8  5  6  8  8

After sorting we get
Element 2 2 5 5 6 8 8 8
Index   1 2 0 4 5 3 6 7
```

Now construct the 2D array as
Index, Count

```
1,     2
0,     2
5,     1
3,     3
```

Sort by count (consider indexes in case of tie)

```
3, 3
0, 2
1, 2
5, 1
```

Print the elements using indexes in the above 2D array.

```
void sortByFrequency(int arr[], int n)
{
    struct ele element[n];
    for (int i = 0; i < n; i++)
    {
        element[i].index = i;    /* Fill Indexes */
        element[i].count = 0;    /* Initialize counts as 0 */
        element[i].val = arr[i]; /* Fill values in structure
                                    elements */
    }

    /* Sort the structure elements according to value,
       we used stable sort so relative order is maintained. */
    stable_sort(element, element+n, mycomp);

    /* initialize count of first element as 1 */
    element[0].count = 1;

    /* Count occurrences of remaining elements */
    for (int i = 1; i < n; i++)
    {
        if (element[i].val == element[i-1].val)
        {
            element[i].count += element[i-1].count+1;

            /* Set count of previous element as -1 , we are
               doing this because we'll again sort on the
               basis of counts (if counts are equal than on
               the basis of index)*/
            element[i-1].count = -1;

            /* Retain the first index (Remember first index
               is always present in the first duplicate we
               used stable sort. */
            element[i].index = element[i-1].index;
        }

        /* Else If previous element is not equal to current
          so set the count to 1 */
        else element[i].count = 1;
    }

    /* Now we have counts and first index for each element so now
       sort on the basis of count and in case of tie use index
       to sort.*/
    stable_sort(element, element+n, mycomp2);
    for (int i = n-1, index=0; i >= 0; i--)
        if (element[i].count != -1)
           for (int j=0; j<element[i].count; j++)
                arr[index++] = element[i].val;
 }
```

METHOD 2(Use BST and Sorting)

1. Insert elements in BST one by one and if an element is already present then increment the count of the node. Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct tree
{
  int element;
  int first_index /*To handle ties in counts*/
  int count;
}BST;
```

2.Store the first indexes and corresponding counts of BST in a 2D array.
3 Sort the 2D array according to counts (and use indexes in case of tie).

Time Complexity: O(nlogn) if a Self Balancing Binary Search Tree is used.

METHOD 3(Use Hashing and Sorting)
Using a hashing mechanism, we can store the elements (also first index) and their counts in a hash. Finally, sort the hash elements according to their counts.

13 Count Inversions in an array | Set 1 (Using Merge Sort)

http://www.geeksforgeeks.org/counting-inversions/
Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that 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:
The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

METHOD 1 (Simple)

For each element, count number of elements which are on right side of it and are smaller than it.
Time Complexity: O(n^2)

METHOD 2(Enhance Merge Sort)
Suppose we know 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 we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

```
/* 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;
}
```

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

Time Complexity: O(nlogn)
Algorithmic Paradigm: Divide and Conquer

14 Two elements whose sum is closest to zero

http://www.geeksforgeeks.org/two-elements-whose-sum-is-closest-to-zero/
An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

METHOD 1 (Simple)
For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.
Time complexity: O(n^2)

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

15 Find the smallest and second smallest element in an array

http://www.geeksforgeeks.org/to-find-smallest-and-second-smallest-element-in-an-array/

A Simple Solution is to sort the array in increasing order. The first two elements in sorted array would be two smallest elements. Time complexity of this solution is O(n Log n).
A Better Solution is to scan the array twice. In first traversal find the minimum element. Let this element be x. In second traversal, find the smallest element greater than x. Time
Time complexity : O(n).
The above solution requires two traversals of input array.
An Efficient Solution can find the minimum two elements in one traversal. Below is complete algorithm.

Algorithm:

```
1) Initialize both first and second smallest as INT_MAX
   first = second = INT_MAX
2) Loop through all the elements.
   a) If the current element is smaller than first, then update first
      and second.
   b) Else if the current element is smaller than second then update
    second
```

16 Check for Majority Element in a sorted array

http://www.geeksforgeeks.org/check-for-majority-element-in-a-sorted-array/

we need to write a function say isMajority() that takes an array (arr[]), array's size (n) and a number to be searched (x) as parameters and returns true if x is a majority element (present more than n/2 times).
Examples:
Input: arr[] = {1, 2, 3, 3, 3, 3, 10}, x = 3
Output: True (x appears more than n/2 times in the given array)

Input: arr[] = {1, 1, 2, 4, 4, 4, 6, 6}, x = 4
Output: False (x doesn't appear more than n/2 times in the given array)

Input: arr[] = {1, 1, 1, 2, 2}, x = 1
Output: True (x appears more than n/2 times in the given array)

METHOD 1 (Using Linear Search)

Linearly search for the first occurrence of the element, once you find it (let at index i), check element at index i + n/2. If element is present at i+n/2 then return 1 else return 0.
Time Complexity: O(n)

METHOD 2 (Using Binary Search)

Use binary search methodology to find the first occurrence of the given number. The criteria for binary search is important here.

```
bool isMajority(int arr[], int n, int x)
{
    /* Find the index of first occurrence of x in arr[] */
    int i = _binarySearch(arr, 0, n-1, x);

    /* If element is not present at all, return false*/
    if (i == -1)
        return false;

    /* check if the element is present more than n/2 times */
    if (((i + n/2) <= (n -1)) && arr[i + n/2] == x)
        return true;
    else
        return false;
}
```

Time Complexity: O(Logn)

Algorithmic Paradigm: Divide and Conquer

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

http://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/

First of all, how do we return multiple values from a C function? We can do it either using structures or pointers.

We have created a structure named pair (which contains min and max) to return multiple values.

```
struct pair
{
  int min;
  int max;
};
```

And the function declaration becomes: struct pair getMinMax(int arr[], int n) where arr[] is the array of size n whose minimum and maximum are needed.

METHOD 1 (Simple Linear Search)

Initialize values of min and max as minimum and maximum of the first two elements respectively. Starting from 3rd, compare each element with max and min, and change max and min accordingly (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element)

Time Complexity: O(n)

In this method, total number of comparisons is 1 + 2(n-2) in worst case and 1 + n – 2 in best case.

In the above implementation, worst case occurs when elements are sorted in descending order and best case occurs when elements are sorted in ascending order.

METHOD 2 (Tournament Method)

Divide the array into two parts and compare the maximums and minimums of the the two parts to get the maximum and the minimum of the 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
```

Time Complexity: O(n)

Total number of comparisons: let 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 above recursion, we get

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

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

METHOD 3 (Compare in Pairs)

If n is odd then initialize min and max as first element.
If n is even then initialize min and max as minimum and maximum of the first two elements respectively.
For rest of the elements, pick them in pairs and compare their
maximum and minimum with max and min respectively.

Time Complexity: O(n)
Total number of comparisons: Different for even and odd n, see below:
 If n is odd: 3*(n-1)/2
 If n is even: 1 Initial comparison for initializing min and max,
 and 3(n-2)/2 comparisons for rest of the elements
 = 1 + 3*(n-2)/2 = 3n/2 -2
Second and third approaches make equal number of comparisons when n is a power of 2.
In general, method 3 seems to be the best.

18 Segregate 0s and 1s in an array

http://www.geeksforgeeks.org/segregate-0s-and-1s-in-an-array-by-traversing-array-once/

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.
Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

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

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

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

```
void segregate0and1(int arr[], int size)
{
    /* Initialize left and right indexes */
    int left = 0, right = size-1;

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

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

19 k largest(or smallest) elements in an array

http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/

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.

Method 1 (Use Bubble k times)
1) Modify Bubble Sort to run the outer loop at most k times.
2) Print the last k elements of the array obtained in step 1.
Time Complexity: O(nk)
Like Bubble sort, other sorting algorithms like Selection Sort can also be modified to get the k largest elements.

Method 2 (Use temporary array)
K largest elements from arr[0..n-1]
1) Store the first k elements in a temporary array temp[0..k-1].
2) Find the smallest element in temp[], let the smallest element be min.
3) For each element x in arr[k] to arr[n-1]
If x is greater than the min then remove min from temp[] and insert x.
4) Print final k elements of temp[]

Time Complexity: O((n-k)*k). If we want the output sorted then O((n-k)*k + klogk)
Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

1) Sort the elements in descending order in O(nLogn)
2) Print the first k numbers of the sorted array O(k).
Time complexity: O(nlogn)

Method 4 (Use Max Heap)

1) Build a Max Heap tree in O(n)
2) Use Extract Max k times to get k maximum elements from the Max Heap O(klogn)
Time complexity: O(n + klogn)

Method 5(Use Oder Statistics)

1) Use order statistic algorithm to find the kth largest element. Please see the topic selection in worst-case linear time O(n)
2) Use QuickSort Partition algorithm to partition around the kth largest number O(n).
3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.
Time complexity: O(n) if we don't need the sorted output, otherwise O(n+kLogk)

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

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 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)
All of the above methods can also be used to find the kth largest (or smallest) element.

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

http://www.geeksforgeeks.org/maximum-difference-between-two-elements/

Given an array arr[] of integers, find out the difference between any two elements such that larger element appears after the smaller number in arr[].
Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then returned value should be 2 (Diff between 7 and 9)

Method 1 (Simple)

Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far.
Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Tricky and Efficient)

In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:
1) Maximum difference found so far (max_diff).
2) Minimum number visited so far (min_element).

```
int maxDiff(int arr[], int n)
{
    int maxDiff = -1; // Initialize Result

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

    for (int i = n-2; i >= 0; i--)
    {
        if (arr[i] > maxRight)
            maxRight = arr[i];
        else
        {
            int diff = maxRight - arr[i];
            if (diff > maxDiff)
            {
                maxDiff = diff;
            }
        }
    }
    return maxDiff;
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)
Like min element, we can also keep track of max element from right side.

Method 3 (Another Tricky Solution)

First find the difference between the adjacent elements of the array and store all differences in an auxiliary array diff[] of size n-1. Now this problems turns into finding the maximum sum subarray of this difference array.

```
int maxDiff(int arr[], int n)
{
    // Create a diff array of size n-1. The array will hold
    //  the difference of adjacent elements
    int diff[n-1];
    for (int i=0; i < n-1; i++)
        diff[i] = arr[i+1] - arr[i];
     // Now find the maximum sum subarray in diff array
    int max_diff = diff[0];
    for (int i=1; i<n-1; i++)
    {
        if (diff[i-1] > 0)
            diff[i] += diff[i-1];
        if (max_diff < diff[i])
            max_diff = diff[i];
    }
    return max_diff;
}
```

This method is also O(n) time complexity solution, but it requires O(n) extra space
Time Complexity: O(n)
Auxiliary Space: O(n)

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

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

        // Calculate current sum
        if (curr_sum > 0)
           curr_sum += diff;
        else
           curr_sum = diff;
         // Update max sum, if needed
        if (curr_sum > max_sum)
           max_sum = curr_sum;
    }
     return max_sum;
}
```

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

21 Union and Intersection of two sorted arrays

http://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/

Given two sorted arrays, find their union and intersection.
For example, if the input arrays are:
arr1[] = {1, 3, 4, 5, 7}
arr2[] = {2, 3, 5, 6}
Then your program should print Union as {1, 2, 3, 4, 5, 6, 7} and Intersection as {3, 5}.

Algorithm Union(arr1[], arr2[]):

For union of two arrays, follow the following merge procedure.
1) Use two index variables i and j, initial values i = 0, j = 0
2) If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i.
3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.
4) If both are same then print any of them and increment both i and j.
5) Print remaining elements of the larger array.
Time Complexity: O(m+n)

Algorithm Intersection(arr1[], arr2[]):

For Intersection of two arrays, print the element only if the element is present in both arrays.
1) Use two index variables i and j, initial values i = 0, j = 0
2) If arr1[i] is smaller than arr2[j] then increment i.
3) If arr1[i] is greater than arr2[j] then increment j.
4) If both are same then print any of them and increment both i and j.
Time Complexity: O(m+n)

Another approach that is useful when difference between sizes of two given arrays is significant.

The **idea** is to iterate through the shorter array and do a binary search for every element of short array in big array (note that arrays are sorted). Time complexity of this solution is **O(min(mLogn, nLogm)).** This solution works better than the above approach when ratio of larger length to smaller is more than logarithmic order.

22 Ceiling in a sorted array

http://www.geeksforgeeks.org/ceiling-in-a-sorted-array/

Given a sorted array and a value x, the ceiling of x is the smallest element in array greater than or equal to x, and the floor is the greatest element smaller than or equal to x. Assume than the array is sorted in non-decreasing order. Write efficient functions to find floor and ceiling of x.
For example, let the input array be {1, 2, 8, 10, 10, 12, 19}
For x = 0:   floor doesn't exist in array, ceil = 1
For x = 5:   floor = 2, ceil = 8
For x = 20:   floor = 19, ceil doesn't exist in array

Method 1 (Linear Search)

Algorithm to search ceiling of x:

1) If x is smaller than or equal to the first element in array then return 0(index of first element)
2) Else Linearly search for an index i such that x lies between arr[i] and arr[i+1].
3) If we do not find an index i in step 2, then return -1

Time Complexity: O(n)

Method 2 (Binary Search)

Instead of using linear search, binary search is used here to find out the index. Binary search reduces time complexity to O(Logn).

```
int ceilSearch(int arr[], int low, int high, int x)
{
  int mid;

  /* If x is smaller than or equal to the first element,
    then return the first element */
  if(x <= arr[low])
    return low;

  /* If x is greater than the last element, then return -1 */
  if(x > arr[high])
    return -1;

  /* get the index of middle element of arr[low..high]*/
  mid = (low + high)/2;  /* low + (high - low)/2 */

  /* If x is same as middle element, then return mid */
  if(arr[mid] == x)
    return mid;

  /* If x is greater than arr[mid], then either arr[mid + 1]
    is ceiling of x or ceiling lies in arr[mid+1...high] */
  else if(arr[mid] < x)
  {
    if(mid + 1 <= high && x <= arr[mid+1])
      return mid + 1;
    else
      return ceilSearch(arr, mid+1, high, x);
  }

  /* If x is smaller than arr[mid], then either arr[mid]
     is ceiling of x or ceiling lies in arr[mid-1...high] */
  else
  {
    if(mid - 1 >= low && x > arr[mid-1])
      return mid;
    else
      return ceilSearch(arr, low, mid - 1, x);
  }
}
```

23 A Product Array Puzzle

http://www.geeksforgeeks.org/a-product-array-puzzle/

Given an array arr[] of n integers, construct a Product Array prod[] (of same size) such that prod[i] is equal to the product of all the elements of arr[] except arr[i]. Solve it without division operator and in O(n).
Example:
arr[] = {10, 3, 5, 6, 2}
prod[] = {180, 600, 360, 300, 900}

Algorithm:
1) Construct a temporary array left[] such that left[i] contains product of all elements on left of arr[i] excluding arr[i].
2) Construct another temporary array right[] such that right[i] contains product of all elements on on right of arr[i] excluding arr[i].
3) To get prod[], multiply left[] and right[].
Time Complexity: O(n)
Space Complexity: O(n)
Auxiliary Space: O(n)

The above method can be optimized to work in space complexity O(1).

```
void productArray(int arr[], int n)
{
  int i, temp = 1;
   /* Allocate memory for the product array */
  int *prod = (int *)malloc(sizeof(int)*n);
   /* Initialize the product array as 1 */
  memset(prod, 1, n);

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

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

Time Complexity: O(n)
Space Complexity: O(n)
Auxiliary Space: O(1)
Related Problem :
Construct an Array from XOR of all elements of array except element at same index

24 Segregate Even and Odd numbers

http://www.geeksforgeeks.org/segregate-even-and-odd-numbers/

Given an array A[], write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.
Example
Input = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 8, 90, 45, 9, 3}
In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.
The problem is very similar to our old post Segregate 0s and 1s in an array, and both of these problems are variation of famous Dutch national flag problem.

Algorithm: segregateEvenOdd()

```
1) Initialize two index variables left and right:
        left = 0,  right = size -1
2) Keep incrementing left index until we see an odd number.
3) Keep decrementing right index until we see an even number.
4) If lef < right then swap arr[left] and arr[right]
```

Time Complexity: O(n)

25 Find the two repeating elements in a given array

http://www.geeksforgeeks.org/find-the-two-repeating-elements-in-a-given-array/

You are given an array of n+2 elements. All elements of the array are in range 1 to n. And all elements occur once except two numbers which occur twice. Find the two repeating numbers.
For example, array = {4, 2, 4, 5, 2, 3, 1} and n = 5
The above array has n + 2 = 7 elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

Method 1 (Basic)
Use two loops. In the outer loop, pick elements one by one and count the number of occurrences of the picked element in the inner loop.
This method doesn't use the other useful data provided in questions like range of numbers is between 1 to n and there are only two repeating elements.
Time Complexity: O(n*n)
Auxiliary Space: O(1)

Method 2 (Use Count array)
Traverse the array once. While traversing, keep track of count of all elements in the array using a temp array count[] of size n, when you see an element whose count is already set, print it as duplicate.
This method uses the range given in the question to restrict the size of count[], but doesn't use the data that there are only two repeating elements.
Time Complexity: O(n)
Auxiliary Space: O(n)

Method 3 (Make two equations)
Let the numbers which are being repeated are X and Y. We make two equations for X and Y and the simple task left is to solve the two equations.
We know the sum of integers from 1 to n is n(n+1)/2 and product is n!. We calculate the sum of input array, when this sum is subtracted from n(n+1)/2, we get X + Y because X and Y are the two numbers missing from set [1..n]. Similarly calculate product of input array, when this product is divided from n!, we get X*Y. Given sum and product of X and Y, we can find easily out X and Y.
Let summation of all numbers in array be S and product be P
X + Y = S – n(n+1)/2
XY = P/n!
Using above two equations, we can find out X and Y. For array = 4 2 4 5 2 3 1, we get S = 21 and P as 960.
X + Y = 21 – 15 = 6
XY = 960/5! = 8
X – Y = sqrt((X+Y)^2 – 4*XY) = sqrt(4) = 2
Using below two equations, we easily get X = (6 + 2)/2 and Y = (6-2)/2
X + Y = 6
X – Y = 2
there can be addition and multiplication overflow problem with this approach.
The methods 3 and 4 use all useful information given in the question
Time Complexity: O(n)
Auxiliary Space: O(1)

Method 4 (Use XOR)

The approach used here is similar to method 2 of this post.

Let the repeating numbers be X and Y, if we xor all the elements in the array and all integers from 1 to n, then the result is X xor Y.
The 1's in binary representation of X xor Y is corresponding to the different bits between X and Y. Suppose that the kth bit of X xor Y is 1, we can xor all the elements in the array and all integers from 1 to n, whose kth bits are 1. The result will be one of X and Y.

```
void printRepeating(int arr[], int size)
{
  int xor = arr[0]; /* Will hold xor of all elements */
  int set_bit_no;  /* Will have only single set bit of xor */
  int i;
  int n = size - 2;
  int x = 0, y = 0;

  /* Get the xor of all elements in arr[] and {1, 2 .. n} */
  for(i = 1; i < size; i++)
    xor ^= arr[i];
  for(i = 1; i <= n; i++)
    xor ^= i;

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

  /* Now divide elements in two sets by comparing rightmost set
   bit of xor with bit at same position in each element. */
  for(i = 0; i < size; i++)
  {
    if(arr[i] & set_bit_no)
      x = x ^ arr[i]; /*XOR of first set in arr[] */
    else
      y = y ^ arr[i]; /*XOR of second set in arr[] */
  }
  for(i = 1; i <= n; i++)
  {
    if(i & set_bit_no)
      x = x ^ i; /*XOR of first set in arr[] and {1, 2, ...n }*/
    else
      y = y ^ i; /*XOR of second set in arr[] and {1, 2, ...n } */
  }

  printf("\n The two repeating elements are %d & %d ", x, y);
}
```

26 Sort an array of 0s, 1s and 2s

http://www.geeksforgeeks.org/sort-an-array-of-0s-1s-and-2s/

Given an array A[] consisting 0s, 1s and 2s, write a function that sorts A[]. The functions should put all 0s first, then all 1s and all 2s in last.
Example
Input = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};
Output = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}

The problem is similar to our old post Segregate 0s and 1s in an array, and both of these problems are variation of famous Dutch national flag problem.
The problem was posed with three colours, here `0′, `1′ and `2′. The array is divided into four sections:

```
void sort012(int a[], int arr_size)
{
    int lo = 0;
    int hi = arr_size - 1;
    int mid = 0;

    while (mid <= hi)
    {
        switch (a[mid])
        {
        case 0:
            swap(&a[lo++], &a[mid++]);
            break;
        case 1:
            mid++;
            break;
        case 2:
            swap(&a[mid], &a[hi--]);
            break;
        }
    }
}
```

Time Complexity: O(n)
The above code performs unnecessary swaps for inputs like 0 0 0 0 1 1 1 2 2 2 2 2 : lo=4 and mid=7 and hi=11. In present code: first 7 exchanged with 11 and hi become 10 and mid is still pointing to 7. again same operation is till the mid <= hi. But it is really not required. We can change the swap function to do a check that the values being swapped are same or not, if not same, then only swap values.

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

http://www.geeksforgeeks.org/minimum-length-unsorted-subarray-sorting-which-makes-the-complete-array-sorted/

Given an unsorted array arr[0..n-1] of size n, find the minimum length subarray arr[s..e] such that sorting this subarray makes the whole array sorted.

Examples:

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*.

Time Complexity: O(n)

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

http://www.geeksforgeeks.org/find-duplicates-in-on-time-and-constant-extra-space/

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

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

This problem is an extended version of following problem.

Find the two repeating elements in a given array

Algorithm:

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

Note: The above program doesn't handle 0 case (If 0 is present in array). The program can be easily modified to handle that also. It is not handled to keep the code simple.
Time Complexity: O(n)
Auxiliary Space: O(1)

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

```
1- Traverse the given array from i= 0 to n-1 elements
     Go to index arr[i]%n and increment its value by n.
3- Now traverse the array again and print all those
   indexes i for which arr[i]/n is greater than 1.
This approach works because all elements are in range
from 0 to n-1 and arr[i]/n would be greater than 1
only if a value "i" has appeared more than once.
```

```
// function to find repeating elements
void printRepeating( int arr[], int n)
{
    // First check all the values that are
    // present in an array then go to that
    // values as indexes and increment by
    // the size of array
    for (int i = 0; i < n; i++)
    {
        int index = arr[i] % n;
        arr[index] += n;
    }

    // Now check which value exists more than one

    for (int i = 0; i < n; i++)
    {
        if ((arr[i]/n) > 1)
            cout << i << " ";
    }
}
```

29 Equilibrium index of an array

http://www.geeksforgeeks.org/equilibrium-index-of-an-array/

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an arrya A:
A[0] = -7, A[1] = 1, A[2] = 5, A[3] = 2, A[4] = -4, A[5] = 3, A[6]=0
3 is an equilibrium index, because:
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]
6 is also an equilibrium index, because sum of zero elements is zero, i.e., A[0] + A[1] + A[2] + A[3] + A[4] + A[5]=0
7 is not an equilibrium index, because it is not a valid index of array A.
Write a function int equilibrium(int[] arr, int n); that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

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

Method 2 (Tricky and Efficient)
The idea is to get total sum of array first. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get right sum by subtracting the elements one by one.

```
1) Initialize leftsum  as 0
2) Get the total sum of the array as sum
3) Iterate through the array and for each index i, do following.
   a)  Update sum to get the right sum.
         sum = sum - arr[i]
      // sum is now right sum
   b) If leftsum is equal to sum, then return current index.
   c) leftsum = leftsum + arr[i] // update leftsum for next iteration.
4) return -1 // If we come out of loop without returning then
```

Time Complexity: O(n)

30 Next Greater Element

http://www.geeksforgeeks.org/next-greater-element/

Elements for which no greater element exist, consider next greater element as -1.
a) For any array, rightmost element always has next greater element as -1.
b) For an array which is sorted in decreasing order, all element have next greater element as-1.
c) For the input array [4, 5, 2, 25}, the next greater elements for each element are as follows.

Element		NGE
4	-->	5
5	-->	25
2	-->	25
25	-->	-1

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Time Complexity: O(n^2). The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

1) Push the first element to stack.
2) Pick rest of the elements one by one and follow following steps in loop.
....a) Mark the current element as *next*.
....b) If stack is not empty, then pop an element from stack and compare it with *next*.
....c) If next is greater than the popped element, then *next* is the next greater element for the popped element.
....d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
....g) If *next* is smaller than the popped element, then push the popped element back.
3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

```
/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];
         if (isEmpty(&s) == false)
        {
            // if stack is not empty, then pop an element from stack
            element = pop(&s);
            /* If the popped element is smaller than next, then
                a) print the pair
                b) keep popping while elements are smaller and
                stack is not empty */
            while (element < next)
            {
                printf("\n %d --> %d", element, next);
                if(isEmpty(&s) == true)
                   break;
                element = pop(&s);
            }

            /* If element is greater than next, then push
               the element back */
            if (element > next)
                push(&s, element);
        }
         /* push next to stack so that we can find
           next greater for it */
        push(&s, next);
    }
     /* After iterating over the loop, the remaining
       elements in stack do not have the next greater
       element, so print -1 for them */
    while (isEmpty(&s) == false)
    {
        element = pop(&s);
        next = -1;
        printf("\n %d -- %d", element, next);
   }
}
```

31 Check if array elements are consecutive

http://www.geeksforgeeks.org/check-if-array-elements-are-consecutive/

Given an unsorted array of numbers, write a function that returns true if array consists of consecutive numbers.
Examples:
a) If array is {5, 2, 3, 1, 4}, then the function should return true because the array has consecutive numbers from 1 to 5.
b) If array is {83, 78, 80, 81, 79, 82}, then the function should return true because the array has consecutive numbers from 78 to 83.
c) If the array is {34, 23, 52, 12, 3 }, then the function should return false because the elements are not consecutive.
d) If the array is {7, 6, 5, 5, 3, 4}, then the function should return false because 5 and 5 are not consecutive.

Method 1 (Use Sorting)
1) Sort all the elements.
2) Do a linear scan of the sorted array. If the difference between current element and next element is anything other than 1, then return false. If all differences are 1, then return true.
Time Complexity: O(nLogn)

Method 2 (Use visited array)
The idea is to check for following two conditions. If following two conditions are true, then return true.
1) max – min + 1 = n where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
2) All elements are distinct.
To check if all elements are distinct, we can create a visited[] array of size n. We can map the ith element of input array arr[] to visited array by using arr[i] – min as index in visited[].
Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Mark visited array elements as negative)
This method is O(n) time complexity and O(1) extra space, but it changes the original array and it works only if all numbers are positive. We can get the original array by adding an extra step though. It is an extension of method 2 and it has the same two steps.
1) max – min + 1 = n where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
2) All elements are distinct.

In this method, the implementation of step 2 differs from method 2. Instead of creating a new array, we modify the input array arr[] to keep track of visited elements. The idea is to traverse the array and for each index i (where 0 <= i < n), make arr[arr[i] – min]] as a negative value. If we see a negative value again then there is repetition.
Note that this method might not work for negative numbers. For example, it returns false for {2, 1, 0, -3, -1, -2}.
Time Complexity: O(n)
Extra Space: O(1)

32 Find the smallest missing number

http://www.geeksforgeeks.org/find-the-first-missing-number/

Given a sorted array of n distinct integers where each integer is in the range from 0 to m-1 and m > n. Find the smallest number that is missing from the array.
Input: {0, 1, 2, 6, 9}, n = 5, m = 10
Output: 3
Input: {4, 5, 10, 11}, n = 4, m = 12
Output: 0

Input: {0, 1, 2, 3}, n = 4, m = 5
Output: 4
Input: {0, 1, 2, 3, 4, 5, 6, 7, 10}, n = 9, m = 11
Output: 8

Method 1 (Use Binary Search)
For i = 0 to m-1, do binary search for i in the array. If i is not present in the array then return i.
Time Complexity: O(m log n)
Method 2 (Linear Search)
If arr[0] is not 0, return 0. Otherwise traverse the input array starting from index 0, and for each pair of elements a[i] and a[i+1], find the difference between them. if the difference is greater than 1 then a[i]+1 is the missing number.
Time Complexity: O(n)

Method 3 (Use Modified Binary Search)
In the standard Binary Search process, the element to be searched is compared with the middle element and on the basis of comparison result, we decide whether to search is over or to go to left half or right half.
In this method, we modify the standard Binary Search algorithm to compare the middle element with its index and make decision on the basis of this comparison.

```
...1) If the first element is not same as its index then return first index
...2) Else get the middle index say mid
............a) If arr[mid] greater than mid then the required element lies in left half.
............b) Else the required element lies in right half.
Note: This method doesn't work if there are duplicate elements in the array.
```

Time Complexity: O(Logn)

33 Count number of occurrences in a sorted array

http://www.geeksforgeeks.org/count-number-of-occurrences-in-a-sorted-array/

Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[]. Expected time complexity is O(Logn)
Examples:
  Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 4
 Output: -1 // 4 doesn't occur in arr[]

Method 1 (Linear Search)
Linearly search for x, count the occurrences of x and return the count.
Time Complexity: O(n)

Method 2 (Use Binary Search)
1) Use Binary search to get index of the first occurrence of x in arr[]. Let the index of the first occurrence be i.
2) Use Binary search to get index of the last occurrence of x in arr[]. Let the index of the last occurrence be j.
3) Return (j – i + 1);
Time Complexity: O(Logn)
Programming Paradigm: Divide & Conquer

34 Given an array arr[], find the maximum j – i such that arr[j] > arr[i]

http://www.geeksforgeeks.org/given-an-array-arr-find-the-maximum-j-i-such-that-arrj-arri/

```
Given an array arr[], find the maximum j – i such that arr[j] > arr[i].
Examples:
 Input: {34, 8, 10, 3, 2, 80, 30, 33, 1}
 Output: 6  (j = 7, i = 1)
 Input: {9, 2, 3, 4, 5, 6, 7, 8, 18, 0}
 Output: 8 ( j = 8, i = 0)
 Input:  {1, 2, 3, 4, 5, 6}
 Output: 5  (j = 5, i = 0)

 Input:  {6, 5, 4, 3, 2, 1}
 Output: -1
```

Method 1 (Simple but Inefficient)
Run two loops. In the outer loop, pick elements one by one from left. In the inner loop, compare the picked element with the elements starting from right side. Stop the inner loop when you see an element greater than the picked element and keep updating the maximum j-i so far.
Time Complexity: O(n^2)

Method 2 (Efficient)
To solve this problem, we need to get two optimum indexes of arr[]: left index i and right index j. For an element arr[i], we do not need to consider arr[i] for left index if there is an element smaller than arr[i] on left side of arr[i]. Similarly, if there is a greater element on right side of arr[j] then we do not need to consider this j for right index. So we construct two auxiliary arrays LMin[] and RMax[] such that LMin[i] holds the smallest element on left side of arr[i] including arr[i], and RMax[j] holds the greatest element on right side of arr[j] including arr[j]. After constructing these two auxiliary arrays, we traverse both of these arrays from left to right. While traversing LMin[] and RMa[] if we see that LMin[i] is greater than RMax[j], then we

must move ahead in LMin[] (or do i++) because all elements on left of LMin[i] are greater than or equal to LMin[i]. Otherwise we must move ahead in RMax[j] to look for a greater j – i value.

```
/* For a given array arr[], returns the maximum j – i such that
    arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff;
    int i, j;
     int *LMin = (int *)malloc(sizeof(int)*n);
    int *RMax = (int *)malloc(sizeof(int)*n);

   /* Construct LMin[] such that LMin[i] stores the minimum value
       from (arr[0], arr[1], ... arr[i]) */
    LMin[0] = arr[0];
    for (i = 1; i < n; ++i)
        LMin[i] = min(arr[i], LMin[i-1]);

    /* Construct RMax[] such that RMax[j] stores the maximum value
       from (arr[j], arr[j+1], ..arr[n-1]) */
    RMax[n-1] = arr[n-1];
    for (j = n-2; j >= 0; --j)
        RMax[j] = max(arr[j], RMax[j+1]);

    /* Traverse both arrays from left to right to find optimum j - i
        This process is similar to merge() of MergeSort */
    i = 0, j = 0, maxDiff = -1;
    while (j < n && i < n)
    {
        if (LMin[i] < RMax[j])
        {
            maxDiff = max(maxDiff, j-i);
            j = j + 1;
        }
        else
            i = i+1;
    }
     return maxDiff;
}
```

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

35 Sliding Window Maximum (Maximum of all subarrays of size k)

http://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-size-k/

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

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

k = 3
Output :
3 3 4 5 5 5 6

Method 1 (Simple)
Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.
Time Complexity: The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(nk).

Method 2 (Use Self-Balancing BST)
1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
2) Run a loop for i = 0 to n – k
…..a) Get the maximum element from the BST, and print it.
…..b) Search for arr[i] in the BST and delete it from the BST.
…..c) Insert arr[i+k] into the BST.
Time Complexity: Time Complexity of step 1 is O(kLogk). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). Since steps 2(a), 2(b) and 2(c) are in a loop that runs n-k+1 times, time complexity of the complete algorithm is O(kLogk + (n-k+1)*Logk) which can also be written as O(nLogk).

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

```
void printKMax(int arr[], int n, int k)
{

    std::deque<int>  Qi(k);

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

```
            Qi.pop_back();  // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }
     // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

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

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

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

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

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

36 Find whether an array is subset of another array

http://www.geeksforgeeks.org/find-whether-an-array-is-subset-of-another-array-set-1/
Given two arrays: arr1[0..m-1] and arr2[0..n-1]. Find whether arr2[] is a subset of arr1[] or not. Both the arrays are not in sorted order. It may be assumed that elements in both array are distinct.
Input: arr1[] = {1, 2, 3, 4, 5, 6}, arr2[] = {1, 2, 4}
Output: arr2[] is a subset of arr1[]
Input: arr1[] = {10, 5, 2, 23, 19}, arr2[] = {19, 5, 3}
Output: arr2[] is not a subset of arr1[]

Method 1 (Simple)
Use two loops: The outer loop picks all the elements of arr2[] one by one. The inner loop linearly searches for the element picked by outer loop. If all elements are found then return 1, else return 0.
Time Complexity: O(m*n)
Method 2 (Use Sorting and Binary Search)
1) Sort arr1[] O(mLogm)
2) For each element of arr2[], do binary search for it in sorted arr1[].
    a) If the element is not found then return 0.
3) If all elements are present then return 1.
Time Complexity: O(mLogm + nLogm). Please note that this will be the complexity if an mLogm algorithm is used for sorting which is not the case in above code. In above code Quick Sort is sued and worst case **time complexity** of Quick Sort is O(m^2)

Method 3 (Use Sorting and Merging)
1) Sort both arrays: arr1[] and arr2[] O(mLogm + nLogn)
2) Use Merge type of process to see if all elements of sorted arr2[] are present in sorted 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 sued and worst case time complexity of Quick Sort is O(n^2)

Method 4 (Use Hashing)
1) Create a Hash Table for all the elements of arr1[].
2) Traverse arr2[] and search for each element of arr2[] in the Hash Table. If element is not found then return 0.
3) If all elements are found then return 1.
Note that method 1, method 2 and method 4 don't handle the cases when we have duplicates in arr2[]. For example, {1, 4, 4, 2} is not a subset of {1, 4, 2}, but these methods will print it as a subset.

37 Find the minimum distance between two numbers

http://www.geeksforgeeks.org/find-the-minimum-distance-between-two-numbers/

Given an unsorted array arr[] and two numbers x and y, find the minimum distance between x and y in arr[]. The array might also contain duplicates. You may assume that both x and y are different and present in arr[].
Examples:
Input: arr[] = {1, 2}, x = 1, y = 2
Output: Minimum distance between 1 and 2 is 1.
Input: arr[] = {3, 4, 5}, x = 3, y = 5
Output: Minimum distance between 3 and 5 is 2.

```
Input: arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3}, x = 3, y = 6
Output: Minimum distance between 3 and 6 is 4.
Input: arr[] = {2, 5, 3, 5, 4, 4, 2, 3}, x = 3, y = 2
Output: Minimum distance between 3 and 2 is 1.
```

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of arr[] one by one. The inner loop picks all the elements after the element picked by outer loop. If the elements picked by outer and inner loops have same values as x or y then if needed update the minimum distance calculated so far.

Time Complexity: O(n^2)

Method 2 (Tricky)

1) Traverse array from left side and stop if either x or y is found. Store index of this first occurrence in a variable say prev
2) Now traverse arr[] after the index prev. If the element at current index i matches with either x or y then check if it is different from arr[prev]. If it is different then update the minimum distance if needed. If it is same then update prev i.e., make prev = i.

Time Complexity: O(n)

38 Find the repeating and the missing

http://www.geeksforgeeks.org/find-a-repeating-and-a-missing-number/

Given an unsorted array of size n. Array elements are in range from 1 to n. One number from set {1, 2, …n} is missing and one number occurs twice in array. Find these two numbers.

```
Examples:
  arr[] = {3, 1, 3}
  Output: 2, 3   // 2 is missing and 3 occurs twice

  arr[] = {4, 3, 6, 2, 1, 1}
  Output: 1, 5  // 5 is missing and 1 occurs twice
```

Method 1 (Use Sorting)

1) Sort the input array.
2) Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Method 2 (Use count array)

1) Create a temp array temp[] of size n with all initial values as 0.
2) Traverse the input array arr[], and do following for each arr[i]
……a) if(temp[arr[i]] == 0) temp[arr[i]] = 1;

……b) if(temp[arr[i]] == 1) output “arr[i]” //repeating
3) Traverse temp[] and output the array element having value as 0 (This is the missing element)
Time Complexity: O(n)
Auxiliary Space: O(n)
Method 3 (Use elements as Index and mark the visited places)
Traverse the array. While traversing, use absolute value of every element as index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.
Time Complexity: O(n)

Method 4 (Make two equations)
Let x be the missing and y be the repeating element.
1) Get sum of all numbers.
Sum of array computed S = n(n+1)/2 – x + y
2) Get product of all numbers.
Product of array computed P = 1*2*3*…*n * y / x
3) The above two steps give us two equations, we can solve the equations and get the values of x and y.
Time Complexity: O(n)
This method can cause arithmetic overflow as we calculate product and sum of all array elements. See this for changes suggested by john to reduce the chances of overflow.

Method 5 (Use XOR)
Let x and y be the desired output elements.
Calculate XOR of all the array elements.
  xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
XOR the result with all numbers from 1 to n
  xor1 = xor1^1^2^.....^n
In the result xor1, all elements would nullify each other except x and y. All the bits that are set in xor1 will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of xor1 and divide the elements of the array in two sets – one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

```
/* The output of this function is stored at *x and *y */
void getTwoElements(int arr[], int n, int *x, int *y)
{
  int xor1;   /* Will hold xor of all elements and numbers from 1 to n */
  int set_bit_no;  /* Will have only single set bit of xor1 */
```

```
   int i;
   *x = 0;
   *y = 0;
    xor1 = arr[0];
    /* Get the xor of all array elements elements */
   for(i = 1; i < n; i++)
      xor1 = xor1^arr[i];
    /* XOR the previous result with numbers from 1 to n*/
   for(i = 1; i <= n; i++)
      xor1 = xor1^i;
    /* Get the rightmost set bit in set_bit_no */
   set_bit_no = xor1 & ~(xor1-1);
    /* Now divide elements in two sets by comparing rightmost set
    bit of xor1 with bit at same position in each element. Also, get XORs
    of two sets. The two XORs are the output elements.
    The following two for loops serve the purpose */
   for(i = 0; i < n; i++)
   {
     if(arr[i] & set_bit_no)
      *x = *x ^ arr[i]; /* arr[i] belongs to first set */
     else
      *y = *y ^ arr[i]; /* arr[i] belongs to second set*/
   }
   for(i = 1; i <= n; i++)
   {
     if(i & set_bit_no)
      *x = *x ^ i; /* i belongs to first set */
     else
      *y = *y ^ i; /* i belongs to second set*/
   }
    /* Now *x and *y hold the desired output elements */
}
```

Time Complexity: O(n)
This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing. We can add one more step that checks which one is missing and which one is repeating. This can be easily done in O(n) time.

39 Median in a stream of integers (running integers)

http://www.geeksforgeeks.org/median-of-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.

Method 1: Insertion Sort
If we can sort the data as it appears, we can easily locate median element. Insertion Sort is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting i-th element, the first i elements of array are sorted. The insertion sort doesn't depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting next element. This is the key part of insertion sort that makes it an online algorithm.
However, insertion sort takes O(n2) time to sort n elements. Perhaps we can use binary search on insertion sort to find location of next element in O(log n) time. Yet, we can't do data movement in O(log n) time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.
Interested reader can try implementation of Method 1

Method 2: Augmented self balanced binary search tree (AVL, RB, etc...)

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.

Method 3: Heaps
Similar to balancing BST in Method 2 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.

```
// Function implementing algorithm to find median so far.
int getMedian(int e, int &m, Heap &l, Heap &r)
{

 // Are heaps balanced? If yes, sig will be 0
    int sig = Signum(l.GetCount(), r.GetCount());
    switch(sig)
    {
    case 1: // There are more elements in left (max) heap

        if( e < m ) // current element fits in left (max) heap
        {
            // Remore top element from left heap and
            // insert into right heap
            r.Insert(l.ExtractTop());
             // current element fits in left (max) heap
            l.Insert(e);
        }
        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());
         break;

    case 0: // The left and right heaps contain same number of elements

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
            m = l.GetTop();
        }
        else
        {   // current element fits in right (min) heap
            r.Insert(e);
            m = r.GetTop();
        }

        break;

    case -1: // There are more elements in right (min) heap

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
        }
        else
        {
            // Remove top element from right heap and
```

```
                // insert into left heap
                l.Insert(r.ExtractTop());

                // current element fits in right (min) heap
                r.Insert(e);
            }

            // Both heaps are balanced
            m = Average(l.GetTop(), r.GetTop());

            break;
        }

    // No need to return, m already updated
    return m;
}
```

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.
At first glance the above code may look complex. If you read the code carefully, it is simple algorithm.

40 Find a Fixed Point in a given array

http://www.geeksforgeeks.org/find-a-fixed-point-in-a-given-array/

Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1. Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.
Examples:

```
 Input: arr[] = {-10, -5, 0, 3, 7}
 Output: 3  // arr[3] == 3
 Input: arr[] = {0, 2, 5, 8, 17}
 Output: 0  // arr[0] == 0
 Input: arr[] = {-10, -5, 3, 4, 7, 9}
 Output: -1  // No Fixed Point
```

Method 1 (Linear Search)
Linearly search for an index i such that arr[i] == i. Return the first such index found.
Time Complexity: O(n)

Method 2 (Binary Search)
First check whether middle element is Fixed Point or not. If it is, then return it; otherwise check whether index of middle element is greater than value at the index. If index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on left side.

Algorithmic Paradigm: Divide & Conquer
Time Complexity: O(Logn)

41 Maximum Length Bitonic Subarray

http://www.geeksforgeeks.org/maximum-length-bitonic-subarray/

Given an array A[0 … n-1] containing n positive integers, a subarray A[i … j] is bitonic if there is a k with i <= k <= j such that A[i] <= A[i + 1] ... <= A[k] >= A[k + 1] >= .. A[j – 1] > = A[j]. Write a function that takes an array as argument and returns the length of the maximum length bitonic subarray.
Expected time complexity of the solution is O(n)
Simple Examples
1) A[] = {12, 4, 78, 90, 45, 23}, the maximum length bitonic subarray is {4, 78, 90, 45, 23} which is of length 5.
2) A[] = {20, 4, 1, 2, 3, 4, 2, 10}, the maximum length bitonic subarray is {1, 2, 3, 4, 2} which is of length 5.
Extreme Examples
1) A[] = {10}, the single element is bitnoic, so output is 1.
2) A[] = {10, 20, 30, 40}, the complete array itself is bitonic, so output is 4.
3) A[] = {40, 30, 20, 10}, the complete array itself is bitonic, so output is 4.

Solution
Let us consider the array {12, 4, 78, 90, 45, 23} to understand the soultion.
1) Construct an auxiliary array inc[] from left to right such that inc[i] contains length of the nondecreaing subarray ending at arr[i].
For for A[] = {12, 4, 78, 90, 45, 23}, inc[] is {1, 1, 2, 3, 1, 1}
2) Construct another array dec[] from right to left such that dec[i] contains length of nonincreasing subarray starting at arr[i].
For A[] = {12, 4, 78, 90, 45, 23}, dec[] is {2, 1, 1, 3, 2, 1}.
3) Once we have the inc[] and dec[] arrays, all we need to do is find the maximum value of (inc[i] + dec[i] – 1).
For {12, 4, 78, 90, 45, 23}, the max value of (inc[i] + dec[i] – 1) is 5 for i = 3.

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

42 Find the maximum element in an array which is first increasing and then decreasing

http://www.geeksforgeeks.org/find-the-maximum-element-in-an-array-which-is-first-increasing-and-then-decreasing/

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.
Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
Output: 500

Input: arr[] = {1, 3, 50, 10, 9, 7, 6}
Output: 50

Corner case (No decreasing part)
Input: arr[] = {10, 20, 30, 40, 50}
Output: 50

Corner case (No increasing part)
Input: arr[] = {120, 100, 80, 20, 0}
Output: 120

Method 1 (Linear Search)
We can traverse the array and keep track of maximum and element. And finally return the maximum element.
Time Complexity: O(n)

Method 2 (Binary Search)

We can modify the standard Binary Search algorithm for the given type of arrays.
i) If the mid element is greater than both of its adjacent elements, then mid is the maximum.
ii) If mid element is greater than its next element and smaller than the previous element then maximum lies on left side of mid. Example array: {3, 50, 10, 9, 7, 6}
iii) If mid element is smaller than its next element and greater than the previous element then maximum lies on right side of mid. Example array: {2, 4, 6, 8, 10, 3, 1}

```
int findMaximum(int arr[], int low, int high)
{
    /* Base Case: Only one element is present in arr[low..high]*/
   if (low == high)
     return arr[low];
    /* If there are two elements and first is greater then
      the first element is maximum */
   if ((high == low + 1) && arr[low] >= arr[high])
```

```
      return arr[low];

   /* If there are two elements and second is greater then
      the second element is maximum */
   if ((high == low + 1) && arr[low] < arr[high])
      return arr[high];

   int mid = (low + high)/2;   /*low + (high - low)/2;*/

   /* If we reach a point where arr[mid] is greater than both of
     its adjacent elements arr[mid-1] and arr[mid+1], then arr[mid]
     is the maximum element*/
   if ( arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1])
      return arr[mid];

   /* If arr[mid] is greater than the next element and smaller than the
previous
    element then maximum lies on left side of mid */
   if (arr[mid] > arr[mid + 1] && arr[mid] < arr[mid - 1])
     return findMaximum(arr, low, mid-1);
   else // when arr[mid] is greater than arr[mid-1] and smaller than
arr[mid+1]
     return findMaximum(arr, mid + 1, high);
}
```

Time Complexity: O(Logn)
This method works only for distinct numbers. For example, it will not work for an array like {0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 5, 3, 3, 2, 2, 1, 1}.

43 Count smaller elements on right side

http://www.geeksforgeeks.org/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}

Method 1 (Simple)
Use two loops. The outer loop picks all elements from left to right. The inner loop iterates through all the elements on right side of the picked element and updates countSmaller[].
Time Complexity: O(n^2)
Auxiliary Space: O(1)

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

44 Implement two stacks in an array

http://www.geeksforgeeks.org/implement-two-stacks-in-an-array/

Create a data structure twoStacks that represents two stacks. Implementation of twoStacks should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by twoStacks.

push1(int x) –> pushes x to first stack
push2(int x) –> pushes x to second stack
pop1() –> pops an element from first stack and return the popped element
pop2() –> pops an element from second stack and return the popped element
Implementation of twoStack should be space efficient.

Method 1 (Divide the space in two halves)
A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[n/2+1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.
The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)
This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks.
Time complexity of all 4 operations of twoStack is O(1).
We will extend to 3 stacks in an array in a separate post.

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

http://www.geeksforgeeks.org/find-subarray-with-given-sum/

Given an unsorted array of nonnegative integers, find a continous subarray which adds to a given number.
Examples:
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Ouptut: Sum found between indexes 2 and 4

Input: arr[] = {1, 4, 0, 0, 3, 10, 5}, sum = 7
Ouptut: Sum found between indexes 1 and 4

Input: arr[] = {1, 4}, sum = 0
Output: No subarray found
There may be more than one subarrays with sum as the given sum. The following solutions print first such subarray.

Method 1 (Simple)

A simple solution is to consider all subarrays one by one and check the sum of every subarray. Following program implements the simple solution. We run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i.
Time Complexity: O(n^2) in worst case.

Method 2 (Efficient)

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

```
/* Returns true if the there is a subarray of arr[] with sum equal to
'sum'
   otherwise returns false.  Also, prints the result */
int subArraySum(int arr[], int n, int sum)
{
    /* Initialize curr_sum as value of first element
       and starting point as 0 */
    int curr_sum = arr[0], start = 0, i;

    /* Add elements one by one to curr_sum and if the curr_sum exceeds the
       sum, then remove starting element */
    for (i = 1; i <= n; i++)
    {
        // If curr_sum exceeds the sum, then remove the starting elements
        while (curr_sum > sum && start < i-1)
        {
            curr_sum = curr_sum - arr[start];
            start++;
        }
         // If curr_sum becomes equal to sum, then return true
        if (curr_sum == sum)
        {
            printf ("Sum found between indexes %d and %d", start, i-1);
            return 1;
        }
         // Add this element to curr_sum
        if (i < n)
          curr_sum = curr_sum + arr[i];
    }
     // If we reach here, then no subarray
    printf("No subarray found");
    return 0;
}
```

Time complexity of method 2 looks more than O(n), but if we take a closer look at the program, then we can figure out the time complexity is O(n). We can prove it by counting the number of operations performed on every element of arr[] in worst case. There are at most 2 operations performed on every element: (a) the element is added to the curr_sum (b) the element is subtracted from curr_sum. So the upper bound on number of operations is 2n which is O(n).
The above solution doesn't handle negative numbers. We can using hashing to handle negative numbers. See below set 2.

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

http://www.geeksforgeeks.org/find-subarray-with-given-sum-in-array-of-integers/

Given an unsorted array of integers, find a subarray which adds to a given number. If there are more than one subarrays with sum as the given number, print any of them.
Examples:
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Ouptut: Sum found between indexes 2 and 4

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

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

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

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

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

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

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

```
                return;
            }

            // If curr_sum - sum already exists in map
            // we have found a subarray with target sum
            if (map.find(curr_sum - sum) != map.end())
            {
                cout << "Sum found between indexes "
                     << map[curr_sum - sum] + 1
                     << " to " << i << endl;
                return;
            }
             map[curr_sum] = i;
        }
         // If we reach here, then no subarray exists
        cout << "No subarray with given sum exists";
}
```

Time complexity of above solution is O(n) as we are doing only one traversal of the array.
Auxiliary space used by the program is O(n).

47 Find a triplet that sum to a given value

http://www.geeksforgeeks.org/find-a-triplet-that-sum-to-a-given-value/

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

Method 1 (Naive)
A simple method is to generate all possible triplets and compare the sum of every triplet with the given value. The following code implements this simple method using three nested loops.
Time Complexity: O(n^3)

Method 2 (Hashing : O(n2))
We iterate through every element. For every element arr[i], we find a pair with sum "-arr[i]". This problem reduces to pairs sum and can be solved in O(n) time using hashing.

```
Run a loop from i=0 to n-2
  Create an empty hash table
  Run inner loop from j=i+1 to n-1
    If -(arr[i] + arr[j]) is present in hash table
      print arr[i], arr[j] and -(arr[i]+arr[j])
    Else
      Insert arr[j] in hash table.
```

Method 2 (Use Sorting)

Time complexity of the method 1 is O(n^3). The complexity can be reduced to O(n^2) by sorting the array first, and then using method 1 of this post in a loop.

1) Sort the input array.

2) Fix the first element as A[i] where i is from 0 to array size – 2. After fixing the first element of triplet, find the other two elements using method 1 of this post.

```
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;
}
```

Time Complexity: O(n^2)

How to print all triplets with given sum?

Please refer Find all triplets with zero sum

http://www.geeksforgeeks.org/find-triplets-array-whose-sum-equal-zero/

48 Find the smallest positive number missing from an unsorted array

http://www.geeksforgeeks.org/find-the-smallest-positive-number-missing-from-an-unsorted-array/

You are given an unsorted array with both positive and negative elements. You have to find the smallest positive number missing from the array in O(n) time using constant extra space. You can modify the original array.
Examples
Input: {2, 3, 7, 6, 8, -1, -10, 15}
Output: 1

Input: { 2, 3, -7, 6, 8, 1, -10, 15 }
Output: 4

Input: {1, 1, 0, -1, -2}
Output: 2

A naive method to solve this problem is to search all positive integers, starting from 1 in the given array. We may have to search at most n+1 numbers in the given array. So this solution takes O(n^2) in worst case.

We can use sorting to solve it in lesser time complexity. We can sort the array in O(nLogn) time. Once the array is sorted, then all we need to do is a linear scan of the array. So this approach takes O(nLogn + n) time which is O(nLogn).
We can also use hashing. We can build a hash table of all positive elements in the given array. Once the hash table is built. We can look in the hash table for all positive integers, starting from 1. As soon as we find a number which is not there in hash table, we return it. This approach may take O(n) time on average, but it requires O(n) extra space.
A O(n) time and O(1) extra space solution:
The idea is similar to this post. We use array elements as index. To mark presence of an element x, we change the value at the index x to negative. But this approach doesn't work if there are non-positive (-ve and 0) numbers. So we segregate positive from negative numbers as first step and then apply the approach.

Following is the two step algorithm.
1) Segregate positive numbers from others i.e., move all non-positive numbers to left side. In the following code, segregate() function does this part.
2) Now we can ignore non-positive elements and consider only the part of array which contains all positive elements. We traverse the array containing all positive numbers and to mark presence of an element x, we change the sign of value at index x to negative. We traverse

the array again and print the first index which has positive value. In the following code, findMissingPositive() function does this part. Note that in findMissingPositive, we have subtracted 1 from the values as indexes start from 0 in C.

```
/* Utility function that puts all non-positive (0 and negative) numbers on
left
  side of arr[] and return count of such numbers */
int segregate (int arr[], int size)
{
    int j = 0, i;
    for(i = 0; i < size; i++)
    {
       if (arr[i] <= 0)
       {
           swap(&arr[i], &arr[j]);
           j++;  // increment count of non-positive integers
       }
    }

    return j;
}
```

```
/* Find the smallest positive missing number in an array that contains
  all positive integers */
int findMissingPositive(int arr[], int size)
{
  int i;

  // Mark arr[i] as visited by making arr[arr[i] - 1] negative. Note that
  // 1 is subtracted because index start from 0 and positive numbers start
from 1
  for(i = 0; i < size; i++)
  {
    if(abs(arr[i]) - 1 < size && arr[abs(arr[i]) - 1] > 0)
      arr[abs(arr[i]) - 1] = -arr[abs(arr[i]) - 1];
  }
   // Return the first index value at which is positive
  for(i = 0; i < size; i++)
    if (arr[i] > 0)
      return i+1;  // 1 is added becuase indexes start from 0
   return size+1;
}
```

```
/* Find the smallest positive missing number in an array that contains
  both positive and negative integers */
int findMissing(int arr[], int size)
{
   // First separate positive and negative numbers
   int shift = segregate (arr, size);

   // Shift the array and call findMissingPositive for
   // positive part
   return findMissingPositive(arr+shift, size-shift);
}
```

Note that this method modifies the original array. We can change the sign of elements in the segregated array to get the same set of elements back. But we still loose the order of elements. If we want to keep the original array as it was, then we can create a copy of the array and run this approach on the temp array.

49 Find the two numbers with odd occurrences in an unsorted array

http://www.geeksforgeeks.org/find-the-two-numbers-with-odd-occurences-in-an-unsorted-array/

Given an unsorted array that contains even number of occurrences for all numbers except two numbers. Find the two numbers which have odd occurrences in O(n) time complexity and O(1) extra space.
Examples:
Input: {12, 23, 34, 12, 12, 23, 12, 45}
Output: 34 and 45

Input: {4, 4, 100, 5000, 4, 4, 4, 4, 100, 100}
Output: 100 and 5000

Input: {10, 20}
Output: 10 and 20

A naive method **to solve this problem is to run two nested loops**. The outer loop picks an element and the inner loop counts the number of occurrences of the picked element. If the count of occurrences is odd then print the number. **The time complexity of this method is O(n^2).**

We can use sorting **to get the odd occurring numbers in O(nLogn) time**. First sort the numbers using an O(nLogn) sorting algorithm like Merge Sort, Heap Sort.. etc. Once the array is sorted, all we need to do is a linear scan of the array and print the odd occurring number.
We can also use hashing. Create an empty hash table which will have elements and their counts. Pick all elements of input array one by one. Look for the picked element in hash table. If the element is found in hash table, increment its count in table. If the element is not found, then enter it in hash table with count as 1. After all elements are entered in hash table, scan the hash table and print elements with odd count. This approach may take O(n) time on average, but it requires O(n) extra space.

A O(n) time and O(1) extra space solution:
The idea is similar to method 2 of this post. Let the two odd occurring numbers be x and y. We use bitwise XOR to get x and y. The first step is to do XOR of all elements present in array. XOR of all elements gives us XOR of x and y because of the following properties of XOR operation.
1) XOR of any number n with itself gives us 0, i.e., n ^ n = 0
2) XOR of any number n with 0 gives us n, i.e., n ^ 0 = n
3) XOR is cumulative and associative.
So we have XOR of x and y after the first step. Let the value of XOR be xor2. Every set bit in xor2 indicates that the corresponding bits in x and y have values different from each other. For example, if x = 6 (0110) and y is 15 (1111), then xor2 will be (1001), the two set bits in xor2 indicate that the corresponding bits in x and y are different. In the second step, we pick a set bit of xor2 and divide array elements in two groups. Both x and y will go to different groups. In the following code, the rightmost set bit of xor2 is picked as it is easy to get rightmost set bit of a number. If we do XOR of all those elements of array which have the corresponding bit set (or 1), then we get the first odd number. And if we do XOR of all those elements which have the corresponding bit 0, then we get the other odd occurring number. This step works because of the same properties of XOR. All the occurrences of a number will go in same set. XOR of all occurrences of a number which occur even number number of times will result in 0 in its set. And the xor of a set will be one of the odd occurring elements.

```
void printTwoOdd(int arr[], int size)
{
  int xor2 = arr[0]; /* Will hold XOR of two odd occurring elements */
  int set_bit_no;  /* Will have only single set bit of xor2 */
  int i,n = size - 2;
  int x = 0, y = 0;
   /* Get the xor of all elements in arr[]. The xor will basically
     be xor of two odd occurring elements */
  for(i = 1; i < size; i++)
    xor2 = xor2 ^ arr[i];
   /* Get one set bit in the xor2. We get rightmost set bit
     in the following line as it is easy to get */
  set_bit_no = xor2 & ~(xor2-1);
   /* Now divide elements in two sets:
    1) The elements having the corresponding bit as 1.
    2) The elements having the corresponding bit as 0.   */
  for(i = 0; i < size; i++)
  {     /* XOR of first set is finally going to hold one odd
         occurring number x */
    if(arr[i] & set_bit_no)
      x = x ^ arr[i];
      /* XOR of second set is finally going to hold the other
       odd occurring number y */
    else
      y = y ^ arr[i];
  }
   printf("\n The two ODD elements are %d & %d ", x, y);
}
```

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

50 The Celebrity Problem

http://www.geeksforgeeks.org/the-celebrity-problem/

In a party of N people, only one person is known to everyone. Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "does A know B? ". Find the stranger (celebrity) in minimum number of questions.
We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function HaveAcquaintance(A, B) which returns true if A knows B, false otherwise. How can we solve the problem.
We measure the complexity in terms of calls made to HaveAcquaintance().

Method 1 (Graph)

We can model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair [i, j]. We have NC2 pairs. If celebrity is present in the party, we will have one sink node in the graph with outdegree of zero, and indegree of N-1. We can find the sink node in (N) time, but the overall complexity is O(N2) as we need to construct the graph first.

Method 2 (Recursion)

We can decompose the problem into combination of smaller instances. Say, if we know celebrity of N-1 persons, can we extend the solution to N? We have two possibilities, Celebrity(N-1) may know N, or N already knew Celebrity(N-1). In the former case, N will be celebrity if N doesn't know anyone else. In the later case we need to check that Celebrity(N-1) doesn't know N.

Solve the problem of smaller instance during divide step. On the way back, we find the celebrity (if present) from the smaller instance. During combine stage, check whether the returned celebrity is known to everyone and he doesn't know anyone. The recurrence of the recursive decomposition is,
T(N) = T(N-1) + O(N)
T(N) = O(N2). You may try writing pseudo code to check your recursion skills.

Method 3 (Using Stack)
The graph construction takes O(N2) time, it is similar to brute force search. In case of recursion, we reduce the problem instance by not more than one, and also combine step may examine M-1 persons (M – instance size).
We have following observation based on elimination technique (Refer Polya's How to Solve It book).
If A knows B, then A can't be celebrity. Discard A, and B may be celebrity.
If A doesn't know B, then B can't be celebrity. Discard B, and A may be celebrity.
Repeat above two steps till we left with only one person.
Ensure the remained person is celebrity. (Why do we need this step?)
We can use stack to verity celebrity.
Push all the celebrities into a stack.
Pop off top two persons from the stack, discard one person based on return status of HaveAcquaintance(A, B).
Push the remained person onto stack.
Repeat step 2 and 3 until only one person remains in the stack.
Check the remained person in stack doesn't have acquaintance with anyone else.
We will discard N elements utmost (Why?). If the celebrity is present in the party, we will call HaveAcquaintance() 3(N-1) times. Here is code using stack.

```
// Returns -1 if celebrity is not present.
// If present, returns id  (value from 0 to n-1).
int findCelebrity(int n)
{
    // Handle trivial case of size = 2

    stack<int> s;

    int C; // Celebrity

    // Push everybody to stack
    for (int i=0; i<n; i++)
        s.push(i);

    // Extract top 2
    int A = s.top();
    s.pop();
    int B = s.top();
    s.pop();

    // Find a potential celevrity
    while (s.size() > 1)
    {
        if (knows(A, B))
        {
            A = s.top();
            s.pop();
        }
        else
        {
            B = s.top();
            s.pop();
        }
    }

    // Potential candidate?
    C = s.top();
    s.pop();

    // Last candidate was not examined, it leads
    // one excess comparison (optimize)
    if (knows(C, B))
        C = B;

    if (knows(C, A))
        C = A;

    // Check if C is actually a celebrity or not
    for (int i = 0; i < n; i++)
    {
        // If any person doesn't know 'a' or 'a'
        // doesn't know any person, return -1
        if ( (i != C) &&
                (knows(C, i) || !knows(i, C)) )
            return -1;
    }

    return C;
}
```

Time Complexity O(N). Total comparisons 3(N-1). Try the above code for successful MATRIX {{0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}}.
Note: You may think that why do we need a new graph as we already have access to input matrix. Note that the matrix MATRIX used to help the hypothetical

function HaveAcquaintance(A, B), but never accessed via usual notation MATRIX[i, j]. We have access to the input only through the function HaveAcquiantance(A, B). Matrix is just a way to code the solution. We can assume the cost of hypothetical function as O(1).
If still not clear, assume that the function HaveAcquiantance accessing information stored in a set of linked lists arranged in levels. List node will have next and nextLevel pointers. Every level will have N nodes i.e. an N element list, next points to next node in the current level list and the nextLevel pointer in last node of every list will point to head of next level list. For example the linked list representation of above matrix looks like,

```
L0 0->0->1->0
          |
L1        0->0->1->0
                |
L2              0->0->1->0
                      |
L3                    0->0->1->0
```

The function HaveAcquanintance(i, j) will search in the list for j-th node in the i-th level. Out goal is to minimize calls to HaveAcquanintance function.

Method 4 (Using two Pointers)

The idea is to use two pointers, one from start and one from the end. Assume the start person is A, and the end person is B. If A knows B, then A must not be the celebrity. Else, B must not be the celebrity. We will find a celebrity candidate at the end of the loop. Go through each person again and check whether this is the celebrity. Below is C++ implementation.

```
// Returns id of celebrity
int findCelebrity(int n)
{
    // Initialize two pointers as two corners
    int a = 0;
    int b = n - 1;
     // Keep moving while the two pointers
    // don't become same.
    while (a < b)
    {
        if (knows(a, b))
            a++;
        else
            b--;
    }
     // Check if a is actually a celebrity or not
    for (int i = 0; i < n; i++)
    {
        // If any person doesn't know 'a' or 'a'
        // doesn't know any person, return -1
```

```
            if ( (i != a) &&
                    (knows(a, i) || !knows(i, a)) )
                return -1;
        }
        return a;
}
```

Exercises:
1. Write code to find celebrity. Don't use any data structures like graphs, stack, etc… you have access to N and HaveAcquaintance(int, int) only.
2. Implement the algorithm using Queues. What is your observation? Compare your solution with Finding Maximum and Minimum in an array and Tournament Tree. What are minimum number of comparisons do we need (optimal number of calls to HaveAcquaintance())?

51 Find a sorted subsequence of size 3 in linear time

http://www.geeksforgeeks.org/find-a-sorted-subsequence-of-size-3-in-linear-time/

Given an array of n integers, find the 3 elements such that a[i] < a[j] < a[k] and i < j < k in 0(n) time. If there are multiple such triplets, then print any one of them.
Examples:
Input: arr[] = {12, 11, 10, 5, 6, 2, 30}
Output: 5, 6, 30
Input: arr[] = {1, 2, 3, 4}
Output: 1, 2, 3 OR 1, 2, 4 OR 2, 3, 4
Input: arr[] = {4, 3, 2, 1}
Output: No such triplet
Source: Amazon Interview Question
Hint: Use Auxiliary Space

Solution:
1) Create an auxiliary array smaller[0..n-1]. smaller[i] should store the index of a number which is smaller than arr[i] and is on left side of arr[i]. smaller[i] should contain -1 if there is no such element.
2) Create another auxiliary array greater[0..n-1]. greater[i] should store the index of a number which is greater than arr[i] and is on right side of arr[i]. greater[i] should contain -1 if there is no such element.
3) Finally traverse both smaller[] and greater[] and find the index i for which both smaller[i] and greater[i] are not -1.

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

Exercise:

1. Find a subsequence of size 3 such that arr[i] < arr[j] > arr[k].
2. Find a sorted subsequence of size 4 in linear time

52 Largest subarray with equal number of 0s and 1s

http://www.geeksforgeeks.org/largest-subarray-with-equal-number-of-0s-and-1s/

Given an array containing only 0s and 1s, find the largest subarray which contain equal no of 0s and 1s. Expected time complexity is O(n).
Examples:
Input: arr[] = {1, 0, 1, 1, 1, 0, 0}
Output: 1 to 6 (Starting and Ending indexes of output subarray)

Input: arr[] = {1, 1, 1, 1}
Output: No such subarray

Input: arr[] = {0, 0, 1, 1, 0}
Output: 0 to 3 Or 1 to 4

Method 1 (Simple)
A simple method is to use two nested loops. The outer loop picks a starting point i. The inner loop considers all subarrays starting from i. If size of a subarray is greater than maximum size so far, then update the maximum size.
In the below code, 0s are considered as -1 and sum of all values from i to j is calculated. If sum becomes 0, then size of this subarray is compared with largest size so far.
Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Tricky)
Following is a solution that uses O(n) extra space and solves the problem in O(n) time complexity.
Let input array be arr[] of size n and maxsize be the size of output subarray.
1) Consider all 0 values as -1. The problem now reduces to find out the maximum length subarray with sum = 0.
2) Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i]. This can be done in O(n) time.
3) There are two cases, the output subarray may start from 0th index or may start from some other index. We will return the max of the values obtained by two cases.
4) To find the maximum length subarray starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.
5) Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we can create a hash table with size = max-min+1 where min

is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. The idea is to hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because there can be these many different possible values in sumleft[]. Initialize all values in hash as -1
6) To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.
7) To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

```
// Returns largest subarray with equal number of 0s and 1s
int maxLen(int arr[], int n)
{
    // Creates an empty hashMap hM
    unordered_map<int, int> hM;

    int sum = 0;      // Initialize sum of elements
    int max_len = 0; // Initialize result
    int ending_index = -1;

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

    // Traverse through the given array
    for (int i = 0; i < n; i++)
    {
        // Add current element to sum
        sum += arr[i];

        // To handle sum=0 at last index
        if (sum == 0)
        {
            max_len = i + 1;
            ending_index = i;
        }
         // If this sum is seen before, then update max_len
        // if required
        if (hM.find(sum + n) != hM.end())
        {
            if (max_len < i - hM[sum + n])
            {
                max_len = i - hM[sum + n];
```

```
                    ending_index = i;
                }
            }
            else // Else put this sum in hash table
                hM[sum + n] = i;
        }
         for (int i = 0; i < n; i++)
            arr[i] = (arr[i] == -1)? 0: 1;

    printf("%d to %d\n", ending_index-max_len+1, ending_index);
     return max_len;
}
```

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

53 Maximum Product Subarray

http://www.geeksforgeeks.org/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, -3, 0, -2, -40}
Output: 80 // The subarray is {-2, -40}

The following solution assumes that the given input array always has a positive output. The solution works for all cases mentioned above. It doesn't work for arrays like {0, 0, -20, 0}, {0, 0, 0}.. etc. The solution can be easily modified to handle this case.
It is similar to Largest Sum Contiguous Subarray problem. The only thing to note here is, maximum product can also be obtained by minimum (negative) product ending with the previous element multiplied by this element. For example, in array {12, 2, -3, -5, -6, -2}, when we are at element -2, the maximum product is multiplication of, minimum product ending with -6 and -2

```
/* Returns the product of max product subarray.
   Assumes that the given array always has a subarray
   with product more than 1 */
int maxSubarrayProduct(int arr[], int n)
{
    // max positive product ending at the current position
```

```
int max_ending_here = 1;

// min negative product ending at the current position
int min_ending_here = 1;

// Initialize overall max product
int max_so_far = 1;
 /* Traverse through the array. Following values are
   maintained after the i'th iteration:
   max_ending_here is always 1 or some positive product
                   ending with arr[i]
   min_ending_here is always 1 or some negative product
                   ending with arr[i] */
for (int i = 0; i < n; i++)
{
    /* If this element is positive, update max_ending_here.
       Update min_ending_here only if min_ending_here is
       negative */
    if (arr[i] > 0)
    {
        max_ending_here = max_ending_here*arr[i];
        min_ending_here = min (min_ending_here * arr[i], 1);
    }
     /* If this element is 0, then the maximum product
       cannot end here, make both max_ending_here and
       min_ending_here 0
       Assumption: Output is alway greater than or equal to 1. */
    else if (arr[i] == 0)
    {
        max_ending_here = 1;
        min_ending_here = 1;
    }
     /* If element is negative. This is tricky
       max_ending_here can either be 1 or positive.
       min_ending_here can either be 1 or negative.
       next min_ending_here will always be prev.
       max_ending_here * arr[i] next max_ending_here
       will be 1 if prev min_ending_here is 1, otherwise
       next max_ending_here will be prev min_ending_here *
       arr[i] */
    else
    {
        int temp = max_ending_here;
        max_ending_here = max (min_ending_here * arr[i], 1);
        min_ending_here = temp * arr[i];
    }
     // update max_so_far, if needed
```

```
        if  (max_so_far <  max_ending_here)
          max_so_far  =  max_ending_here;
    }
     return max_so_far;
}
```

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

54 Find a pair with the given difference

http://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/

Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.
Examples:
Input: arr[] = {5, 20, 3, 2, 50, 80}, n = 78
Output: Pair Found: (2, 80)

Input: arr[] = {90, 70, 20, 80, 50}, n = 45
Output: No Such Pair
Source: find pair

The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n. Time complexity of this method is O(n^2).
We can use sorting and Binary Search to improve time complexity to O(nLogn). The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element arr[i], binary search for arr[i] + n in arr[i+1..n-1]. If the element is found, return the pair.
Both first and second steps take O(nLogn). So overall **complexity is O(nLogn).**

The second step of the above algorithm can be improved to O(n). The first step remain same. The idea for second step is take two index variables i and j, initialize them as 0 and 1 respectively. Now run a linear loop. If arr[j] – arr[i] is smaller than n, we need to look for greater arr[j], so increment j. If arr[j] – arr[i] is greater than n, we need to look for greater arr[i], so increment i.
The following code is only for the second step of the algorithm, it assumes that the array is already sorted.

```
// The function assumes that the array is sorted
bool findPair(int arr[], int size, int n)
{     // Initialize positions of two elements
    int i = 0;
    int j = 1;
     // Search for a pair
    while (i<size && j<size)
    {
        if (i != j && arr[j]-arr[i] == n)
        {
            printf("Pair Found: (%d, %d)", arr[i], arr[j]);
            return true;
        }
        else if (arr[j]-arr[i] < n)
            j++;
        else
            i++;
    }
     printf("No such pair");
    return false;
}
```

Hashing can also be used to solve this problem. Create an empty hash table HT. Traverse the array, use array elements as hash keys and enter them in HT. Traverse the array again look for value n + arr[i] in HT.

55 Replace every element with the greatest element on right side

http://www.geeksforgeeks.org/replace-every-element-with-the-greatest-on-right-side/

Given an array of integers, replace every element with the next greatest element (greatest element on the right side) in the array. Since there is no element next to the last element, replace it with -1. For example, if the array is {16, 17, 4, 3, 5, 2}, then it should be modified to {17, 5, 5, 5, 2, -1}.

A **naive method** is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the greatest element present after the picked element. Finally the outer loop will replace the picked element with the greatest element found by inner loop. The **time complexity** of this method will be O(n*n).

A **tricky method** is to replace all elements using one traversal of the array. The idea is to start from the rightmost element, move to the left side one by one, and keep track of the maximum element. Replace every element with the maximum element.
Time Complexity: O(n) where n is the number of elements in array.

56 Find four elements that sum to a given value | Set 1 (n^3 solution)

http://www.geeksforgeeks.org/find-four-numbers-with-sum-equal-to-given-sum/

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X.
For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X = 23, then your function should print “3 5 7 8” (3 + 5 + 7 + 8 = 23).

A **Naive Solution** is to generate all possible quadruples and compare the sum of every quadruple with X. The following code implements this simple method using four nested loops
Time Complexity: O(n^4)

The time complexity can be improved to O(n^3) with the use of sorting as a preprocessing step, and then using method 1 of this post to reduce a loop.
Following are the detailed steps.
1) Sort the input array.
2) Fix the first element as A[i] where i is from 0 to n–3. After fixing the first element of quadruple, fix the second element as A[j] where j varies from i+1 to n-2. Find remaining two elements in O(n) time, using the method 1 of this post
Following is the implementation of O(n^3) solution.
Time Complexity: O(n^3)

This problem can also be solved in O(n^2Logn) complexity. We will soon be publishing the O(n^2Logn) solution as a separate post.

57 Find four elements that sum to a given value | Set 2 (O(n^2Logn) Solution)

http://www.geeksforgeeks.org/find-four-elements-that-sum-to-a-given-value-set-2/

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X.
For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X = 23, then your function should print “3 5 7 8” (3 + 5 + 7 + 8 = 23).
We have discussed a O(n^3) algorithm in the previous post on this topic. The problem can be solved in O(n^2Logn) time with the help of auxiliary space.
Let the input array be A[].
1) Create an auxiliary array aux[] and store sum of all possible pairs in aux[]. The size of aux[] will be n*(n-1)/2 where n is the size of A[].
2) Sort the auxiliary array aux[].

3) Now the problem reduces to find two elements in aux[] with sum equal to X. We can use method 1 of this post to find the two elements efficiently. There is following important point to note though. An element of aux[] represents a pair from A[]. While picking two elements from aux[], we must check whether the two elements have an element of A[] in common. For example, if first element sum of A[1] and A[2], and second element is sum of A[2] and A[4], then these two elements of aux[] don't represent four distinct elements of input array A[].

```
void findFourElements (int arr[], int n, int X)
{     int i, j;
     // Create an auxiliary array to store all pair sums
    int size = (n*(n-1))/2;
    struct pairSum aux[size];
    int k = 0;
    for (i = 0; i < n-1; i++)
    {
        for (j = i+1; j < n; j++)
        {
            aux[k].sum = arr[i] + arr[j];
            aux[k].first = i;
            aux[k].sec = j;
            k++;
        }
    }
     // Sort the aux[] array using library function for sorting
    qsort (aux, size, sizeof(aux[0]), compare);
    i = 0;
    j = size-1;
    while (i < size && j >=0 )
    {   if ((aux[i].sum + aux[j].sum == X) && noCommon(aux[i], aux[j]))
        {
        printf ("%d, %d, %d, %d\n", arr[aux[i].first], arr[aux[i].sec],
                                     arr[aux[j].first], arr[aux[j].sec]);
            return;
        }
        else if (aux[i].sum + aux[j].sum < X)
            i++;
        else
            j--;
    }}
```

Please note that the above code prints only one quadruple. If we remove the return statement and add statements "i++; j–;", then it prints same quadruple five times. The code can modified to print all quadruples only once. It has been kept this way to keep it simple.
Time complexity: The step 1 takes O(n^2) time. The second step is sorting an array of size O(n^2). Sorting can be done in O(n^2Logn) time using merge sort or heap sort or any other O(nLogn) algorithm. The third step takes O(n^2) time. So overall complexity is O(n^2Logn).
Auxiliary Space: O(n^2). The big size of auxiliary array can be a concern in this method.

58 Sort a nearly sorted (or K sorted) array

http://www.geeksforgeeks.org/nearly-sorted-algorithm/

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.
We can **use Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.
The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall **complexity** will be O(nk)

We can sort such arrays more efficiently with the help of Heap data structure. Following is the detailed process that uses Heap.
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time
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)

```
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++)//i < n condition is needed when k>n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);
     // i is index for remaining elements in arr[] and ti
    // is target index of for cuurent minimum element in
    // Min Heapm 'hp'.
    for(int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[i]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }}
```

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

59 Maximum circular subarray sum

http://www.geeksforgeeks.org/maximum-contiguous-circular-sum/

Given n numbers (both +ve and -ve), arranged in a circle, fnd the maximum sum of consecutive number.
Examples:
Input: a[] = {8, -8, 9, -9, 10, -11, 12}
Output: 22 (12 + 8 - 8 + 9 - 9 + 10)

Input: a[] = {10, -3, -4, 7, 6, 5, -4, -1}
Output: 23 (7 + 6 + 5 - 4 -1 + 10)

Input: a[] = {-1, 40, -14, 7, 6, 5, -4, -1}
Output: 52 (7 + 6 + 5 - 4 - 1 - 1 + 40)

There can be two cases for the maximum sum:
Case 1: The elements that contribute to the maximum sum are arranged such that no wrapping is there. Examples: {-10, 2, -1, 5}, {-2, 4, -1, 4, -1}. In this case, Kadane's algorithm will produce the result.
Case 2: The elements which contribute to the maximum sum are arranged such that wrapping is there. Examples: {10, -12, 11}, {12, -5, 4, -8, 11}. In this case, we change wrapping to non-wrapping. Let us see how. Wrapping of contributing elements implies non wrapping of non contributing elements, so find out the sum of non contributing elements and subtract this sum from the total sum. To find out the sum of non contributing, invert sign of each element and then run Kadane's algorithm.
Our array is like a ring and we have to eliminate the maximum continuous negative that implies maximum continuous positive in the inverted arrays.
Finally we compare the sum obtained by both cases, and return the maximum of the two sums.

```
int maxCircularSum(int a[], int n)
{
   // Case 1: get the maximum sum using standard kadane'
   // s algorithm
   int max_kadane = kadane(a, n);
    // Case 2: Now find the maximum sum that includes
   // corner elements.
```

```
    int max_wrap = 0, i;
    for (i=0; i<n; i++)
    {     max_wrap += a[i]; // Calculate array-sum
          a[i] = -a[i];  // invert the array (change sign)
    }
     // max sum with corner elements will be:
    // array-sum - (-max subarray sum of inverted array)
    max_wrap = max_wrap + kadane(a, n);
     // The maximum circular sum will be maximum of two sums
    return (max_wrap > max_kadane)? max_wrap: max_kadane;
}
```

Time Complexity: O(n) where n is the number of elements in input array.
Note that the above algorithm doesn't work if all numbers are negative e.g., {-1, -2, -3}. It returns 0 in this case.

This case can be handled by adding a pre-check to see if all the numbers are negative before running the above algorithm.

60 Median of two sorted arrays of different sizes

http://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/
The approach discussed in this post is similar to method 2 of equal size post. The basic idea is same, we find the median of two arrays and compare the medians to discard almost half of the elements in both arrays. Since the number of elements may differ here, there are many base cases that need to be handled separately. Before we proceed to complete solution, let us first talk about all base cases.

Let the two arrays be A[N] and B[M]. In the following explanation, it is assumed that N is smaller than or equal to M.
Base cases:
The smaller array has only one element
Case 0: N = 0, M = 2
Case 1: N = 1, M = 1.
Case 2: N = 1, M is odd
Case 3: N = 1, M is even
The smaller array has only two elements
Case 4: N = 2, M = 2
Case 5: N = 2, M is odd
Case 6: N = 2, M is even
Case 0: There are no elements in first array, return median of second array. If second array is also empty, return -1.
Case 1: There is only one element in both arrays, so output the average of A[0] and B[0].

61 Shuffle a given array

http://www.geeksforgeeks.org/shuffle-a-given-array/

Given an array, write a program to generate a random permutation of array elements. This question is also asked as “shuffle a deck of cards” or “randomize a given array”.

Let the given array be arr[]. A simple solution is to create an auxiliary array temp[] which is initially a copy of arr[]. Randomly select an element from temp[], copy the randomly selected element to arr[0] and remove the selected element from temp[]. Repeat the same process n times and keep copying elements to arr[1], arr[2], … . The time complexity of this solution will be O(n^2).

Fisher–Yates shuffle Algorithm works in O(n) time complexity. The assumption here is, we are given a function rand() that generates random number in O(1) time.
The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to n-2 (size reduced by 1), and repeat the process till we hit the first element.
Following is the detailed algorithm

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n - 1 downto 1 do
       j = random integer with 0 <= j <= i
       exchange a[j] and a[i]
```

```
// A function to generate a random permutation of arr[]
void randomize ( int arr[], int n )
{
    // Use a different seed value so that we don't get same
    // result each time we run this program
    srand ( time(NULL) );

    // Start from the last element and swap one by one. We don't
    // need to run for the first element that's why i > 0
    for (int i = n-1; i > 0; i--)
    {
        // Pick a random index from 0 to i
        int j = rand() % (i+1);
         // Swap arr[i] with the element at random index
        swap(&arr[i], &arr[j]);
   }}
```

The above function assumes that rand() generates a random number.
Time Complexity: O(n), assuming that the function rand() takes O(1) time.

How does this work?
The probability that ith element (including the last one) goes to last position is 1/n, because we randomly pick an element in first iteration.
The probability that ith element goes to second last position can be proved to be 1/n by dividing it in two cases.
Case 1: i = n-1 (index of last element):
The probability of last element going to second last position is = (probability that last element doesn't stay at its original position) x (probability that the index picked in previous step is picked again so that the last element is swapped)
So the probability = ((n-1)/n) x (1/(n-1)) = 1/n
Case 2: 0 < i < n-1 (index of non-last):
The probability of ith element going to second position = (probability that ith element is not picked in previous iteration) x (probability that ith element is picked in this iteration)
So the probability = ((n-1)/n) x (1/(n-1)) = 1/n
We can easily generalize above proof for any other position.

62 Count the number of possible triangles

http://www.geeksforgeeks.org/find-number-of-triangles-possible/
Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).
For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.
As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Method 1 (Brute force)
The brute force method is to run three loops and keep track of the number of triangles possible so far. The three loops select three different values from array, the innermost loop checks for the triangle property (the sum of any two sides must be greater than the value of third side).
Time Complexity: O(N^3) where N is the size of input array.
Method 2 (Tricky and Efficient)
Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)
i) a + b > c
ii) b + c > a
iii) a + c > b

Following are steps to count triangle.

1. Sort the array in non-decreasing order.
2. Initialize two pointers ‘i’ and ‘j’ to first and second elements respectively, and initialize count of triangles as 0.
3. Fix ‘i’ and ‘j’ and find the rightmost index ‘k’ (or largest ‘arr[k]’) such that ‘arr[i] + arr[j] > arr[k]’. The number of triangles that can be formed with ‘arr[i]’ and ‘arr[j]’ as two sides is ‘k – j’. Add ‘k – j’ to count of triangles.

Let us consider ‘arr[i]’ as ‘a’, ‘arr[j]’ as b and all elements between ‘arr[j+1]’ and ‘arr[k]’ as ‘c’. The above mentioned conditions (ii) and (iii) are satisfied because ‘arr[i] < arr[j] < arr[k]'. And we check for condition (i) when we pick 'k'4. Increment ‘j’ to fix the second element again.
Note that in step 3, we can use the previous value of ‘k’. The reason is simple, if we know that the value of ‘arr[i] + arr[j-1]’ is greater than ‘arr[k]’, then we can say ‘arr[i] + arr[j]’ will also be greater than ‘arr[k]’, because the array is sorted in increasing order.
5. If ‘j’ has reached end, then increment ‘i’. Initialize ‘j’ as ‘i + 1’, ‘k’ as ‘i+2’ and repeat the steps 3 and 4.
Following is implementation of the above approach

```
// Function to count all possible triangles with arr[]
// elements
int findNumberOfTriangles(int arr[], int n)
{
    // Sort the array elements in non-decreasing order
    qsort(arr, n, sizeof( arr[0] ), comp);

    // Initialize count of triangles
    int count = 0;

    // Fix the first element.  We need to run till n-3
    // as the other two elements are selected from
    // arr[i+1...n-1]

    for (int i = 0; i < n-2; ++i)
    {
        // Initialize index of the rightmost third
        // element
        int k = i+2;
         // Fix the second element
        for (int j = i+1; j < n; ++j)
        {
            // Find the rightmost element which is
            // smaller than the sum of two fixed elements
            // The important thing to note here is, we
            // use the previous  value of k. If value of
            // arr[i] + arr[j-1] was greater than arr[k],
            // then arr[i] + arr[j] must be greater than k,
            // because the array is sorted.

           while (k < n && arr[i] + arr[j] > arr[k])
               ++k;

            // Total number of possible triangles that can
            // be formed with the two fixed elements is
            //  k - j - 1.  The two fixed elements are arr[i]
            // and arr[j].  All elements between arr[j+1]/ to
            // arr[k-1] can form a triangle with arr[i] and arr[j].
            // One is subtracted from k because k is incremented
            // one extra in above while loop.
            // k will always be greater than j. If j becomes equal
            // to k, then above loop will increment k, because arr[k]
            //  + arr[i] is always greater than arr[k]

           count += k - j - 1;
        }
    }

    return count;
}
```

Time Complexity: O(n^2). The time complexity looks more because of 3 nested loops. If we take a closer look at the algorithm, we observe that k is initialized only once in the outermost loop. The innermost loop executes at most O(n) time for every iteration of outer most loop, because k starts from i+2 and goes upto n for all values of j. Therefore, the time complexity is O(n^2).

63 Iterative Quick Sort

http://www.geeksforgeeks.org/iterative-quick-sort/
Following is a typical recursive implementation of Quick Sort that uses last element as pivot.

```
/* A[] --> Array to be sorted,
  l  --> Starting index,
  h  --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        /* Partitioning index */
        int p = partition(A, l, h);
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

The above implementation can be optimized in many ways
1) The above implementation uses last index as pivot. This causes worst-case behavior on already sorted arrays, which is a commonly occurring case. The problem can be solved by choosing either a random index for the pivot, or choosing the middle index of the partition or choosing the median of the first, middle and last element of the partition for the pivot. (See this for details)
2) To reduce the recursion depth, recur first for the smaller half of the array, and use a tail call to recurse into the other.
3) Insertion sort works better for small subarrays. Insertion sort can be used for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). For example, this library implementation of qsort uses insertion sort below size 7.
Despite above optimizations, the function remains recursive and uses function call stack to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution.
The above function can be easily converted to iterative version with the help of an auxiliary stack. Following is an iterative implementation of the above recursive code

```
/* A[] --> Array to be sorted,
   l  --> Starting index,
   h  --> Ending index */
void quickSortIterative (int arr[], int l, int h)
{
    // Create an auxiliary stack
    int stack[ h - l + 1 ];

    // initialize top of stack
    int top = -1;

    // push initial values of l and h to stack
    stack[ ++top ] = l;
    stack[ ++top ] = h;

    // Keep popping from stack while is not empty
    while ( top >= 0 )
    {
        // Pop h and l
        h = stack[ top-- ];
        l = stack[ top-- ];

        // Set pivot element at its correct position
        // in sorted array
        int p = partition( arr, l, h );

        // If there are elements on left side of pivot,
        // then push left side to stack
        if ( p-1 > l )
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if ( p+1 < h )
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }
}
```

The above mentioned optimizations for recursive quick sort can also be applied to iterative version.
1) Partition process is same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to iterative version.
2) To reduce the stack size, first push the indexes of smaller half.
3) Use insertion sort when the size reduces below a experimentally calculated threshold.

64 Find the first circular tour that visits all petrol pumps

http://www.geeksforgeeks.org/find-a-tour-that-visits-all-stations/

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.
1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.
Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be "start = 1" (index of 2nd petrol pump).
A Simple Solution is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point.
The worst case time complexity of this solution is O(n^2).

We can use a Queue to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.
Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

```
// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end =  1;
     int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
      And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If curremt amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
```

```
                start = (start + 1)%n;

                // If 0 is being considered as start again, then there is no
                // possible solution
                if (start == 0)
                   return -1;
            }
             // Add a petrol pump to current tour
            curr_petrol += arr[end].petrol - arr[end].distance;
             end = (end + 1)%n;
        }
         // Return starting point
        return start;
}
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the **time complexity is O(n).**
Auxiliary Space: O(1)

65 Arrange given numbers to form the biggest number

http://www.geeksforgeeks.org/given-an-array-of-numbers-arrange-the-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.

A simple solution that comes to our mind is to sort all numbers in descending order, but simply sorting doesn't work. For example, 548 is greater than 60, but in output 60 comes before 548. As a second example, 98 is greater than 9, but 9 comes before 98 in output.

So how do we go about it? The idea is to 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.
Following is C++ implementation of the above approach. To keep the code simple, numbers are considered as strings, and vector is used instead of normal array.

```
// A comparison function which is used by sort() in printLargest()
int myCompare(string X, string Y)
{
    // first append Y at the end of X
    string XY = X.append(Y);
     // then append X at the end of Y
    string YX = Y.append(X);
     // Now see which of the two formed numbers is greater
    return XY.compare(YX) > 0 ? 1: 0;
}
 // The main function that prints the arrangement with the largest value.
// The function accepts a vector of strings
void printLargest(vector<string> arr)
{
    // Sort the numbers using library sort funtion. The function uses
    // our comparison function myCompare() to compare two strings.
    // See http://www.cplusplus.com/reference/algorithm/sort/ for details
    sort(arr.begin(), arr.end(), myCompare);
     for (int i=0; i < arr.size() ; i++ )
        cout << arr[i];
}
```

66 Pancake sorting

http://www.geeksforgeeks.org/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
Unlike a traditional sorting algorithm, which attempts to sort with the fewest comparisons possible, the goal is to sort the sequence in as few reversals as possible.
The idea is to do something similar to 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)

```
// The main function that sorts given array using flip
// operations
int pancakeSort(int *arr, int n)
{
    // Start from the complete array and one by one reduce
    // current size by one
    for (int curr_size = n; curr_size > 1; --curr_size)
    {
        // Find index of the maximum element in
        // arr[0..curr_size-1]
        int mi = findMax(arr, curr_size);

        // Move the maximum element to end of current array
        // if it's not already at the end
        if (mi != curr_size-1)
        {
            // To move at the end, first move maximum number
            // to beginning
            flip(arr, mi);

            // Now move the maximum number to end by reversing
            // current array
            flip(arr, curr_size-1);
        }
    }
}
```

Total O(n) flip operations are performed in above code. **The overall time complexity is O(n^2).**

67 A Pancake Sorting Problem

http://www.geeksforgeeks.org/a-pancake-sorting-question/

We have discussed Pancake Sorting in the previous post. Following is a problem based on 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
Imagine a hypothetical machine where flip(i) always takes O(1) time. Write an efficient program for sorting a given array in O(nLogn) time on the given machine.

If we apply the same algorithm here, the time taken will be O(n^2) because the algorithm calls findMax() in a loop and find findMax() takes O(n) time even on this hypothetical machine.
We can use insertion sort that uses binary search. The idea is to run a loop from second element to last element (from i = 1 to n-1), and one by one insert arr[i] in arr[0..i-1] (like standard insertion sort algorithm). When we insert an element arr[i], we can use binary

search to find position of arr[i] in O(Logi) time. Once we have the position, we can use some flip operations to put arr[i] at its new place. Following are abstract steps.

```
// Standard Insertion Sort Loop that starts from second element
for (i=1; i < n; i++) ----> O(n)
{
  int key = arr[i];

  // Find index of ceiling of arr[i] in arr[0..i-1] using binary search
  j = celiSearch(arr, key, 0, i-1); ----> O(logn) (See this)

  // Apply some flip operations to put arr[i] at correct place
}
```

Since flip operation takes O(1) on given hypothetical machine, total running time of above algorithm is O(nlogn).

Let us see how does the above algorithm work. ceilSearch() actually returns the index of the smallest element which is greater than arr[i] in arr[0..i-1]. If there is no such element, it returns -1. Let the returned value be j. If j is -1, then we don't need to do anything as arr[i] is already the greatest element among arr[0..i]. Otherwise we need to put arr[i] just before arr[j].
So how to apply flip operations to put arr[i] just before arr[j] using values of i and j. Let us take an example to understand this. Let i be 6 and current array be {12, 15, 18, 30, 35, 40, 20, 6, 90, 80}. To put 20 at its new place, the array should be changed to {12, 15, 18, 20, 30, 35, 40, 6, 90, 80}. We apply following steps to put 20 at its new place.
1) Find j using ceilSearch (In the above example j is 3).
2) flip(arr, j-1) (array becomes {18, 15, 12, 30, 35, 40, 20, 6, 90, 80})
3) flip(arr, i-1); (array becomes {40, 35, 30, 12, 15, 18, 20, 6, 90, 80})
4) flip(arr, i); (array becomes {20, 18, 15, 12, 30, 35, 40, 6, 90, 80})
5) flip(arr, j); (array becomes {12, 15, 18, 20, 30, 35, 40, 6, 90, 80})

```
/* Function to sort an array using insertion sort, binary search and flip
*/
void insertionSort(int arr[], int size)
{
    int i, j;

    // Start from the second element and one by one insert arr[i]
    // in already sorted arr[0..i-1]
    for(i = 1; i < size; i++)
    {
        // Find the smallest element in arr[0..i-1] which is also greater
than
        // or equal to arr[i]
        int j = ceilSearch(arr, 0, i-1, arr[i]);

        // Check if there was no element greater than arr[i]
        if (j != -1)
        {
            // Put arr[i] before arr[j] using following four flip 
operations
            flip(arr, j-1);
            flip(arr, i-1);
            flip(arr, i);
            flip(arr, j);
        }}}
```

68 Divide and Conquer | Set 3 (Maximum Subarray Sum)

http://www.geeksforgeeks.org/divide-and-conquer-maximum-sum-subarray/

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.
For example, if the given array is {-2, -5, 6, -2, -3, 1, 5, -6}, then the maximum subarray sum is 7 (see highlighted elements).

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. **The time complexity of the Naive method is O(n^2).**

Using Divide and Conquer approach,
we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1) Divide the given array in two halves
2) Return the maximum of following three
....a) Maximum subarray sum in left half (Make a recursive call)
....b) Maximum subarray sum in right half (Make a recursive call)
....c) Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.

```
// Returns sum of maxium sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
   // Base Case: Only one element
   if (l == h)
     return arr[l];

   // Find middle point
   int m = (l + h)/2;

   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpoint
*/
   return max(maxSubArraySum(arr, l, m),
              maxSubArraySum(arr, m+1, h),
              maxCrossingSum(arr, l, m, h));
}
```

Time Complexity: maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \Theta(n)$
The above recurrence is similar to Merge Sort and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n\text{Log}n)$.
The Kadane's Algorithm for this problem takes O(n) time. Therefore the Kadane's algorithm is better than the Divide and Conquer approach, but this problem can be considered as a good example to show power of Divide and Conquer. The above simple approach where we divide the array in two halves, reduces the time complexity from O(n^2) to O(nLogn).

69 Counting Sort

http://www.geeksforgeeks.org/counting-sort/
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.

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.

70 Merge Overlapping Intervals

http://www.geeksforgeeks.org/merging-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}

A simple approach is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from list and merge the other into the first interval. Repeat the same steps for remaining intervals after first. This approach cannot be implemented in better than O(n^2) time.

An efficient approach is to first sort the intervals according to starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if interval[i] doesn't overlap with interval[i-1], then interval[i+1] cannot overlap with interval[i-1] because starting time of interval[i+1] must be greater than or equal to interval[i]. Following is the detailed step by step algorithm.

```
1. Sort the intervals based on increasing order of
   starting time.
2. Push the first interval on to a stack.
3. For each interval do the following
  a. If the current interval does not overlap with the stack
     top, push it.
  b. If the current interval overlaps with stack top and ending
     time of current interval is more than that of stack top,
     update stack top with the ending  time of current interval.
4. At the end stack contains the merged intervals.
```

Time complexity of the method is O(nLogn) which is for sorting. Once the array of intervals is sorted, merging takes linear time.
A O(n Log n) and O(1) Extra Space Solution

The above solution requires O(n) extra space for stack.
We can avoid use of extra space by doing merge operations in-place. Below are detailed steps.

```
1) Sort all intervals in decreasing order of start time.
2) Traverse sorted intervals starting from first interval,
  do following for every interval.
    a) If current interval is not first interval and it
      overlaps with previous interval, then merge it with
      previous interval. Keep doing it while the interval
      overlaps with the previous one.
    b) Else add current interval to output list of intervals.
```

Note that if intervals are sorted by decreasing order of start times, we can quickly check if intervals overlap or not by comparing start time of previous interval with end time of current interval.

```
void mergeIntervals(Interval arr[], int n)
{
    // Sort Intervals in decreasing order of
    // start time
    sort(arr, arr+n, mycomp);

    int index = 0; // Stores index of last element
    // in output array (modified arr[])

    // Traverse all input Intervals
    for (int i=0; i<n; i++)
    {
        // If this is not first Interval and overlaps
        // with the previous one
        if (index != 0 && arr[index-1].s <= arr[i].e)
        {
            while (index != 0 && arr[index-1].s <= arr[i].e)
            {
                // Merge previous and current Intervals
                arr[index-1].e = max(arr[index-1].e, arr[i].e);
                arr[index-1].s = min(arr[index-1].s, arr[i].s);
                index--;
            }
        }
        else // Doesn't overlap with previous, add to
            // solution
            arr[index] = arr[i];

        index++;
    }

    // Now arr[0..index-1] stores the merged Intervals
    cout << "\n The Merged Intervals are: ";
    for (int i = 0; i < index; i++)
        cout << "[" << arr[i].s << ", " << arr[i].e << "] ";
}
```

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

http://www.geeksforgeeks.org/find-the-maximum-repeating-number-in-ok-time/

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.

The naive approach is to run two loops, the outer loop picks an element one by one, the inner loop counts number of occurrences of the picked element. Finally return the element with maximum count. **Time complexity of this approach is O(n^2).**

A better approach is to create a count array of size k and initialize all elements of count[] as 0. Iterate through all elements of input array, and for every element arr[i], increment count[arr[i]]. Finally, iterate through count[] and return the index with maximum value. This approach takes O(n) time, but requires O(k) space.

Following is the O(n) time and O(1) extra space approach. 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.

How does the above algorithm work? Since we use arr[i]%k as index and add value k at the index arr[i]%k, the index which is equal to maximum repeating element will have the maximum value in the end. Note that k is added maximum number of times at the index equal to maximum repeating element and all array elements are smaller than k.

Exercise:

The above solution prints only one repeating element and doesn't work if we want to print all maximum repeating elements. For example, if the input array is {2, 3, 2, 3}, the above solution will print only 3. What if we need to print both of 2 and 3 as both of them occur maximum number of times. Write a O(n) time and O(1) extra space function that prints all maximum repeating elements. (Hint: We can use maximum quotient arr[i]/n instead of maximum value in step 2).

Note that the above solutions may cause overflow if adding k repeatedly makes the value more than INT_MAX.

72 Find a peak element

http://www.geeksforgeeks.org/find-a-peak-in-a-given-array/

Given an array of integers. Find a peak element in it. An array element is peak if it is NOT smaller than its neighbors. For corner elements, we need to consider only one neighbor. For example, for input array {5, 10, 20, 15}, 20 is the only peak element. For input array {10, 20, 15, 2, 23, 90, 67}, there are two peak elements: 20 and 90. Note that we need to return any one peak element.

Following corner cases give better idea about the problem.
1) If input array is sorted in strictly increasing order, the last element is always a peak element. For example, 50 is peak element in {10, 20, 30, 40, 50}.
2) If input array is sorted in strictly decreasing order, the first element is always a peak element. 100 is the peak element in {100, 80, 60, 50, 20}.
3) If all elements of input array are same, every element is a peak element.
It is clear from above examples that there is always a peak element in input array in any input array.

A simple solution is to do a linear scan of array and as soon as we find a peak element, we return it. The worst case **time complexity of this method would be O(n).**

Can we find a peak element in worst time complexity better than O(n)?
We can use **Divide and Conquer to find a peak in O(Logn) time**. The idea is Binary Search based, we compare middle element with its neighbors. If middle element is not smaller than any of its neighbors, then we return it. If the middle element is smaller than the its left neighbor, then there is always a peak in left half (Why? take few examples). If the middle element is smaller than the its right neighbor, then there is always a peak in right half (due to same reason as left half). Following are C and Java implementations of this approach.

```
// A binary search based function that returns index of a peak element
int findPeakUtil(int arr[], int low, int high, int n)
{   // Find index of middle element
    int mid = low + (high - low)/2;  /* (low + high)/2 */

    // Compare middle element with its neighbours (if neighbours exist)
    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&
            (mid == n-1 || arr[mid+1] <= arr[mid]))
        return mid;

    // If middle element is not peak and its left neighbour is greater
    // than it, then left half must have a peak element
    else if (mid > 0 && arr[mid-1] > arr[mid])
        return findPeakUtil(arr, low, (mid -1), n);
     // If middle element is not peak and its right neighbour is greater
    // than it, then right half must have a peak element
    else return findPeakUtil(arr, (mid + 1), high, n);
}
 // A wrapper over recursive function findPeakUtil()
int findPeak(int arr[], int n)
{
    return findPeakUtil(arr, 0, n-1, n);
}
```

Time Complexity: O(Logn) where n is number of elements in input array.

73 Print all possible combinations of r elements in a given array of size n

http://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/

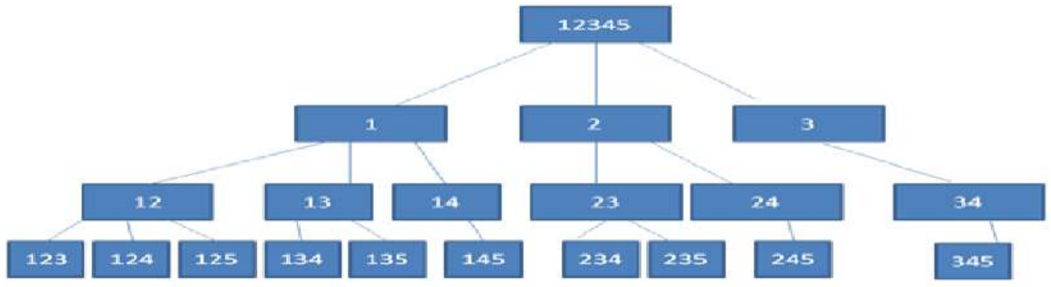
Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

Method 1 (Fix Elements and Recur)

We create a temporary array 'data[]' which stores all outputs one by one. The idea is to start from first index (index = 0) in data[], one by one fix elements at this index and recur for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for remaining indexes. When number of elements in data[] becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.



```
// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[]  ---> Input Array
   data[] ---> Temporary array to store current combination
   start & end ---> Staring and Ending indexes in arr[]
   index  ---> Current index in data[]
   r ---> Size of a combination to be printed */
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r)
{
    // Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}
```

How to handle duplicates?
Note that the above method doesn't handle duplicates. For example, if input array is {1, 2, 1} and r is 2, then the program prints {1, 2} and {2, 1} as two different combinations. We can avoid duplicates by adding following two additional things to above code.
1) Add code to sort the array before calling combinationUtil() in printCombination()
2) Add following lines at the end of for loop in combinationUtil()

```
    // Since the elements are sorted, all occurrences of an element
    // must be together
    while (arr[i] == arr[i+1])
      i++;
```

Method 2 (Include and Exclude every element)
Like the above method, We create a temporary array data[]. The idea here is similar to Subset Sum Problem. We one by one consider every element of input array, and recur for two cases:
1) The element is included in current combination (We put the element in data[] and increment next available index in data[])
2) The element is excluded in current combination (We do not put the element and do not change index)
When number of elements in data[] become equal to r (size of a combination), we print it.
This method is mainly based on Pascal's Identity, i.e. ncr = n-1cr + n-1cr-1

```
// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];
     // Print all combination using temprary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}
/* arr[]  ---> Input Array
   n      ---> Size of input array
   r      ---> Size of a combination to be printed
   index  ---> Current index in data[]
   data[] ---> Temporary array to store current combination
   i      ---> index of current element in arr[]     */
void combinationUtil(int arr[], int n, int r, int index, int data[], int
i)
{
    // Current cobination is ready, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ",data[j]);
        printf("\n");
        return;
    }
     // When no more elements are there to put in data[]
    if (i >= n)
        return;
     // current is included, put next at next location
    data[index] = arr[i];
    combinationUtil(arr, n, r, index+1, data, i+1);

    // current is excluded, replace it with next (Note that
    // i+1 is passed, but index is not changed)
    combinationUtil(arr, n, r, index, data, i+1);
}
```

How to handle duplicates in method 2?

Like method 1, we can following two things to handle duplicates.

1) Add code to sort the array before calling combinationUtil() in printCombination()

2) Add following lines between two recursive calls of combinationUtil() in combinationUtil()

```
    // Since the elements are sorted, all occurrences of an element
    // must be together
    while (arr[i] == arr[i+1])
      i++;
```

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

http://www.geeksforgeeks.org/given-an-array-of-of-size-n-finds-all-the-elements-that-appear-more-than-nk-times/

Given an array of size n, find all elements in array that appear more than n/k times. For example, if the input arrays is {3, 1, 2, 2, 1, 2, 3, 3} and k is 4, then the output should be [2, 3]. Note that size of array is 8 (or n = 8), so we need to find all elements that appear more than 2 (or 8/4) times. There are two elements that appear more than two times, 2 and 3.

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).**

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) which is O(nLogn).

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. 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 C++ implementation of above algorithm.

```
// Prints elements with more than n/k occurrences in arr[] of
// size n. If there are no such elements, then it prints nothing.
void moreThanNdK(int arr[], int n, int 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 */
    struct eleCount temp[k-1];
    for (int i=0; i<k-1; i++)
        temp[i].c = 0;

    /* Step 2: Process all elements of input array */
    for (int i = 0; i < n; i++)
```

```
{
    int j;

    /* If arr[i] is already present in
       the element count array, then increment its count */
    for (j=0; j<k-1; j++)
    {
        if (temp[j].e == arr[i])
        {
             temp[j].c += 1;
             break;
        }
    }

    /* If arr[i] is not present in temp[] */
    if (j == k-1)
    {
        int l;

        /* If there is position available in temp[], then place
          arr[i] in the first available position and set count as 1*/
        for (l=0; l<k-1; l++)
        {
            if (temp[l].c == 0)
            {
                temp[l].e = arr[i];
                temp[l].c = 1;
                break;
            }
        }

        /* If all the position in the temp[] are filled, then
           decrease count of every element by 1 */
        if (l == k-1)
            for (l=0; l<k; l++)
                temp[l].c -= 1;
    }
}

/*Step 3: Check actual counts of potential candidates in temp[]*/
for (int i=0; i<k-1; i++)
{
    // Calculate actual count of elements
    int ac = 0;  // actual count
    for (int j=0; j<n; j++)
        if (arr[j] == temp[i].e)
            ac++;
        // If actual count is more than n/k, then print it
        if (ac > n/k)
           cout << "Number:" << temp[i].e
                << " Count:" << ac << endl;
    }
}
```

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.

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 above variations with O(1) extra space.

Exercise:
The above problem can be solved in O(nLogk) time with the help of more appropriate data structures than array for auxiliary storage of k-1 elements. Suggest a O(nLogk) approach.

75 Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time)

http://www.geeksforgeeks.org/find-the-point-where-a-function-becomes-negative/

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).

```
// Returns the value x where above function f() becomes positive
// first time.
int findFirstPositive()
{
    // When first value itself is positive
    if (f(0) > 0)
        return 0;

    // Find 'high' for binary search by repeated doubling
    int i = 1;
    while (f(i) <= 0)
        i = i*2;

    //  Call binary search
    return binarySearch(i/2, i);
}
 // Searches first positive value of f(i) where low <= i <= high
int binarySearch(int low, int high)
{
    if (high >= low)
    {
        int mid = low + (high - low)/2; /* mid = (low + high)/2 */

        // If f(mid) is greater than 0 and one of the following two
        // conditions is true:
        // a) mid is equal to low
        // b) f(mid-1) is negative
        if (f(mid) > 0 && (mid == low || f(mid-1) <= 0))
            return mid;

        // If f(mid) is smaller than or equal to 0
        if (f(mid) <= 0)
            return binarySearch((mid + 1), high);
        else // f(mid) > 0
            return binarySearch(low, (mid -1));
    }

    /* Return -1 if there is no positive value in given range */
    return -1;
}
```

Related Article:
http://www.geeksforgeeks.org/exponential-search/

76 Find the Increasing subsequence of length three with maximum product

http://www.geeksforgeeks.org/increasing-subsequence-of-length-three-with-maximum-product/

Given a sequence of non-negative integers, find the subsequence of length 3 having maximum product with the numbers of the subsequence being in ascending order.
Examples:
Input:
arr[] = {6, 7, 8, 1, 2, 3, 9, 10}
Output:
8 9 10
Input:
arr[] = {1, 5, 10, 8, 9}
Output: 5 8 9

Since we want to find the maximum product, we need to find following two things for every element in the given sequence:
LSL: The largest smaller element on left of given element
LGR: The largest greater element on right of given element.
Once we find LSL and LGR for an element, we can find the product of element with its LSL and LGR (if they both exist). We calculate this product for every element and return maximum of all products.

A **simple method** is to use nested loops. The outer loop traverses every element in sequence. Inside the outer loop, run two inner loops (one after other) to find LSL and LGR of current element. **Time complexity of this method is O(n2)**.

We can do this **in O(nLogn) time**. For simplicity, let us first create two arrays LSL[] and LGR[] of size n each where n is number of elements in input array arr[]. The main task is to fill two arrays LSL[] and LGR[]. Once we have these two arrays filled, all we need to find maximum product LSL[i]*arr[i]*LGR[i] where 0 < i < n-1 (Note that LSL[i] doesn't exist for i = 0 and LGR[i] doesn't exist for i = n-1). We can fill LSL[] in O(nLogn) time. The idea is to use a Balanced Binary Search Tree like AVL. We start with empty AVL tree, insert the leftmost element in it. Then we traverse the input array starting from the second element to second last element. For every element currently being traversed, we find the floor of it in AVL tree. If floor exists, we store the floor in LSL[], otherwise we store NIL. After storing the floor, we insert the current element in the AVL tree.

We can fill LGR[] in O(n) time. We traverse from right side and keep track of the largest element. If the largest element is greater than current element, we store it in LGR[], otherwise we store NIL.

Finally, we run a O(n) loop and return maximum of LSL[i]*arr[i]*LGR[i]
Overall complexity of this approach is O(nLogn) + O(n) + O(n) which is O(nLogn). Auxiliary space required is O(n). Note that we can avoid space required for LSL, we can find and use LSL values in final loop.

77 Find the minimum element in a sorted and rotated array

http://www.geeksforgeeks.org/find-minimum-element-in-a-sorted-and-rotated-array/

A sorted array is rotated at some unknown point, find the minimum element in it.
Following solution assumes that all elements are distinct.
Examples
Input: {5, 6, 1, 2, 3, 4}
Output: 1
Input: {1, 2, 3, 4}
Output: 1
Input: {2, 1}
Output: 1

A simple solution is to traverse the complete array and find minimum. **This solution requires Θ(n) time.**

We can do it in O(Logn) using Binary Search. If we take a closer look at above examples, we can easily figure out following pattern:

The minimum element is the only element whose previous is greater than it. If there is no previous element element, then there is no rotation (first element is minimum). We check this condition for middle element by comparing it with (mid-1)'th and (mid+1)'th elements.
If minimum element is not at middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.
If middle element is smaller than last element, then the minimum element lies in left half
Else minimum element lies in right half.

```
int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
       return arr[mid+1];

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
       return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}
```

How to handle duplicates?
It turned out that duplicates can't be handled in O(Logn) time in all cases. The special cases that cause problems are like {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to go to left half or right half by doing constant number of comparisons at the middle. So the problem with repetition can be solved in O(n) worst case.

78 Merge k sorted arrays | Set 1

http://www.geeksforgeeks.org/merge-k-sorted-arrays/

Given k sorted arrays of size n each, merge them and print the sorted output.
Example:
Input:
k = 3, n = 4
arr[][] = { {1, 3, 5, 7},
        {2, 4, 6, 8},
        {0, 9, 10, 11}} ;

Output: 0 1 2 3 4 5 6 7 8 9 10 11

A simple solution is to create an output array of size n*k and one by one copy all arrays to it. Finally, sort the output array using any O(nLogn) sorting algorithm. **This approach takes O(nkLognk) time.**

We can merge arrays in O(nk*Logk) time using Min Heap. Following is detailed algorithm.

1. Create an output array of size n*k.
2. Create a min heap of size k and insert 1st element in all the arrays into a the heap
3. Repeat following steps n*k times.
   a) Get minimum element from heap (minimum is always at root) and store it in output array.
   b) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

```
// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them together
// and prints the final sorted output.
int *mergeKArrays(int arr[][n], int k)
{
    int *output = new int[n*k];  // To store output array

    // Create a min heap with k heap nodes.  Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[k];
    for (int i = 0; i < k; i++)
    {
        harr[i].element = arr[i][0]; // Store the first element
        harr[i].i = i;  // index of array
        harr[i].j = 1;  // Index of next element to be stored from array
    }
    MinHeap hp(harr, k); // Create the heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < n*k; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();
        output[count] = root.element;

        // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < n)
        {
            root.element = arr[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element =  INT_MAX; //INT_MAX is for infinite

        // Replace root with next element of array
        hp.replaceMin(root);
    }

    return output;
}
```

Time Complexity: The main step is 3rd step, the loop runs n*k times. In every iteration of loop, we call heapify which takes O(Logk) time. Therefore, the time complexity is O(nk Logk). There are other interesting methods to merge k sorted arrays in O(nkLogk), we will sonn be discussing them as separate posts.

79 Radix Sort

http://www.geeksforgeeks.org/radix-sort/

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

What if the elements are in range from 1 to n2?

We can't use counting sort because counting sort will take O(n2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.
Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?
If we have log2n bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.
Implementation of Radix Sort
Following is a simple C++ implementation of Radix Sort. For simplicity, the value of d is assumed to be 10.

```
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

80 Move all zeroes to end of array

http://www.geeksforgeeks.org/move-zeroes-end-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).
Input : arr[] = {1, 2, 0, 4, 3, 0, 5, 0};
Output : arr[] = {1, 2, 4, 3, 5, 0, 0};

There can be many ways to solve this problem. Following is a simple and interesting way to solve this problem.

Traverse the given array 'arr' from left to right. While traversing, maintain count of non-zero elements in array. Let the count be 'count'. For every non-zero element arr[i], put the element at 'arr[count]' and increment 'count'. After complete traversal, all non-zero elements have already been shifted to front end and 'count' is set as index of first 0. Now all we need to do is that run a loop which makes all elements zero from 'count' till end of the array.

Time Complexity: O(n) where n is number of elements in input array.
Auxiliary Space: O(1)

81 Find number of pairs (x, y) in an array such that x^y > y^x

http://www.geeksforgeeks.org/find-number-pairs-xy-yx/

```
Given two arrays X[] and Y[] of positive integers, find number of pairs such that x^y >
y^x where x is an element from X[] and y is an element from Y[].
Examples:
 Input: X[] = {2, 1, 6}, Y = {1, 5}
 Output: 3
 // There are total 3 pairs where pow(x, y) is greater than pow(y, x)
 // Pairs are (2, 1), (2, 5) and (6, 1)

 Input: X[] = {10, 19, 18}, Y[] = {11, 15, 9};
 Output: 2
 // There are total 2 pairs where pow(x, y) is greater than pow(y, x)
 // Pairs are (10, 11) and (10, 15)
```

The **brute force solution** is to consider each element of X[] and Y[], and check whether the given condition satisfies or not. Time Complexity of this solution is O(m*n) where m and n are sizes of given arrays.

Efficient Solution:

The problem can be solved in O(nLogn + mLogn) time. The trick here is, if y > x then x^y > y^x with some exceptions. Following are simple steps based on this trick.

1) Sort array Y[].
2) For every x in X[], find the index idx of smallest number greater than x (also called ceil of x) in Y[] using binary search or we can use the inbuilt function upper_bound() in algorithm library.
3) All the numbers after idx satisfy the relation so just add (n-idx) to the count.

Base Cases and Exceptions:
Following are exceptions for x from X[] and y from Y[]
If x = 0, then the count of pairs for this x is 0.
If x = 1, then the count of pairs for this x is equal to count of 0s in Y[].
The following cases must be handled separately as they don't follow the general rule that x smaller than y means x^y is greater than y^x.
a) x = 2, y = 3 or 4
b) x = 3, y = 2
Note that the case where x = 4 and y = 2 is not there

Time Complexity : Let m and n be the sizes of arrays X[] and Y[] respectively. The sort step takes O(nLogn) time. Then every element of X[] is searched in Y[] using binary search. This step takes O(mLogn) time. Overall time complexity is O(nLogn + mLogn).

82 Count all distinct pairs with difference equal to k

http://www.geeksforgeeks.org/count-pairs-difference-equal-k/

```
Given an integer array and a positive integer k,count all distinct pairs with difference equal to k
Input: arr[] = {1, 5, 3, 4, 2}, k = 3
Output: 2
There are 2 pairs with difference 3, the pairs are {1, 4} and {5, 2}
Input: arr[] = {8, 12, 16, 4, 0, 20}, k = 4
Output: 5
There are 5 pairs with difference 4, the pairs are {0, 4}, {4, 8}, {8, 12}, {12, 16} and {16, 20}
```

Method 1 (Simple)

A simple solution is to consider all pairs one by one and check difference between every pair. Following program implements the simple solution. We run two loops: the outer loop picks the first element of pair, the inner loop looks for the other element. This solution doesn't work if there are duplicates in array as the requirement is to count only distinct pairs.

Time Complexity: (n2)

Method 2 (Use Sorting)

We can find the count in O(nLogn) time using a O(nLogn) sorting algorithm like Merge Sort, Heap Sort, etc. Following are the detailed steps.

```
1) Initialize count as 0
2) Sort all numbers in increasing order.
3) Remove duplicates from array.
4) Do following for each element arr[i]
  a) Binary Search for arr[i] + k in subarray from i+1 to n-1.
  b) If arr[i] + k found, increment count.
5) Return count.
```

Time complexity: The first step (sorting) takes O(nLogn) time. The second step runs binary search n times, so the time complexity of second step is also O(nLogn). Therefore, overall time complexity is O(nLogn). The second step can be optimized to O(n), see this.

Method 3 (Use Self-balancing BST)

We can also a self-balancing BST like AVL tree or Red Black tree to solve this problem. Following is detailed algorithm.

```
1) Initialize count as 0.
2) Insert all elements of arr[] in an AVL tree. While inserting,
  ignore an element if already present in AVL tree.
3) Do following for each element arr[i].
  a) Search for arr[i] + k in AVL tree, if found then increment count.
  b) Search for arr[i] - k in AVL tree, if found then increment count.
  c) Remove arr[i] from AVL tree.
```

Time complexity of above solution is also O(nLogn) as search and delete operations take O(Logn) time for a self-balancing binary search tree.

Method 4 (Use Hashing)
We can also use hashing to achieve the average time complexity as O(n) for many cases.

```
1) Initialize count as 0.
2) Insert all distinct elements of arr[] in a hash map.  While inserting,
  Ignore an element if already present in the hash map.
3) Do following for each element arr[i].
  a) Look for arr[i] + k in the hash map, if found then increment count.
  b) Look for arr[i] - k in the hash map, if found then increment count.
  c) Remove arr[i] from hash table.
```

Method 5 (Use Sorting)

```
Sort the array arr
Take two pointers, l and r, both pointing to 1st element
Take the difference arr[r] – arr[l]
If value diff is K, increment count and move both pointers to next element
if value diff > k, move l to next element
if value diff < k, move r to next element
Return count
```

Time Complexity: O(nlogn)

83 Find if there is a subarray with 0 sum

http://www.geeksforgeeks.org/find-if-there-is-a-subarray-with-0-sum/

```
Given an array of positive and negative numbers, find if there is a subarray (of size at-least one) with 0 sum.
Examples:
Input: {4, 2, -3, 1, 6}
Output: true
There is a subarray with zero sum from index 1 to 3.

Input: {4, 2, 0, 1, 6}
Output: true
There is a subarray with zero sum from index 2 to 2.

Input: {-3, 2, 3, 1, 6}
Output: false
There is no subarray with zero sum.
```

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

We can also **use hashing**. The idea is to iterate through the array and for every element arr[i], calculate sum of elements form 0 to i (this can simply be done as sum += arr[i]). If the current sum has been seen before, then there is a zero sum array. Hashing is used to store the sum values, so that we can quickly store sum and find out whether the current sum is seen before or not.
Example :
arr[] = {1, 4, -2, -2, 5, -4, 3}

```
If we consider all prefix sums, we can
notice that there is a subarray with 0
sum when :
1) Either a prefix sum repeats or
2) Or prefix sum becomes 0.

Prefix sums for above array are:
1, 5, 3, 1, 6, 2, 5

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

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.

Exercise:
Extend the above program to print starting and ending indexes of all subarrays with 0 sum.

84 Smallest subarray with sum greater than a given value

http://www.geeksforgeeks.org/minimum-length-subarray-sum-greater-given-value/

```
Given an array of integers and a number x, find the smallest subarray with sum greater than
the given value.
Examples:
arr[] = {1, 4, 45, 6, 0, 19}
  x  =  51
Output: 3
Minimum length subarray is {4, 45, 6}
arr[] = {1, 10, 5, 2, 7}
```

```
  x  = 9
Output: 1
Minimum length subarray is {10}

arr[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250}
   x = 280
Output: 4
Minimum length subarray is {100, 1, 0, 200}
arr[] = {1, 2, 4}
   x = 8
Output : Not Possible
Whole array sum is smaller than 8.
```

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.
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 post.

```
// 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 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;
}
```

How to handle negative numbers?

The above solution may not work if input array contains negative numbers. For example arr[] = {- 8, 1, 4, 2, -6}. To handle negative numbers, add a condition to ignore subarrays with negative sums.

```
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 x
        while (curr_sum <= x && end < n)
        {
            // Ignore subarrays with negative sum if
            // x is positive.
            if (curr_sum <= 0 && x > 0)
            {
                start = end;
                curr_sum = 0;
            }

            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;
}
```

85 Sort an array according to the order defined by another array

http://www.geeksforgeeks.org/sort-array-according-order-defined-another-array/

Given two arrays A1[] and A2[], sort A1 in such a way that the relative order among the elements will be same as those are in A2. For the elements not present in A2, append them at last in sorted order.
Input: A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}
A2[] = {2, 1, 8, 3}
Output: A1[] = {2, 2, 1, 1, 8, 8, 3, 5, 6, 7, 9}
The code should handle all cases like number of elements in A2[] may be more or less compared to A1[]. A2[] may have some elements which may not be there in A1[] and vice versa is also possible.

Method 1 (Using Sorting and Binary Search)

```
Let size of A1[] be m and size of A2[] be n.
1) Create a temporary array temp of size m and copy contents of A1[] to it.
2) Create another array visited[] and initialize all entries in it as false. visited[] is used to mark those elements in temp[] which are copied to A1[].
3) Sort temp[]
4) Initialize the output index ind as 0.
5) Do following for every element of A2[i] in A2[]
.....a) Binary search for all occurrences of A2[i] in temp[], if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited[]
6) Copy all unvisited elements from temp[] to A1[].
```

Time complexity: The steps 1 and 2 require O(m) time. Step 3 requires O(mLogm) time. Step 5 requires O(nLogm) time. Therefore overall time complexity is O(m + nLogm).

Method 2 (Using Self-Balancing Binary Search Tree)

We can also use a self balancing BST like AVL Tree, Red Black Tree, etc. Following are detailed steps.

```
1) Create a self balancing BST of all elements in A1[]. In every node of BST, also keep track of count of occurrences of the key and a bool field visited which is initialized as false for all nodes.
2) Initialize the output index ind as 0.
3) Do following for every element of A2[i] in A2[]
.....a) Search for A2[i] in the BST, if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited in the BST node.
4) Do an inorder traversal of BST and copy all unvisited keys to A1[].
```

Time Complexity of this method is same as the previous method. Note that in a self balancing Binary Search Tree, all operations require logm time.

Method 3 (Use Hashing)
1. Loop through A1[], store the count of every number in a HashMap (key: number, value: count of number) .
2. Loop through A2[], check if it is present in HashMap, if so, put in output array that many times and remove the number from HashMap.
3. Sort the rest of the numbers present in HashMap and put in output array.

Method 4 (By Writing a Customized Compare Method)
We can also customize compare method of a sorting algorithm to solve the above problem. For example qsort() in C allows us to pass our own customized compare method.
1. If num1 and num2 both are in A2 then number with lower index in A2 will be treated smaller than other.
2. If only one of num1 or num2 present in A2, then that number will be treated smaller than the other which doesn't present in A2.
3. If both are not in A2, then natural ordering will be taken.
Time complexity of this method is O(mnLogm) if we use a O(nLogn) time complexity sorting algorithm. We can improve **time complexity** to O(mLogm) by using a Hashing instead of doing linear search.

86 Maximum Sum Path in Two Arrays

http://www.geeksforgeeks.org/maximum-sum-path-across-two-arrays/

Given two sorted arrays such the arrays may have some common elements. Find the sum of the maximum sum path to reach from beginning of any array to end of any of the two arrays. We can switch from one array to another array only at common elements.
Expected time complexity is O(m+n) where m is the number of elements in ar1[] and n is the number of elements in ar2[].
Input: ar1[] = {2, 3, 7, 10, 12}, ar2[] = {1, 5, 7, 8}
Output: 35
35 is sum of 1 + 5 + 7 + 10 + 12.
We start from first element of arr2 which is 1, then we
move to 5, then 7. From 7, we switch to ar1 (7 is common)
and traverse 10 and 12.
Input: ar1[] = {10, 12}, ar2 = {5, 7, 9}
Output: 22
22 is sum of 10 and 12.
Since there is no common element, we need to take all
elements from the array with more sum.
Input: ar1[] = {2, 3, 7, 10, 12, 15, 30, 34}
 ar2[] = {1, 5, 7, 8, 10, 15, 16, 19}
Output: 122
122 is sum of 1, 5, 7, 8, 10, 12, 15, 30, 34

The idea is to do something similar to merge process of merge sort. We need to calculate sums of elements between all common points for both arrays. Whenever we see a common point, we compare the two sums and add the maximum of two to the result. Following are detailed steps.

1) Initialize result as 0. Also initialize two variables sum1 and sum2 as 0. Here sum1 and sum2 are used to store sum of element in ar1[] and ar2[] respectively. These sums are between two common points.
2) Now run a loop to traverse elements of both arrays. While traversing compare current elements of ar1[] and ar2[].
   2.a) If current element of ar1[] is smaller than current element of ar2[], then update sum1, else if current element of ar2[] is smaller, then update sum2.
   2.b) If current element of ar1[] and ar2[] are same, then take the maximum of sum1 and sum2 and add it to the result. Also add the common element to the result.

```
// This function returns the sum of elements on maximum path
// from beginning to end
int maxPathSum(int ar1[], int ar2[], int m, int n)
{
    // initialize indexes for ar1[] and ar2[]
    int i = 0, j = 0;

    // Initialize result and current sum through ar1[] and ar2[].
    int  result = 0, sum1 = 0, sum2 = 0;

    // Below 3 loops are similar to merge in merge sort
    while (i < m && j < n)
    {
        // Add elements of ar1[] to sum1
        if (ar1[i] < ar2[j])
            sum1 += ar1[i++];

        // Add elements of ar2[] to sum2
        else if (ar1[i] > ar2[j])
            sum2 += ar2[j++];
         else  // we reached a common point
        {
            // Take the maximum of two sums and add to result
            result += max(sum1, sum2);

            // Update sum1 and sum2 for elements after this
            // intersection point
            sum1 = 0, sum2 = 0;

            // Keep updating result while there are more common
```

```
                // elements
                while (i < m &&  j < n && ar1[i] == ar2[j])
                {
                    result = result + ar1[i++];
                    j++;
                }
            }
        }
         // Add remaining elements of ar1[]
        while (i < m)
            sum1  +=  ar1[i++];

        // Add remaining elements of ar2[]
        while (j < n)
            sum2 +=  ar2[j++];

        // Add maximum of two sums of remaining elements
        result +=  max(sum1, sum2);

        return result;
    }
```

Time complexity: In every iteration of while loops, we process an element from either of the two arrays. There are total m + n elements. Therefore, time complexity is O(m+n).

87 Sort an array in wave form

http://www.geeksforgeeks.org/sort-array-wave-form-2/

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] >= .....

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

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

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

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

A **Simple Solution** is to **use sorting**. First sort the input array, then swap all adjacent elements.
For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.
The time complexity of the above solution is O(nLogn) if a O(nLogn) sorting algorithm like Merge Sort, Heap Sort, .. etc is used.
This can be done in O(n) time by doing a single traversal of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about odd positioned element.
Following are simple steps.
1) Traverse all even positioned elements of input array, and do following.
....a) If current element is smaller than previous odd element, swap previous and current.
....b) If current element is smaller than next odd element, swap next and current.

```
// 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]);
    }
}
```

88 K'th Smallest/Largest Element in Unsorted Array | Set 1

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.
Examples:
Input: arr[] = {7, 10, 4, 3, 20, 15}
    k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
    k = 4
Output: 10

Method 1 (Simple Solution)

A Simple Solution is to sort the given array using a O(nlogn) sorting algorithm like Merge Sort, Heap Sort, etc and return the element at index k-1 in the sorted array. **Time Complexity of this solution is O(nLogn)**

Method 2 (Using Min Heap – HeapSelect)

We can find k'th smallest element in time complexity better than O(nLogn). A simple optomization is to create a Min Heap of the given n elements and call extractMin() k times.

```
// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of n elements: O(n) time
    MinHeap mh(arr, n);

    // Do extract min (k-1) times
    for (int i=0; i<k-1; i++)
        mh.extractMin();

    // Return root
    return mh.getMin();
}
```

Time complexity of this solution is O(n + kLogn).

Method 3 (Using Max-Heap)

We can also use Max Heap for finding the k'th smallest element. Following is algorithm.
1) Build a Max-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 k'th element (arr[k] to arr[n-1]), compare it with root of MH.
……a) If the element is less than the root then make it root and call heapify for MH
……b) Else ignore it.
// The step 2 is O((n-k)*logk)
3) Finally, root of the MH is the kth smallest element.
Time complexity of this solution is O(k + (n-k)*Logk)

```
// Function to return k'th largest element in a given array
int kthSmallest(int arr[], int n, int k)
{    // Build a heap of first k elements: O(k) time
    MaxHeap mh(arr, k);
     // Process remaining n-k elements.  If current element is
    // smaller than root, replace root with current element
    for (int i=k; i<n; i++)
        if (arr[i] < mh.getMax())
           mh.replaceMax(arr[i]);
     // Return root
    return mh.getMax(); }
```

Method 4 (QuickSelect)

This is an optimization over method 1 if QuickSort is used as a sorting algorithm in first step. In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the array around it. 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.

The worst case time complexity of this method is O(n2), but it works in O(n) on average.

```
// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}
```

There are two more solutions which are better than above discussed ones:
One solution is to do randomized version of quickSelect() and
other solution is worst case linear time algorithm (see the following posts).

89 K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

We recommend to read following post as a prerequisite of this post.
K'th Smallest/Largest Element in Unsorted Array | Set 1
Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.

```
Examples:
Input: arr[] = {7, 10, 4, 3, 20, 15}
     k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
     k = 4
Output: 10
```

QuickSelect discussed in the previous post. The idea is to **randomly pick a pivot element**. To implement randomized partition, we use a random function, 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.
Following is implementation of above Randomized QuickSelect.

```
// Picks a random pivot element between l and r and partitions
// arr[l..r] arount the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}
```

Time Complexity:
The worst case time complexity of the above solution is still O(n2). In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is Θ(n), The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

90 K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/

In previous post, we discussed an expected linear time algorithm. In this post, a worst case linear time method is discussed. The idea in this new method is similar to quickSelect(), we get worst case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on other side). After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of pivot.

Following is complete algorithm.

```
kthSmallest(arr[0..n-1], k)
1) Divide arr[] into ⌈n/5rceil; groups where size of each group is 5
   except possibly the last group which may have less  than 5 elements.

2) Sort the above created ⌈n/5⌉ groups and find median
   of all groups. Create an auxiliary array 'median[]' and store medians
   of all ⌈n/5⌉ groups in this median array.

// Recursively call this method to find median of median[0..⌈n/5⌉-1]
3) medOfMed = kthSmallest(median[0..⌈n/5⌉-1], ⌈n/10⌉)

4) Partition arr[] around medOfMed and obtain its position.
    pos = partition(arr, n, medOfMed)

5) If pos == k return medOfMed
6) If pos < k return kthSmallest(arr[l..pos-1], k)
7) If poa > k return kthSmallest(arr[pos+1..r], k-pos+l-1)
```

In above algorithm, last 3 steps are same as algorithm in previous post. The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

→ Details Solution followed above link.

91 Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array

http://www.geeksforgeeks.org/find-index-0-replaced-1-get-longest-continuous-sequence-1s-binary-array/

Given an array of 0s and 1s, find the position of 0 to be replaced with 1 to get longest continuous sequence of 1s. Expected time complexity is O(n) and auxiliary space is O(1).
Example:
Input:
  arr[] = {1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1}
Output:
 Index 9
Assuming array index starts from 0, replacing 0 with 1 at index 9 causes
the maximum continuous sequence of 1s.

Input:
  arr[] = {1, 1, 1, 1, 0}
Output:
 Index 4

A **Simple Solution** is to traverse the array, for every 0, count the number of 1s on both sides of it. Keep track of maximum count for any 0. Finally return index of the 0 with maximum number of 1s around it. The **time complexity** of this solution is O(n2).

Using an **Efficient Solution**, the problem can solved in O(n) time. The idea is to keep track of three indexes, current index (curr), previous zero index (prev_zero) and previous to previous zero index (prev_prev_zero). Traverse the array, if current element is 0, calculate the difference between curr and prev_prev_zero (This difference minus one is the number of 1s around the prev_zero). If the difference between curr and prev_prev_zero is more than maximum so far, then update the maximum. Finally return index of the prev_zero with maximum difference.

```
// Returns index of 0 to be replaced with 1 to get longest
// continuous sequence of 1s.  If there is no 0 in array, then
// it returns -1.
int maxOnesIndex(bool arr[], int n)
{
    int max_count = 0;  // for maximum number of 1 around a zero
    int max_index;  // for storing result
    int prev_zero = -1;  // index of previous zero
    int prev_prev_zero = -1; // index of previous to previous zero
    // Traverse the input array
    for (int curr=0; curr<n; ++curr)
    {
        // If current element is 0, then calculate the difference
        // between curr and prev_prev_zero
        if (arr[curr] == 0)
        {
            // Update result if count of 1s around prev_zero is
more
            if (curr - prev_prev_zero > max_count)
            {
                max_count = curr - prev_prev_zero;
                max_index = prev_zero;
            }

            // Update for next iteration
            prev_prev_zero = prev_zero;
            prev_zero = curr;
        }
    }

    // Check for the last encountered zero
    if (n-prev_prev_zero > max_count)
       max_index = prev_zero;

    return max_index;
}
```

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

92 Find the closest pair from two sorted arrays

http://www.geeksforgeeks.org/given-two-sorted-arrays-number-x-find-pair-whose-sum-closest-x/

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
```

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between ar1[i] + ar2[j] and x.

We can do it in O(n) time **using following steps**.
1) Merge given two arrays into an auxiliary array of size m+n using merge process of merge sort. While merging keep another boolean array of size m+n to indicate whether the current element in merged array is from ar1[] or ar2[].
2) Consider the merged array and use the linear time algorithm to find the pair with sum closest to x. One extra thing we need to consider only those pairs which have one element from ar1[] and other from ar2[], we use the boolean array for this purpose.

Can we do it in a single pass and O(1) extra space?
The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach.

Following is detailed algorithm.

```
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.
```

93 Given a sorted array and a number x, find the pair in array whose sum is closest to x

http://quiz.geeksforgeeks.org/given-sorted-array-number-x-find-pair-array-whose-sum-closest-x/

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:
Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54
Output: 22 and 30
Input: arr[] = {1, 3, 4, 7, 10}, x = 15
Output: 4 and 10

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. **Time complexity** of this solution is O(n2)

An **efficient solution** can find the pair in **O(n) time**. The idea is similar to method 2 of this post. Following is detailed algorithm.

```
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:  l = 0
     (b) Initialize second  the rightmost index:  r = n-1

3) Loop while l < r.
     (a) If  abs(arr[l] + arr[r] - sum) < diff  then
        update diff and result
     (b) Else if(arr[l] + arr[r] <  sum )  then l++
     (c) Else r--
```

94 Count 1's in a sorted binary array

http://quiz.geeksforgeeks.org/count-1s-sorted-binary-array/

Given a binary array sorted in non-increasing order, count the number of 1's in it.
Examples:
Input: arr[] = {1, 1, 0, 0, 0, 0, 0}
Output: 2
Input: arr[] = {1, 1, 1, 1, 1, 1, 1}
Output: 7
Input: arr[] = {0, 0, 0, 0, 0, 0, 0}
Output: 0

A simple solution is to linearly traverse the array. The **time complexity** of the simple solution is O(n).

We can use **Binary Search** to find count in **O(Logn) time**. The idea is to look for last occurrence of 1 using Binary Search. Once we find the index last occurrence, we return index + 1 as count.

95 Print All Distinct Elements of a given integer array

http://quiz.geeksforgeeks.org/print-distinct-elements-given-integer-array/

Given an integer array, print all distinct elements in array. The given array may contain duplicates and the output should print every element only once. The given array is not sorted.
Examples:
Input: arr[] = {12, 10, 9, 45, 2, 10, 10, 45}
Output: 12, 10, 9, 45, 2

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

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

A **Simple Solution** is to use two nested loops. The outer loop picks an element one by one starting from the leftmost element. The inner loop checks if the element is present on left side of it. If present, then ignores the element, else prints the element.
Time Complexity of above solution is O(n2).

We can **Use Sorting** to solve the problem in **O(nLogn) time**. The idea is simple, first sort the array so that all occurrences of every element become consecutive. Once the occurrences become consecutive, we can traverse the sorted array and print distinct elements in O(n) time

We can Use **Hashing** to solve this in O(n) time on average. The idea is to traverse the given array from left to right and keep track of visited elements in a hash table.
One more advantage of hashing over sorting is, the elements are printed in same order as they are in input array.

96 Construct an array from its pair-sum array

http://quiz.geeksforgeeks.org/construct-array-pair-sum-array/

Given a pair-sum array and size of the original array (n), construct the original array.
A pair-sum array for an array is the array that contains sum of all pairs in ordered form. For example pair-sum array for arr[] = {6, 8, 3, 4} is {14, 9, 10, 11, 12, 7}.
In general, pair-sum array for arr[0..n-1] is {arr[0]+arr[1], arr[0]+arr[2], ……., arr[1]+arr[2], arr[1]+arr[3], ……., arr[2]+arr[3], arr[2]+arr[4], …., arr[n-2]+arr[n-1}.
“Given a pair-sum array, construct the original array.”

Let the given array be "pair[]" and let there be n elements in original array. If we take a look at few examples, we can observe that arr[0] is half of pair[0] + pair[1] – pair[n-1]. Note that the value of pair[0] + pair[1] – pair[n-1] is (arr[0] + arr[1]) + (arr[0] + arr[2]) – (arr[1] + arr[2]). Once we have evaluated arr[0], we can evaluate other elements by subtracting arr[0]. For example arr[1] can be evaluated by subtracting arr[0] from pair[0], arr[2] can be evaluated by subtracting arr[0] from pair[1].

```
// Fills element in arr[] from its pair sum array pair[].
// n is size of arr[]
void constructArr(int arr[], int pair[], int n)
{
    arr[0] = (pair[0]+pair[1]-pair[n-1]) / 2;
    for (int i=1; i<n; i++)
        arr[i] = pair[i-1]-arr[0];
}
```

Time complexity of constructArr() is O(n) where n is number of elements in arr[].

97 Find common elements in three sorted arrays

http://www.geeksforgeeks.org/find-common-elements-three-sorted-arrays/

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.
Examples:
ar1[] = {1, 5, 10, 20, 40, 80}
ar2[] = {6, 7, 20, 80, 100}
ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}
Output: 20, 80

ar1[] = {1, 5, 5}
ar2[] = {3, 4, 5, 5, 10}
ar3[] = {5, 5, 10, 20}
Output: 5, 5

A **simple solution** is to first find intersection of two arrays and store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is O(n1 + n2 + n3) where n1, n2 and n3 are sizes of ar1[], ar2[] and ar3[] respectively.

The **above solution requires extra space** and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to intersection of two arrays. Like two arrays loop, we run a loop and traverse three arrays.
Let the current element traversed in ar1[] be x, in ar2[] be y and in ar3[] be z.

We can have following cases inside the loop.
1) If x, y and z are same, we can simply print any of them as common element and move ahead in all three arrays.
2) Else If x < y, we can move ahead in ar1[] as x cannot be a common element 3) Else If y < z, we can move ahead in ar2[] as y cannot be a common element 4) Else (We reach here when x > y and y > z), we can simply move ahead in ar3[] as z cannot be a common element.

```
// 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;

    // Iterate through three arrays while all arrays have elements
    while (i < n1 && j < n2 && k < n3)
    {
         // If x = y and y = z, print any of them and move ahead
         // in all arrays
         if (ar1[i] == ar2[j] && ar2[j] == ar3[k])
         {   cout << ar1[i] << " ";   i++; j++; k++; }

         // x < y
         else if (ar1[i] < ar2[j])
             i++;
          // y < z
         else if (ar2[j] < ar3[k])
             j++;

         // We reach here when x > y and z < y, i.e., z is smallest
         else
             k++;
    }
}
```

Time complexity of the above solution is O(n1 + n2 + n3). In worst case, the largest sized array may have all small elements and middle sized array has all middle elements.

98 Find the first repeating element in an array of integers

http://www.geeksforgeeks.org/find-first-repeating-element-array-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]

Input: arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10}
Output: 6 [6 is the first element that repeats]

A **Simple Solution** is to use two nested loops. The outer loop picks an element one by one, the inner loop checks whether the element is repeated or not. Once we find an element that repeats, we break the loops and print the element.
Time Complexity of this solution is O(n2)
We can **Use Sorting** to solve the problem in O(nLogn) time. Following are detailed steps.
1) Copy the given array to an auxiliary array temp[].
2) Sort the temp array using a O(nLogn) time sorting algorithm.
3) Scan the input array from left to right. For every element, count its occurrences in temp[] using binary search. As soon as we find an element that occurs more than once, we return the element. This step can be done in O(nLogn) time.

We can **Use Hashing** to solve this in **O(n) time** on average. The idea is to traverse the given array from right to left and update the minimum index whenever we find an element that has been visited on right side.

```
// This function prints the first repeating element in arr[]
    static void printFirstRepeating(int arr[])
    {
        // Initialize index of first repeating element
        int min = -1;

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array from right to left
        for (int i=arr.length-1; i>=0; i--)
        {
            // If element is already in hash set, update min
            if (set.contains(arr[i]))
                min = i;

            else   // Else add element to hash set
                set.add(arr[i]);
        }

        // Print the result
        if (min != -1)
          System.out.println("The first repeating element is " +
arr[min]);
        else
          System.out.println("There are no repeating elements");
    }
```

99 Find the smallest positive integer value that cannot be represented as sum of any subset of a given array

http://www.geeksforgeeks.org/find-smallest-value-represented-sum-subset-given-array/

Given a sorted array (sorted in non-decreasing order) of positive numbers, find the smallest positive integer value that cannot be represented as sum of elements of any subset of given set.
Expected time complexity is O(n).

Examples:
Input: arr[] = {1, 3, 6, 10, 11, 15};
Output: 2

Input: arr[] = {1, 1, 1, 1};
Output: 5

Input: arr[] = {1, 1, 3, 4};
Output: 10

Input: arr[] = {1, 2, 5, 10, 20, 40};
Output: 4

Input: arr[] = {1, 2, 3, 4, 5, 6};
Output: 22

A **Simple Solution** is to start from value 1 and check all values one by one if they can sum to values in the given array. This solution is very inefficient as it reduces to subset sum problem which is a well known NP Complete Problem.

We can solve this problem in O(n) time **using a simple loop**. Let the input array be arr[0..n-1]. We initialize the result as 1 (smallest possible outcome) and traverse the given array. Let the smallest element that cannot be represented by elements at indexes from 0 to (i-1) be 'res', there are following two possibilities when we consider element at index i:

1) We decide that 'res' is the final result: If arr[i] is greater than 'res', then we found the gap which is 'res' because the elements after arr[i] are also going to be greater than 'res'.

2) The value of 'res' is incremented after considering arr[i]: The value of 'res' is incremented by arr[i] (why? If elements from 0 to (i-1) can represent 1 to 'res-1', then elements from 0 to i can represent from 1 to 'res + arr[i] – 1' be adding 'arr[i]' to all subsets that represent 1 to 'res')

```
// Returns the smallest number that cannot be represented as sum
// of subset of elements from set represented by sorted array arr[0..n-1]

int findSmallest(int arr[], int n)
{
   int res = 1; // Initialize result

   // Traverse the array and increment 'res' if arr[i] is
   // smaller than or equal to 'res'.
   for (int i = 0; i < n && arr[i] <= res; i++)
       res = res + arr[i];

   return res;
}
```

Time Complexity of above program is O(n).

100 Rearrange an array such that 'arr[j]' becomes 'i' if 'arr[i]' is 'j'

http://www.geeksforgeeks.org/rearrange-array-arrj-becomes-arri-j/

Given an array of size n where all elements are in range from 0 to n-1, change contents of arr[] so that arr[i] = j is changed to arr[j] = i.

Examples:

Example 1:

Input: arr[] = {1, 3, 0, 2};

Output: arr[] = {2, 0, 3, 1};

Explanation for the above output.

Since arr[0] is 1, arr[1] is changed to 0

Since arr[1] is 3, arr[3] is changed to 1

Since arr[2] is 0, arr[0] is changed to 2

Since arr[3] is 2, arr[2] is changed to 3

Example 2:

Input: arr[] = {2, 0, 1, 4, 5, 3};

Output: arr[] = {1, 2, 0, 5, 3, 4};

Example 3:

Input: arr[] = {0, 1, 2, 3};

Output: arr[] = {0, 1, 2, 3};

Example 4:

Input: arr[] = {3, 2, 1, 0};

Output: arr[] = {3, 2, 1, 0};

A **Simple Solution** is to create a temporary array and one by one copy 'i' to 'temp[arr[i]]' where i varies from 0 to n-1.

Time complexity of the above solution is O(n) and auxiliary space needed is O(n).

Can we solve this in O(n) time and O(1) auxiliary space?
The idea is based on the fact that the modified array is basically a permutation of input array. We can find the target permutation by storing the next item before updating it.
Let us consider array '{1, 3, 0, 2}' for example. We start with i = 0, arr[i] is 1. So we go to arr[1] and change it to 0 (because i is 0). Before we make the change, we store old value of arr[1] as the old value is going to be our new index i. In next iteration, we have i = 3, arr[3] is 2, so we change arr[2] to 3. Before making the change we store next i as old value of arr[2].
The below code gives idea about this approach.

```
void rearrangeUtil(int arr[], int n, int i)
{
    // 'val' is the value to be stored at 'arr[i]'
    int val = -(i+1);   // The next value is determined
                        // using current index
    i = arr[i] - 1;  // The next index is determined
                     // using current value

    // While all elements in cycle are not processed
    while (arr[i] > 0)
    {
        // Store value at index as it is going to be
        // used as next index
        int new_i = arr[i] - 1;

        // Update arr[]
        arr[i] = val;

        // Update value and index for next iteration
        val = -(i + 1);
        i = new_i;
    }
}
```

The above function **doesn't work** for inputs like {2, 0, 1, 4, 5, 3}; as there are two cycles. One cycle is (2, 0, 1) and other cycle is (4, 5, 3).

How to handle multiple cycles with the O(1) space constraint?
The idea is to process all cycles one by one. To check whether an element is processed or not, we change the value of processed items arr[i] as -arr[i]. Since 0 can not be made negative, we first change all arr[i] to arr[i] + 1. In the end, we make all values positive and subtract 1 to get old values back.

```
// A space efficient method to rearrange 'arr[0..n-1]'
// so that 'arr[j]' becomes 'i' if 'arr[i]' is 'j'
void rearrange(int arr[], int n)
{
    // Increment all values by 1, so that all elements
    // can be made negative to mark them as visited
    int i;
    for (i=0; i<n; i++)
        arr[i]++;

    // Process all cycles
    for (i=0; i<n; i++)
    {
       // Process cycle starting at arr[i] if this cycle is
       // not already processed
       if (arr[i] > 0)
          rearrangeUtil(arr, n, i);
    }

    // Change sign and values of arr[] to get the original
    //  values back, i.e., values in range from 0 to n-1
    for (i=0; i<n; i++)
          arr[i] = (-arr[i]) - 1;
}
```

The **time complexity** of this method seems to be more than O(n) at first look. If we take a closer look, we can notice that no element is processed more than constant number of times. The time complexity of this method seems to be more than O(n) at first look. If we take a closer look, we can notice that no element is processed more than constant number of times.

101 Find position of an element in a sorted array of infinite numbers

http://www.geeksforgeeks.org/find-position-element-sorted-array-infinite-numbers/

Suppose you have a sorted array of infinite numbers, how would you search an element in the array?

Since array is sorted, the first thing clicks into mind is binary search, but the problem here is that we don't know size of array.
If the array is infinite, that means we don't have proper bounds to apply binary search.
So in order to find position of key, first we find bounds and then apply binary search algorithm.

Let low be pointing to 1st element and high pointing to 2nd element of array, Now compare key with high index element,
->if it is greater than high index element then copy high index in low index and double the high index.
->if it is smaller, then apply binary search on high and low indices found.
Below are implementations of above algorithm

```
// function takes an infinite size array and a key to be
//  searched and returns its position if found else -1.
// We don't know size of arr[] and we can assume size to be
// infinite in this function.
// NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE
// THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING
int findPos(int arr[], int key)
{
    int l = 0, h = 1;
    int val = arr[0];

    // Find h to do binary search
    while (val < key)
    {
        l = h;        // store previous high
        h = 2*h;      // double high index
        val = arr[h]; // update new val
    }

    // at this point we have updated low and high indices,
    // thus use binary search between them
    return binarySearch(arr, l, h, key);
}
```

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

102 Check if a given array contains duplicate elements within k distance from each other

http://www.geeksforgeeks.org/check-given-array-contains-duplicate-elements-within-k-distance/

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.
Examples:
Input: k = 3, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}

Output: false
All duplicates are more than k distance away.

Input: k = 3, arr[] = {1, 2, 3, 1, 4, 5}
Output: true
1 is repeated at distance 3.

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

Input: k = 3, arr[] = {1, 2, 3, 4, 4}
Output: true

A **Simple Solution** is to run two loops. The outer loop picks every element 'arr[i]' as a starting element, the inner loop compares all elements which are within k distance of 'arr[i]'. The time complexity of this solution is O(kn).

We can **solve this problem** in Θ(n) time **using Hashing**. The idea is to one by add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

```
1) Create an empty hashtable.
2) Traverse all elements from left from right. Let the current element be 'arr[i]'
....a) If current element 'arr[i]' is present in hashtable, then return true.
....b) Else add arr[i] to hash and remove arr[i-k] from hash if i is greater than or equal to k
```

```
static boolean checkDuplicatesWithinK(int arr[], int k)
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();
         // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If already present n hash, then we found
            // a duplicate within k distance
            if (set.contains(arr[i]))
               return true;
             // Add this item to hashset
            set.add(arr[i]);
             // Remove the k+1 distant item
            if (i >= k)
              set.remove(arr[i-k]);
        }
        return false;
}
```

103 Pythagorean Triplet in an array

http://www.geeksforgeeks.org/find-pythagorean-triplet-in-an-unsorted-array/

Given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies a2 + b2 = c2.
Example:
Input: arr[] = {3, 1, 4, 6, 5}
Output: True
There is a Pythagorean triplet (3, 4, 5).

Input: arr[] = {10, 4, 6, 12, 5}
Output: False
There is no Pythagorean triplet.

Method 1 (Naive)
A simple solution is to run three loops, three loops pick three array elements and check if current three elements form a Pythagorean Triplet.
Time Complexity of the above solution is O(n3).

Method 2 (Use Sorting)
We can solve this in O(n2) time by sorting the array first.
1) Do square of every element in input array. This step takes O(n) time.
2) Sort the squared array in increasing order. This step takes O(nLogn) time.
3) To find a triplet (a, b, c) such that a = b + c, do following.
Fix 'a' as last element of sorted array.
Now search for pair (b, c) in subarray between first element and 'a'. A pair (b, c) with given sum can be found in O(n) time using meet in middle algorithm discussed in method 1 of this post.
If no pair found for current 'a', then move 'a' one position back and repeat step 3.2.
Time complexity of this method is O(n2).

104 Find Union and Intersection of two unsorted arrays

http://www.geeksforgeeks.org/find-union-and-intersection-of-two-unsorted-arrays/

Given two unsorted arrays that represent two sets (elements in every array are distinct), find union and intersection of two arrays.
For example, if the input arrays are:
arr1[] = {7, 1, 5, 2, 3, 6}
arr2[] = {3, 8, 6, 20, 7}
Then your program should print Union as {1, 2, 3, 5, 6, 7, 8, 20} and Intersection as {3, 6}. Note that the elements of union and intersection can be printed in any order.

Method 1 (Naive)

Union:

1) Initialize union U as empty.
2) Copy all elements of first array to U.
3) Do following for every element x of second array:
.....a) If x is not present in first array, then copy x to U.
4) Return U.

Intersection:

1) Initialize intersection I as empty.
2) Do following for every element x of first array
.....a) If x is present in second array, then copy x to I.
4) Return I.

Time complexity of this method is O(mn) for both operations. Here m and n are number of elements in arr1[] and arr2[] respectively.

Method 2 (Use Sorting)

1) Sort arr1[] and arr2[]. This step takes O(mLogm + nLogn) time.
2) Use O(m + n) algorithms to find union and intersection of two sorted arrays.
Overall time complexity of this method is O(mLogm + nLogn).

Method 3 (Use Sorting and Searching)

Union:

1) Initialize union U as empty.
2) Find smaller of m and n and sort the smaller array.
3) Copy the smaller array to U.
4) For every element x of larger array, do following
.......b) Binary Search x in smaller array. If x is not present, then copy it to U.
5) Return U.

Intersection:

1) Initialize intersection I as empty.
2) Find smaller of m and n and sort the smaller array.
3) For every element x of larger array, do following
.......b) Binary Search x in smaller array. If x is present, then copy it to I.
4) Return I.

Time complexity of this method is min(mLogm + nLogm, mLogn + nLogn) which can also be written as O((m+n)Logm, (m+n)Logn). This approach works much better than the previous approach when difference between sizes of two arrays is significant.

Method 4 (Use Hashing)
Union:
1) Initialize union U as empty.
1) Initialize an empty hash table.
2) Iterate through first array and put every element of first array in the hash table, and in U.
4) For every element x of second array, do following
…….a) Search x in the hash table. If x is not present, then copy it to U.
5) Return U.
Intersection:
1) Initialize intersection I as empty.
2) In initialize an empty hash table.
3) Iterate through first array and put every element of first array in the hash table.
4) For every element x of second array, do following
…….a) Search x in the hash table. If x is present, then copy it to I.
5) Return I.
Time complexity of this method is Θ(m+n) under the assumption that hash table search and insert operations take Θ(1) time.

105 Count frequencies of all elements in array in O(1) extra space and O(n) time

http://www.geeksforgeeks.org/count-frequencies-elements-array-o1-extra-space-time/

Given an unsorted array of n integers which can contain integers from 1 to n. Some elements can be repeated multiple times and some other elements can be absent from the array. Count frequency of all elements that are present and print the missing elements.
Examples:

```
Input: arr[] = {2, 3, 3, 2, 5}
Output: Below are frequencies of all elements
       1 -> 0
       2 -> 2
       3 -> 2
       4 -> 0
       5 -> 1

Input: arr[] = {4, 4, 4, 4}
Output: Below are frequencies of all elements
       1 -> 0
       2 -> 0
       3 -> 0
       4 -> 4
```

A **Simple Solution** is to create a count array of size n as the elements are in range from 1 to n. This solution works in O(n) time, but requires O(n) extra space.

How to do it in O(1) extra space and O(n) time?
Below are two methods to solve this in O(n) time and O(1) extra space. Both method modify given array to achieve O(1) extra space.

Method 1 (By making elements negative)
The idea is to traverse the given array, use elements as index and store their counts at the index. For example, when we see element 7, we go to index 6 and store the count. There are few problems to handle, one is the counts can get mixed with the elements, this is handled by storing the counts as negative. Other problem is loosing the element which is replaced by count, this is handled by first storing the element to be replaced at current index.

```
// Function to find counts of all elements present in
// arr[0..n-1]. The array elements must be range from
// 1 to n
void findCounts(int *arr, int n)
{
    // Traverse all array elements
    int i = 0;
    while (i<n)
    {
        // If this element is already processed,
        // then nothing to do
        if (arr[i] <= 0)
        {
            i++;
            continue;
        }

        // Find index corresponding to this element
        // For example, index for 5 is 4
        int elementIndex = arr[i]-1;

        // If the elementIndex has an element that is not
        // processed yet, then first store that element
        // to arr[i] so that we don't loose anything.
        if (arr[elementIndex] > 0)
        {
            arr[i] = arr[elementIndex];

            // After storing arr[elementIndex], change it
            // to store initial count of 'arr[i]'
            arr[elementIndex] = -1;
        }
        else
        {    // If this is NOT first occurrence of arr[i],
            // then increment its count.
            arr[elementIndex]--;

            // And initialize arr[i] as 0 means the element
            // 'i+1' is not seen so far
            arr[i] = 0;
            i++;
        }
    }
     printf("\nBelow are counts of all elements\n");
    for (int i=0; i<n; i++)
        printf("%d -> %d\n", i+1, abs(arr[i]));
}
```

<u>Method 2 (By adding n to keep track of counts)</u>

```
// Function to find counts of all elements present in
// arr[0..n-1]. The array elements must be range from
// 1 to n
void printfrequency(int arr[],int n)
{
 // Subtract 1 from every element so that the elements
    // become in range from 0 to n-1
    for (int j =0; j<n; j++)
        arr[j] = arr[j]-1;
     // Use every element arr[i] as index and add 'n' to
    // element present at arr[i]%n to keep track of count of
    // occurrences of arr[i]
    for (int i=0; i<n; i++)
        arr[arr[i]%n] = arr[arr[i]%n] + n;
     // To print counts, simply print the number of times n
    // was added at index corresponding to every element
    for (int i =0; i<n; i++)
        cout << i + 1 << " ->  " << arr[i]/n << endl; }
}
```

106 Generate all possible sorted arrays from alternate elements of two given sorted arrays

http://www.geeksforgeeks.org/generate-all-possible-sorted-arrays-from-alternate-elements-of-two-given-arrays/

Given two sorted arrays A and B, generate all possible arrays such that first element is taken from A then from B then from A and so on in increasing order till the arrays exhausted. The generated arrays should end with an element from B.
For Example

```
A = {10, 15, 25}
B = {1, 5, 20, 30}
```

The resulting arrays are:

```
 10 20
 10 20 25 30
 10 30
 15 20
 15 20 25 30
 15 30
 25 30
```

The idea is to use recursion. In the recursive function, a flag is passed to indicate whether current element in output should be taken from 'A' or 'B'.

107 Minimum number of swaps required for arranging pairs adjacent to each other

http://www.geeksforgeeks.org/minimum-number-of-swaps-required-for-arranging-pairs-adjacent-to-each-other/

```
There are n-pairs and therefore 2n people. everyone has one unique number ranging from 1 to
2n. All these 2n persons are arranged in random fashion in an Array of size 2n. We are also
given who is partner of whom. Find the minimum number of swaps required to arrange these
pairs such that all pairs become adjacent to each other.

Example:
Input:
n = 3
pairs[] = {1->3, 2->6, 4->5}  // 1 is partner of 3 and so on
arr[] = {3, 5, 6, 4, 1, 2}

Output: 2
We can get {3, 1, 5, 4, 6, 2} by swapping 5 & 6, and 6 & 1
```

The **idea** is to start from first and second elements and recur for remaining elements. Below are detailed steps

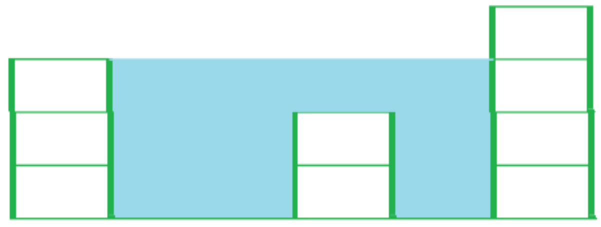
```
1) If first and second elements are pair, then simply recur
   for remaining n-1 pairs and return the value returned by
   recursive call.
```

```
2) If first and second are NOT pair, then there are two ways to
  arrange. So try both of them return the minimum of two.
  a) Swap second with pair of first and recur for n-1 elements.
    Let the value returned by recursive call be 'a'.
  b) Revert the changes made by previous step.
  c) Swap first with pair of second and recur for n-1 elements.
    Let the value returned by recursive call be 'b'.
  d) Revert the changes made by previous step before returning
    control to parent call.
  e) Return 1 + min(a, b)
```

108 Trapping Rain Water

http://www.geeksforgeeks.org/trapping-rain-water/

```
Given n non-negative integers representing an elevation map where the width of each bar is 1,
compute how much water it is able to trap after raining.
Examples:
Input: arr[]   = {3, 0, 0, 2, 0, 4}
Output: 10
Structure is like below
     |
|    |
|  | |
|__|_|
We can trap "3*2 units" of water between 3 an 2,
"1 unit" on top of bar 2 and "3 units" between 2
and 4.  See below diagram also.
```


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

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

A **Simple Solution** is to traverse every array element and find the highest bars on left and right sides. Take the smaller of two heights. The difference between smaller height and height of current element is the amount of water that can be stored in this array element.
Time complexity of this solution is O(n2).

An **Efficient Solution** is to prre-compute highest bar on left and right of every bar in **O(n) time**.

Then use these pre-computed values to find the amount of water in every array element. Below is C++ implementation of this solution.

```
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;
}
```

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

109 Convert array into Zig-Zag fashion

http://www.geeksforgeeks.org/convert-array-into-zig-zag-fashion/

Given an array of distinct elements, rearrange the elements of array in zig-zag fashion in O(n) time. The converted array should be in form a < b > c < d > e < f.
Input: arr[] = {4, 3, 7, 8, 6, 2, 1}
Output: arr[] = {3, 7, 4, 8, 2, 6, 1}
Input: arr[] = {1, 4, 3, 2}
Output: arr[] = {1, 4, 2, 3}

A **Simple Solution** is to first sort the array. After sorting, exclude the first element, swap the remaining elements in pairs. (i.e. keep arr[0] as it is, swap arr[1] and arr[2], swap arr[3] and arr[4], and so on). **Time complexity** is O(nlogn) since we need to sort the array first.

We can convert in O(n) time using an Efficient Approach. The idea is to use modified one pass of bubble sort. Maintain a flag for representing which order(i.e. < or >) currently we need. If the current two elements are not in that order then swap those elements otherwise not.
Let us see the main logic using three consecutive elements A, B, C. Suppose we are processing B and C currently and the current relation is '<'. But we have B > C. Since current relation is '<' previous relation must be '>' i.e., A must be greater than B. So, the relation is A > B and B > C. We can deduce A > C. So if we swap B and C then the relation is A > C and C < B. Finally we get the desired order A C B

```
void zigZag(int arr[], int n)
{   bool flag = true;
    for (int i=0; i<=n-2; i++)
    {
        if (flag)  /* "<" relation expected */
        {
            /* If we have a situation like A > B > C,
               we get A > B < C by swapping B and C */
            if (arr[i] > arr[i+1])
                swap(arr[i], arr[i+1]);
        }
        else /* ">" relation expected */
        {      /* If we have a situation like A < B < C,
               we get A < C > B by swapping B and C */
            if (arr[i] < arr[i+1])
                swap(arr[i], arr[i+1]);
        }
        flag = !flag; /* flip flag */}}
```

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

110 Find maximum value of Sum(i*arr[i]) with only rotations on given array allowed

http://www.geeksforgeeks.org/find-maximum-value-of-sum-iarri-with-only-rotations-on-given-array-allowed/

Given an array, only rotation operation is allowed on array. We can rotate the array as many times as we want. Return the maximum possbile of summation of i*arr[i].
Example:
Input: arr[] = {1, 20, 2, 10}
Output: 72
We can 72 by rotating array twice.
{2, 10, 1, 20}
20*3 + 1*2 + 10*1 + 2*0 = 72

Input: arr[] = {10, 1, 2, 3, 4, 5, 6, 7, 8, 9};
Output: 330
We can 330 by rotating array 9 times.
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
0*1 + 1*2 + 2*3 ... 9*10 = 330

A **Simple Solution** is to find all rotations one by one, check sum of every rotation and return the maximum sum. Time complexity of this solution is O(n2).

We can solve this problem in O(n) time using an Efficient Solution.
Let Rj be value of i*arr[i] with j rotations. The idea is to calculate next rotation value from previous rotation, i.e., calculate Rj from Rj-1. We can calculate initial value of result as R0, then keep calculating next rotation values.

How to efficiently calculate Rj from Rj-1?
This can be done in O(1) time. Below are details.
Let us calculate initial value of i*arr[i] with no rotation
R0 = 0*arr[0] + 1*arr[1] +...+ (n-1)*arr[n-1]

After 1 rotation arr[n-1], becomes first element of array,
arr[0] becomes second element, arr[1] becomes third element
and so on.
R1 = 0*arr[n-1] + 1*arr[0] +...+ (n-1)*arr[n-2]
R1 - R0 = arr[0] + arr[1] + ... + arr[n-2] - (n-1)*arr[n-1]

After 2 rotations arr[n-2], becomes first element of array,
arr[n-1] becomes second element, arr[0] becomes third element

```
and so on.
R2 = 0*arr[n-2] + 1*arr[n-1] +...+ (n?1)*arr[n-3]
R2 - R1 = arr[0] + arr[1] + ... + arr[n-3] - (n-1)*arr[n-2] + arr[n-1]
If we take a closer look at above values, we can observe
below pattern

Rj - Rj-1 = arrSum - n * arr[n-j]

Where arrSum is sum of all array elements, i.e.,
arrSum = ∑ arr[i]
       i<=0<=n-1
```

Below is complete algorithm:

```
1) Compute sum of all array elements. Let this sum be 'arrSum'.
2) Compute R0 by doing i*arr[i] for given array.
  Let this value be currVal.
3) Initialize result: maxVal = currVal // maxVal is result.

// This loop computes Rj from  Rj-1
4) Do following for j = 1 to n-1
......a) currVal = currVal + arrSum-n*arr[n-j];
......b) If (currVal > maxVal)
        maxVal = currVal
5) Return maxVal
```

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

111 Reorder an array according to given indexes

http://www.geeksforgeeks.org/reorder-a-array-according-to-given-indexes/

```
Given two integer arrays of same size, “arr[]” and “index[]”, reorder elements in “arr[]”
according to given index array. It is not allowed to given array arr’s length.
Input:  arr[]   = [10, 11, 12];
       index[] = [1, 0, 2];
Output: arr[]   = [11, 10, 12]
       index[] = [0,  1,  2]
Input:  arr[]   = [50, 40, 70, 60, 90]
       index[] = [3,  0,  4,  1,  2]
Output: arr[]   = [40, 60, 90, 50, 70]
       index[] = [0,  1,  2,  3,   4]
Expected time complexity O(n) and auxiliary space O(1)
```

A **Simple Solution** is to use an auxiliary array temp[] of same size as given arrays. Traverse the given array and put all elements at their correct place in temp[] using index[]. Finally copy temp[] to arr[] and set all values of index[i] as i.

We can solve it Without Auxiliary Array. Below is algorithm.

```
1) Do following for every element arr[i]
  a) While index[i] is not equal to i
     (i)  Store array and index values of the target (or
          correct) position where arr[i] should be placed.
          The correct position for arr[i] is index[i]
     (ii) Place arr[i] at its correct position. Also
          update index value of correct position.
     (iii) Copy old values of correct position (Stored in
          step (i)) to arr[i] and index[i] as the while
          loop continues for i.
```

112 Find zeroes to be flipped so that number of consecutive 1's is maximized

http://www.geeksforgeeks.org/find-zeroes-to-be-flipped-so-that-number-of-consecutive-1s-is-maximized/

```
Given a binary array and an integer m, find the position of zeroes flipping which creates
maximum number of consecutive 1s in array.
Examples:
Input:   arr[] = {1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1}
         m = 2
Output:  5 7

We are allowed to flip maximum 2 zeroes. If we flip
arr[5] and arr[7], we get 8 consecutive 1's which is
maximum possible under given constraints
Input:   arr[] = {1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1}
         m = 1
Output:  7
We are allowed to flip maximum 1 zero. If we flip
arr[7], we get 5 consecutive 1's which is maximum
possible under given constraints.
Input:   arr[] = {0, 0, 0, 1}
         m = 4
Output:  0 1 2
```

Since m is more than number of zeroes, we can flip

all zeroes.
zeroes in it. Return the maximum size subarray with m or less zeroes. **Time Complexity** of this solution is O(n2).

A **Better Solution** is to use auxiliary space to solve the problem in O(n) time.
For all positions of 0's calculate left[] and right[] which defines the number of consecutive 1's to the left of i and right of i respectively.

For example, for arr[] = {1, 1, 0, 1, 1, 0, 0, 1, 1, 1} and m = 1, left[2] = 2 and right[2] = 2, left[5] = 2 and right[5] = 0, left[6] = 0 and right[6] = 3.
left[] and right[] can be filled in O(n) time by traversing array once and keeping track of last seen 1 and last seen 0. While filling left[] and right[], we also store indexes of all zeroes in a third array say zeroes[]. For above example, this third array stores {2, 5, 6}

Now traverse zeroes[] and for all consecutive m entries in this array, compute the sum of 1s that can be produced. This step can be done in O(n) using left[] and right[].
An Efficient Solution can solve the problem in O(n) time and O(1) space. The idea is to use Sliding Window for the given array.

```
wL = 0; wR = 0;
nZero = 0;
bestWindowWidth = -1;

while(wR < A.length()){
   //expand to the right, update '0' count:
   if (nZero <= m){
         wR++;
         if (A[wR] == '0') nZero++;
   };

   //shrink from left, update '0' count:
   if (nZero > m){
         if (A[wL] == '0') nZero--;
         wL++;
   };

   //update best window:
   if (wR - wL > bestWindowWidth){
         bestWindowWidth = wR - wL;
         bestWR = wR;
         bestWL = wL;
   };
};
```

113 Count inversions in an array | Set 3 (Using BIT)

http://www.geeksforgeeks.org/count-inversions-array-set-3-using-bit/

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j. For simplicity, we may assume that all elements are unique.

Example:
Input: arr[] = {8, 4, 2, 1}
Output: 6
Given array has six inversions (8,4), (4,2), (8,2), (8,1), (4,1), (2,1).

Basic Approach using BIT of size Θ(maxElement):

The idea is to iterate the array from n-1 to 0. When we are at i'th index, we check how many numbers less than arr[i] are present in BIT and add it to the result. To get the count of smaller elements, getSum() of BIT is used. In his basic idea, BIT is represented as an array of size equal to maximum element plus one. So that elements can be used as an index.
After that we add current element to to the BIT[] by doing an update operation that updates count of current element from 0 to 1, and therefore updates ancestors of current element in BIT

```
int getInvCount(int arr[], int n)
{
    int invcount = 0; // Initialize result

    // Find maximum element in arr[]
    int maxElement = 0;
    for (int i=0; i<n; i++)
        if (maxElement < arr[i])
            maxElement = arr[i];

    // Create a BIT with size equal to maxElement+1 (Extra
    // one is used so that elements can be directly be
    // used as index)
    int BIT[maxElement+1];
    for (int i=1; i<=maxElement; i++)
        BIT[i] = 0;
    // Traverse all elements from right.
    for (int i=n-1; i>=0; i--)
    {
        // Get count of elements smaller than arr[i]
        invcount += getSum(BIT, arr[i]-1);

        // Add current element to BIT
        updateBIT(BIT, maxElement, arr[i], 1);
    }

    return invcount;
}
```

Time Complexity :- The update function and getSum function runs for O(log(maximumelement)) and we are iterating over n elements. So overall time complexity is : O(nlog(maximumelement)).
Auxiliary space : O(maxElement)

Better Approach using BIT of size Θ(n):
The problem with the previous approach is that it doesn't work for negative numbers as index cannot be negative. Also by updating the value till maximum element we waste time and space as it is quite possible that we may never use intermediate value. For example, lots of space an time is wasted for an array like {1, 100000}.
The idea is to convert given array to an array with values from 1 to n and relative order of smaller and greater elements remains

Example :-
arr[] = {7, -90, 100, 1}

It gets converted to,
arr[] = {3, 1, 4 ,2 }
as -90 < 1 < 7 < 100.

We only have to make BIT[] of number of elements instead of maximum element.
Changing element will not have any change in the answer as the greater elements remain greater and at same position.

```
int getInvCount(int arr[], int n)
{
    int invcount = 0; // Initialize result

     // Convert arr[] to an array with values from 1 to n and
     // relative order of smaller and greater elements remains
     // same.  For example, {7, -90, 100, 1} is converted to
    //  {3, 1, 4 ,2 }
    convert(arr, n);
    // Create a BIT with size equal to maxElement+1 (Extra
    // one is used so that elements can be directly be
    // used as index)
    int BIT[n+1];
    for (int i=1; i<=n; i++)
        BIT[i] = 0;

    // Traverse all elements from right.
    for (int i=n-1; i>=0; i--)
    {
        // Get count of elements smaller than arr[i]
        invcount += getSum(BIT, arr[i]-1);

        // Add current element to BIT
        updateBIT(BIT, n, arr[i], 1);
    }

    return invcount;
}
```

Time Complexity :- The update function and getSum function runs for O(log(n)) and we are iterating over n elements. So overall time complexity is : O(nlogn).
Auxiliary space : O(n)

114 Count Inversions of size three in a give array

http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

Given an array arr[] of size n. Three elements arr[i], arr[j] and arr[k] form an inversion of size 3 if a[i] > a[j] >a[k] and i < j < k. Find total number of inversions of size 3.
Example:
Input: {8, 4, 2, 1}
Output: 4
The four inversions are (8,4,2), (8,4,1), (4,2,1) and (8,2,1).

Input: {9, 6, 4, 5, 8}
Output: 2
The two inversions are {9, 6, 4} and {9, 6, 5}

We have already discussed inversion count of size two by merge sort, Self Balancing BST and BIT.
Simple approach :- Loop for all possible value of i, j and k and check for the condition a[i] > a[j] > a[k] and i < j < k.

Binary Indexed Tree Approach :
Like inversions of size 2, we can use Binary indexed tree to find inversions of size 3. It is strongly recommended to refer below article first.
Count inversions of size two Using BIT
The idea is similar to above method. We count the number of greater elements and smaller elements for all the elements and then multiply greater[] to smaller[] and add it to the result.
Solution :
To find out the number of smaller elements for an index we iterate from n-1 to 0. For every element a[i] we calculate the getSum() function for (a[i]-1) which gives the number of elements till a[i]-1.
To find out the number of greater elements for an index we iterate from 0 to n-1. For every element a[i] we calculate the sum of numbers till a[i] (sum smaller or equal to a[i]) by getSum() and subtract it from i (as i is the total number of element till that point) so that we can get number of elements greater than a[i].

```
int getInvCount(int arr[], int n)
{
    // Convert arr[] to an array with values from 1 to n and
    // relative order of smaller and greater elements remains
    // same.  For example, {7, -90, 100, 1} is converted to
    //  {3, 1, 4 ,2 }
    convert(arr, n);

    // Create and initialize smaller and greater arrays
    int greater1[n], smaller1[n];
    for (int i=0; i<n; i++)
        greater1[i] = smaller1[i] = 0;

    // Create and initialize an array to store Binary
    // Indexed Tree
    int BIT[n+1];
    for (int i=1; i<=n; i++)
        BIT[i]=0;

    for(int i=n-1; i>=0; i--)
    {
        smaller1[i] = getSum(BIT, arr[i]-1);
        updateBIT(BIT, n, arr[i], 1);
    }

    // Reset BIT
    for (int i=1; i<=n; i++)
        BIT[i] = 0;

    // Count greater elements
    for (int i=0; i<n; i++)
    {
        greater1[i] = i - getSum(BIT,arr[i]);
        updateBIT(BIT, n, arr[i], 1);
    }
    // Compute Inversion count using smaller[] and
    // greater[].
    int invcount = 0;
    for (int i=0; i<n; i++)
        invcount += smaller1[i]*greater1[i];

    return invcount;
}
```

Time Complexity : O(n log n)
Auxiliary Space : O(n)
We can also use Self-Balancing Binary Search Tree to count greater elements on left and smaller on right. Time complexity of this method would also be O(n Log n), But BIT based method is easy to implement.

115 Stable Marriage Problem

http://www.geeksforgeeks.org/stable-marriage-problem/
Given N men and N women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable" (Source Wiki).
Consider the following example.
Let there be two men m1 and m2 and two women w1 and w2.
Let m1's list of preferences be {w1, w2}
Let m2's list of preferences be {w1, w2}
Let w1's list of preferences be {m1, m2}
Let w2's list of preferences be {m1, m2}
The matching { {m1, w2}, {w1, m2} } is not stable because m1 and w1 would prefer each other over their assigned partners. The matching {m1, w1} and {m2, w2} is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

```
Initialize all men and women to free
while there exist a free man m who still has a woman w to propose to
{
   w = m's highest ranked such woman to whom he has not yet proposed
   if w is free
     (m, w) become engaged
   else some pair (m', w) already exists
     if w prefers m to m'
       (m, w) become engaged
        m' becomes free
     else
       (m', w) remain engaged
}
```

116 Tug of War

http://www.geeksforgeeks.org/tug-of-war/

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

Tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

117 Longest Span with same Sum in two Binary arrays

http://www.geeksforgeeks.org/longest-span-sum-two-binary-arrays/

Given two binary arrays arr1[] and arr2[] of same size n. Find length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + ... + arr1[j] = arr2[i] + arr2[i+1] + ... + arr2[j].
Expected time complexity is Θ(n).
Examples:
Input: arr1[] = {0, 1, 0, 0, 0, 0};
    arr2[] = {1, 0, 1, 0, 0, 1};
Output: 4
The longest span with same sum is from index 1 to 4.
Input: arr1[] = {0, 1, 0, 1, 1, 1, 1};
    arr2[] = {1, 1, 1, 1, 1, 0, 1};
Output: 6
The longest span with same sum is from index 1 to 6.
Input: arr1[] = {0, 0, 0};
    arr2[] = {1, 1, 1};
Output: 0

Method 1 (Simple Solution)
One by one by consider same subarrays of both arrays. For all subarrays, compute sums and if sums are same and current length is more than max length, then update max length. Below is C++ implementation of simple approach.
Time Complexity: O(n2)
Auxiliary Space: O(1)

Method 2 (Using Auxiliary Array)
The idea is based on below observations.
Since there are total n elements, maximum sum is n for both arrays.
Difference between two sums varies from -n to n. So there are total 2n + 1 possible values of difference.
If differences between prefix sums of two arrays become same at two points, then subarrays between these two points have same sum.

```
// Returns length of the longest common sum in arr1[]
// and arr2[]. Both are of same size n.
int longestCommonSum(bool arr1[], bool arr2[], int n)
{
    // Initialize result
    int maxLen = 0;

    // Initialize prefix sums of two arrays
    int preSum1 = 0, preSum2 = 0;

    // Create an array to store staring and ending
    // indexes of all possible diff values. diff[i]
    // would store starting and ending points for
    // difference "i-n"
    int diff[2*n+1];

    // Initialize all starting and ending values as -1.
    memset(diff, -1, sizeof(diff));

    // Traverse both arrays
    for (int i=0; i<n; i++)
    {
        // Update prefix sums
        preSum1 += arr1[i];
        preSum2 += arr2[i];

        // Comput current diff and index to be used
        // in diff array. Note that diff can be negative
        // and can have minimum value as -1.
        int curr_diff = preSum1 - preSum2;
        int diffIndex = n + curr_diff;

        // If current diff is 0, then there are same number
        // of 1's so far in both arrays, i.e., (i+1) is
        // maximum length.
        if (curr_diff == 0)
            maxLen = i+1;

        // If current diff is seen first time, then update
        // starting index of diff.
        else if ( diff[diffIndex] == -1)
            diff[diffIndex] = i;

        // Current diff is already seen
        else
        {
            // Find lenght of this same sum common span
            int len = i - diff[diffIndex];

            // Update max len if needed
            if (len > maxLen)
                maxLen = len;
        }
    }
    return maxLen;
}
```

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

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

http://www.geeksforgeeks.org/merge-two-sorted-arrays-o1-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[] = {10};
    ar2[] = {2, 3};
Output: ar1[] = {2}
    ar2[] = {3, 10}
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}

This task is simple and O(m+n) if we are allowed to use extra space. But it becomes really complicated when extra space is not allowed and doesn't look **possible in less than O(m*n) worst case time.**

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 type of insertion for this.

```
// Merge ar1[] and ar2[] with O(1) extra space
void merge(int ar1[], int ar2[], int m, int n)
{    // Iterate through all elements of ar2[] starting from
    // the last element
    for (int i=n-1; i>=0; i--)
    {       /* Find the smallest element greater than ar2[i]. Move all
            elements one position ahead till the smallest greater
            element is not found */
        int j, last = ar1[m-1];
        for (j=m-2; j >= 0 && ar1[j] > ar2[i]; j--)
            ar1[j+1] = ar1[j];
         // If there was a greater element
        if (j != m-2 || last > ar2[i])
        {
            ar1[j+1] = ar2[i];
            ar2[i] = last;
        }
}
}
```

Time Complexity: The worst case time complexity of code/algorithm is O(m*n). The worst case occurs when all elements of ar1[] are greater than all elements of ar2[].

119 Form minimum number from given sequence

http://www.geeksforgeeks.org/form-minimum-number-from-given-sequence/

Given a pattern containing only I's and D's. I for increasing and D for decreasing. Devise an algorithm to print the minimum number following that pattern. Digits from 1-9 and digits can't repeat.

```
Input: D        Output: 21
Input: I        Output: 12
Input: DD       Output: 321
Input: II       Output: 123
Input: DIDI     Output: 21435
Input: IIDDD    Output: 126543
Input: DDIDDIID Output: 321654798
```

120 Subarray/Substring vs Subsequence and Programs to Generate them

http://www.geeksforgeeks.org/subarraysubstring-vs-subsequence-and-programs-to-generate-them/

Subarray/Substring

A subbarray is a contiguous part of array. An array that is inside another array. For example, consider the array [1, 2, 3, 4], There are 10 non-empty sub-arrays. The subbarays are (1), (2), (3), (4), (1,2), (2,3), (3,4), (1,2,3), (2,3,4) and (1,2,3,4). In general, **for an array/string of size n, there are** n*(n+1)/2 **non-empty subarrays/subsrings**.

How to generate all subarrays?
We can run two nested loops, the outer loop picks starting element and inner loop considers all elements on right of the picked elements as ending element of subarray.

```
// Prints all subarrays in arr[0..n-1]
void subArray(int arr[], int n)
{
    // Pick starting point
    for (int i=0; i <n; i++)
    {
        // Pick ending point
        for (int j=i; j<n; j++)
        {
            // Print subarray between current starting
            // and ending points
            for (int k=i; k<=j; k++)
                cout << arr[k] << " ";

            cout << endl;
        }
    }
}
```

Subsequence

A subsequence is a sequence that can be derived from another sequence by zero or more elements, without changing the order of the remaining elements.
For the same example, there are 15 sub-sequences. They are (1), (2), (3), (4), (1,2), (1,3),(1,4), (2,3), (2,4), (3,4), (1,2,3), (1,2,4), (1,3,4), (2,3,4), (1,2,3,4). More generally, we can say that for a sequence of size n, we can have (2n-1) non-empty sub-sequences in total.

A string example to differentiate: Consider strings “geeksforgeeks” and “gks”. “gks” is a subsequence of “geeksforgeeks” but not a substring. “geeks” is both a subsequence and subarray. Every subarray is a subsequence. More specifically, Subsequence is a generalization of substring.

How to generate all Subsequences?
We can use algorithm to generate power set for generation of all subsequences.

```
void printSubsequences(int arr[], int n)
{
    /* Number of subsequences is (2**n -1)*/
    unsigned int opsize = pow(2, n);

    /* Run from counter 000..1 to 111..1*/
    for (int counter = 1; counter < opsize; counter++)
    {
        for (int j = 0; j < n; j++)
        {
            /* Check if jth bit in the counter is set
                If set then print jth element from arr[] */
            if (counter & (1<<j))
                cout << arr[j] << " ";
        }
        cout << endl;
    }
}
```

121 Count Strictly Increasing Subarrays

http://www.geeksforgeeks.org/count-strictly-increasing-subarrays/

Given an array of integers, count number of subarrays (of size more than one) that are strictly increasing.
Expected Time Complexity : O(n)
Expected Extra Space: O(1)
Input: arr[] = {1, 4, 3}
Output: 1
There is only one subarray {1, 4}
Input: arr[] = {1, 2, 3, 4}
Output: 6
There are 6 subarrays {1, 2}, {1, 2, 3}, {1, 2, 3, 4}
{2, 3}, {2, 3, 4} and {3, 4}
Input: arr[] = {1, 2, 2, 4}
Output: 2
There are 2 subarrays {1, 2} and {2, 4}

A **Simple Solution** is to generate all possible subarrays, and for every subarray check if subarray is strictly increasing or not. Worst case time complexity of this solution would be O(n3).

A **Better Solution** is to use the fact that if subarray arr[i:j] is not strictly increasing, then subarrays arr[i:j+1], arr[i:j+2], .. arr[i:n-1] cannot be strictly increasing.
Time complexity of the above solution is O(m) where m is number of subarrays in output

An **Efficient Solution** can count subarrays in O(n) time. The idea is based on fact that a sorted subarray of length 'len' adds len*(len-1)/2 to result. For example, {10, 20, 30, 40} adds 6 to the result.

```
int countIncreasing(int arr[], int n)
{
    int cnt = 0;  // Initialize result

    // Initialize length of current increasing
    // subarray
    int len = 1;

    // Traverse through the array
    for (int i=0; i < n-1; ++i)
    {
        // If arr[i+1] is greater than arr[i],
        // then increment length
        if (arr[i + 1] > arr[i])
            len++;

        // Else Update count and reset length
        else
        {
            cnt += (((len - 1) * len) / 2);
            len = 1;
        }
    }

    // If last length is more than 1
    if (len > 1)
        cnt += (((len - 1) * len) / 2);

    return cnt;
}
```

122 Rearrange an array in maximum minimum form | Set 1

http://www.geeksforgeeks.org/rearrange-array-maximum-minimum-form/

Given a sorted array of positive integers, rearrange the array alternately i.e first element should be maximum value, second minimum value, third second max, fourth second min and so on.
Examples:
Input : arr[] = {1, 2, 3, 4, 5, 6, 7}
Output : arr[] = {7, 1, 6, 2, 5, 3, 4}

Input : arr[] = {1, 2, 3, 4, 5, 6}
Output : arr[] = {6, 1, 5, 2, 4, 3}
Expected time complexity is O(n).

The idea is use an auxiliary array. We maintain two pointers one to leftmost or smallest element and other to rightmost or largest element. We more both pointers toward each other and alternatively copy elements at these pointers to an auxiliary array. Finally we copy auxiliary array back to original array
Time Complexity : O(n)
Auxiliary Space : O(n)
Exercise : How to solve this problem if extra space is not allowed?

123 Rearrange an array in maximum minimum form | Set 2 (O(1) extra space)

http://www.geeksforgeeks.org/rearrange-array-maximum-minimum-form-set-2-o1-extra-space/

Given a sorted array of positive integers, rearrange the array alternately i.e first element should be the maximum value, second minimum value, third-second max, fourth-second min and so on.
Examples:
Input : arr[] = {1, 2, 3, 4, 5, 6, 7}
Output : arr[] = {7, 1, 6, 2, 5, 3, 4}

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

In this post a solution that requires O(n) time and O(1) extra space is discussed. The idea is to use multiplication and modular trick to store two elements at an index.

How does expression “arr[i] += arr[max_index] % max_element * max_element” work ?

The purpose of this expression is to store two elements at index arr[i]. arr[max_index] is stored as multiplier and “arr[i]” is stored as remainder. For example in {1 2 3 4 5 6 7 8 9}, max_element is 10 and we store 91 at index 0. With 91, we can get original element as 91%10 and new element as 91/10.

```
// Prints max at first position, min at second position
// second max at third position, second min at fourth
// position and so on.


void rearrange(int arr[], int n)
{

    // initialize index of first minimum and first
    // maximum element
    int max_idx = n-1 , min_idx = 0 ;

    // store maximum element of array
    int max_elem = arr[n-1] + 1 ;

    // traverse array elements
    for (int i=0; i < n ; i++)
    {
        // at even index : we have to put maximum element
        if (i % 2 == 0)
        {
            arr[i] += (arr[max_idx] % max_elem ) * max_elem;
            max_idx--;
        }

        // at odd index : we have to put minimum element
        else
        {
            arr[i] += (arr[min_idx] % max_elem ) * max_elem;
            min_idx++;
        }
    }

    // array elements back to it's original form
    for (int i = 0 ; i < n; i++)
        arr[i] = arr[i] / max_elem ;
}
```

124 Find minimum difference between any two elements

http://www.geeksforgeeks.org/find-minimum-difference-pair/

Given an unsorted array, find the minimum difference between any pair in given array.
Examples :
Input : {1, 5, 3, 19, 18, 25};
Output : 1
Minimum difference is between 18 and 19

Input : {30, 5, 20, 9};
Output : 4
Minimum difference is between 5 and 9

Input : {1, 19, -4, 31, 38, 25, 100};
Output : 5
Minimum difference is between 1 and -4

Method 1 (Simple: O(n2)
A simple solution is to use two loops

Method 2 (Efficient: O(n Log n)
The idea is to use sorting. Below are steps.
1) Sort array in ascending order. This step takes O(n Log n) time.
2) Initialize difference as infinite. This step takes O(1) time.
3) Compare all adjacent pairs in sorted array and keep track of minimum difference. This step takes O(n) time.

125 Find lost element from a duplicated array

http://www.geeksforgeeks.org/find-lost-element-from-a-duplicated-array/

Given two arrays which are duplicates of each other except one element, that is one element from one of the array is missing, we need to find that missing element.
Examples:
Input: arr1[] = {1, 4, 5, 7, 9}
 arr2[] = {4, 5, 7, 9}
Output: 1
1 is missing from second array.

Input: arr1[] = {2, 3, 4, 5}
 arr2[] = {2, 3, 4, 5, 6}
Output: 6
6 is missing from first array.

One **simple solution** is to iterate over arrays and check element by element and flag the missing element when an unmatch is found, but this solution requires linear time over size of array.

Another **efficient solution** is based on binary search approach. Algorithm steps are as follows:

Start binary search in bigger array and get mid as (lo + hi) / 2
If value from both array is same then missing element must be in right part so set lo as mid
Else set hi as mid because missing element must be in left part of bigger array if mid elements are not equal.

Special case are handled separately as for single element and zero element array, single element itself will be the missing element.
If first element itself is not equal then that element will be the missing element./li>

What if input arrays are not in same order?

In this case, missing element is simply XOR of all elements of both arrays

```
void findMissing(int arr1[], int arr2[], int M,
                 int N)
{
    if (M != N-1 && N != M-1)
    {
        cout << "Invalid Input";
        return;
    }


    // Do XOR of all element



    int res = 0;
    for (int i=0; i<M; i++)
       res = res^arr1[i];
    for (int i=0; i<N; i++)
       res = res^arr2[i];

    cout << "Missing element is " << res;
}
```

126 Count minimum steps to get the given desired array

http://www.geeksforgeeks.org/count-minimum-steps-get-given-desired-array/

Consider an array with n elements and value of all the elements is zero. We can perform following operations on the array.
Incremental operations:Choose 1 element from the array and increment its value by 1.
Doubling operation: Double the values of all the elements of array.
Sample test cases:
Input: target[] = {2, 3}
Output: 4
To get the target array from {0, 0}, we
first increment both elements by 1 (2
operations), then double the array (1
operation). Finally increment second
element (1 more operation)

Input: target[] = {2, 1}
Output: 3
One of the optimal solution is to apply the
incremental operation 2 times to first and
once on second element.

Input: target[] = {16, 16, 16}
Output: 7
The output solution looks as follows. First
apply an incremental operation to each element.
Then apply the doubling operation four times.
Total number of operations is 3+4 = 7

One important thing to note is that the task is to count the number of steps to get the given target array (not to convert zero array to target array).

The idea is to follow reverse steps, i.e. to convert target to array of zeros. Below are steps.

Take the target array first.
Initialize result as 0.
If all are even, divide all elements by 2
and increment result by 1.
Find all odd elements, make them even by
reducing them by 1. and for every reduction,
increment result by 1.
Finally we get all zeros in target array.

```
int countMinOperations(unsigned int target[], int n)
{
    // Initialize result (Count of minimum moves)
    int result = 0;

    // Keep looping while all elements of target
    // don't become 0.
    while (1)
    {
        // To store count of zeroes in current
        // target array
        int zero_count = 0;

        int i;  // To find first odd element
        for (i=0; i<n; i++)
        {
            // If odd number found
            if (target[i] & 1)
                break;

            // If 0, then increment zero_count
            else if (target[i] == 0)
                zero_count++;
        }
         // All numbers are 0
        if (zero_count == n)
          return result;
         // All numbers are even
        if (i == n)
        {
            // Divide the whole array by 2
            // and increment result
            for (int j=0; j<n; j++)
               target[j] = target[j]/2;
            result++;
        }
         // Make all odd numbers even by subtracting
        // one and increment result.
        for (int j=i; j<n; j++)
        {
           if (target[j] & 1)
           {
              target[j]--;
              result++;
           }
        }}}
```

127 Find minimum number of merge operations to make an array palindrome

http://www.geeksforgeeks.org/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'. Only allowed operation on array is merge. Merging two adjacent elements means replacing them with their sum. The task is to find minimum number of merge operations required to make given array a 'Palindrome'.
To make an array a palindromic we can simply apply merging operations n-1 times where n is the size of array (Note a single element array is alway palindrome similar to single character string). In that case, size of array will be reduced to 1. But in this problem we are asked to do it in 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.
Expected time complexity is O(n).

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.
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).
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).
Our answer will be f(0, n-1), where n is 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 last element of the array) method and keeping count of total merging operations done till now.

```
// 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;
}
```

Time complexity for the given program is : O(n)

128 Minimize the maximum difference between the heights

http://www.geeksforgeeks.org/minimize-the-maximum-difference-between-the-heights/

Given heights of n towers and a value k. We need to either increase or decrease height of every tower by k (only once) where k > 0. The task is to minimize the difference between the heights of the longest and the shortest tower after modifications, and output this difference.

```
Examples:
Input  : arr[] = {1, 15, 10}, k = 6
Output : arr[] = {7, 9, 4}
         Maximum difference is 5.
Explanation : We change 1 to 6, 15 to
9 and 10 to 4. Maximum difference is 5
(between 4 and 9). We can't get a lower
difference.

Input : arr[] = {1, 5, 15, 10}
      k = 3
Output : arr[] = {4, 8, 12, 7}
Maximum difference is 8

Input : arr[] = {4, 6}
      k = 10
Output : arr[] = {14, 16} OR {-6, -4}
Maximum difference is 2

Input : arr[] = {6, 10}
      k = 3
Output : arr[] = {9, 7}
Maximum difference is 2

Input : arr[] = {1, 10, 14, 14, 14, 15}
      k = 6
Output: arr[] = {7, 4, 8, 8, 8, 9}
Maximum difference is 5

Input : arr[] = {1, 2, 3}
      k = 2
Output: arr[] = {3, 4, 5}
Maximum difference is 2
```

The idea is to sort all elements increasing order. Below are steps.

Sort array in increasing order
Initialize maximum and minimum elements.
maxe = arr[n-1], mine = arr[0]

If k is more than difference between maximum and minimum, add/subtract k to all elements as k cannot decrease the difference. Example {6, 4}, k = 10.
In sorted array, update first and last elements.
arr[0] += k; // arr[0] is minimum and k is +ve
arr[n-1] -= k; // arr[n-1] is maximum and k is -ve

Initialize mac and min of modified array (only two elements have been finalized)
new_max = max(arr[0], arr[n-1]), new_min = min(arr[0], arr[n-1])
Finalize middle n-2 elements. Do following for every element arr[j] where j lies from 1 to n-2.
If current element is less than min of modified array, add k.
Else If current element is more than max of modified array, subtract k.
arr[j] is between new_min and new_max.
If arr[j] is closer to new_max, subtract k

Else add k to arr[j].
Update new_max and new_min if required
new_max = max(arr[j], new_max), new_min = min(arr[j], new_min)
Returns difference between new_max and new_min
return (new_max – new_min);

```
int getMinDiff(int arr[], int n, int k)
{
    // There should be at least two elements
    if (n <= 1)
        return 0;

    // Sort array in increasing order
    sort(arr, arr+n);

    // Initialize maximum and minimum
    int maxe = arr[n-1];
    int mine = arr[0];
     // If k is more than difference between maximum
    // and minimum, add/subtract k to all elements
    // as k cannot decrease the difference
    if (k >= maxe - mine)
    {
        for (int i=0; i<n; i++)
```

```
            arr[i] += k; // Subtract would also work
        return (maxe - mine);
    }
     // In sorted array, first element is minimum
    // and last is maximum, we must add k to minium
    // and subtract k from maximum
    arr[0]    += k;
    arr[n-1] -= k;
     // Initialize mac and min of modified array (only
    // two elements have been finalized)
    int new_max = max(arr[0], arr[n-1]);
    int new_min = min(arr[0], arr[n-1]);

    // Finalize middle n-2 elements
    for (int j=1; j<n-1; j++)
    {
        // If current element is less than min of
        // modified array, add k.
        if (arr[j] < new_min)
            arr[j] += k;

        // If current element is more than max of
        // modified array, subtract k.
        else if (arr[j] > new_max)
            arr[j] -= k;

        // arr[j] is between new_min and new_max

        // If arr[j] is closer to new_max, subtract k
        else if ((arr[j] - new_min) > (new_max - arr[j]))
            arr[j] -= k;

        // Else add k
        else
            arr[j] += k;

        // Update new_max and new_min if required
        new_max = max(arr[j], new_max);
        new_min = min(arr[j], new_min);
    }

    // Returns difference between new_max and new_min
    return (new_max - new_min);
}
```

Time Complexity : O(n Log n)

129 Search in a row wise and column wise sorted matrix

http://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/

Given an n x n matrix, where every row and column is sorted in increasing order. Given a number x, how to decide whether this x is in the matrix. The designed algorithm should have linear time complexity.

1) Start with top right element
2) Loop: compare this element e with x
....i) if they are equal then return its position
...ii) e < x then move it to down (if out of bound of matrix then break return false) ..iii) e > x then move it to left (if out of bound of matrix then break return false)
3) repeat the i), ii) and iii) till you find element or returned false
Time Complexity: O(n)
The above approach will also work for m x n matrix (not only for n x n).
Time Complexity would be O(m + n).

130 Print a given matrix in spiral form

http://www.geeksforgeeks.org/print-a-given-matrix-in-spiral-form/

Given a 2D array, print it in spiral form. See the following examples.
Input:
```
    1   2  3  4
    5   6  7  8
    9  10  11  12
    13  14  15  16
```
Output:
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Input:
```
    1  2  3  4  5  6
    7  8  9  10  11  12
    13  14  15 16  17  18
```
Output:
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11

```
void spiralPrint(int m, int n, int a[R][C])
{
    int i, k = 0, l = 0;

    /*  k - starting row index
        m - ending row index
        l - starting column index
        n - ending column index
        i - iterator
    */

    while (k < m && l < n)
    {
        /* Print the first row from the remaining rows */
        for (i = l; i < n; ++i)
        {
            printf("%d ", a[k][i]);
        }
        k++;

        /* Print the last column from the remaining columns */
        for (i = k; i < m; ++i)
        {
            printf("%d ", a[i][n-1]);
        }
        n--;

        /* Print the last row from the remaining rows */
        if ( k < m)
        {
            for (i = n-1; i >= l; --i)
            {
                printf("%d ", a[m-1][i]);
            }
            m--;
        }

        /* Print the first column from the remaining columns */
        if (l < n)
        {
            for (i = m-1; i >= k; --i)
            {
                printf("%d ", a[i][l]);
            }
            l++;
        }
    }
}
```

Time Complexity: Time complexity of the above solution is O(mn).

131 A Boolean Matrix Question

http://www.geeksforgeeks.org/a-boolean-matrix-question/

Given a boolean matrix mat[M][N] of size M X N, modify it such that if a matrix cell mat[i][j] is 1 (or true) then make all the cells of ith row and jth column as 1.

```
Example 1
The matrix
0 0 0
0 0 1
should be changed to following
0 0 1
1 1 1

Example 2
The matrix
1 0 0 1
0 0 1 0
0 0 0 0
should be changed to following
1 1 1 1
1 1 1 1
1 0 1 1
```

Method 1 (Use two temporary arrays)

1) Create two temporary arrays row[M] and col[N]. Initialize all values of row[] and col[] as 0.

2) Traverse the input matrix mat[M][N]. If you see an entry mat[i][j] as true, then mark row[i] and col[j] as true.

3) Traverse the input matrix mat[M][N] again. For each entry mat[i][j], check the values of row[i] and col[j]. If any of the two values (row[i] or col[j]) is true, then mark mat[i][j] as true.
Time Complexity: O(M*N)
Auxiliary Space: O(M + N)

Method 2 (A Space Optimized Version of Method 1)

This method is a space optimized version of above method 1. This method uses the first row and first column of the input matrix in place of the auxiliary arrays row[] and col[] of method 1. So what we do is: first take care of first row and column and store the info about these two in two flag variables rowFlag and colFlag. Once we have this info, we can use first row and first column as auxiliary arrays and apply method 1 for submatrix (matrix excluding first row and first column) of size (M-1)*(N-1).

1) Scan the first row and set a variable rowFlag to indicate whether we need to set all 1s in first row or not.

2) Scan the first column and set a variable colFlag to indicate whether we need to set all 1s in first column or not.

3) Use first row and first column as the auxiliary arrays row[] and col[] respectively, consider the matrix as submatrix starting from second row and second column and apply method 1.

4) Finally, using rowFlag and colFlag, update first row and first column if needed.

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

132 Print unique rows in a given boolean matrix

http://www.geeksforgeeks.org/print-unique-rows/

Given a binary matrix, print all unique rows of the given matrix.

```
Input:
    {0, 1, 0, 0, 1}
     {1, 0, 1, 1, 0}
     {0, 1, 0, 0, 1}
     {1, 1, 1, 0, 0}
Output:
    0 1 0 0 1
    1 0 1 1 0
    1 1 1 0 0
```

Method 1 (Simple)
A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.
Time complexity: O(ROW^2 x COL)
Auxiliary Space: O(1)

Method 2 (Use Binary Search Tree)
Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.
Time complexity: O(ROW x COL + ROW x log(ROW))
Auxiliary Space: O(ROW)
This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)
Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Time complexity: O(ROW x COL)
Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

133 Create a matrix with alternating rectangles of 0 and X

http://www.geeksforgeeks.org/create-a-matrix-with-alternating-rectangles-of-0-and-x/

Write a code which inputs two numbers m and n and creates a matrix of size m x n (m rows and n columns) in which every elements is either X or 0. The Xs and 0s must be filled alternatively, the matrix should have outermost rectangle of Xs, then a rectangle of 0s, then a rectangle of Xs, and so on.
Examples:
Input: m = 3, n = 3
Output: Following matrix
X X X
X 0 X
X X X

Input: m = 4, n = 5
Output: Following matrix
X X X X X
X 0 0 0 X
X 0 0 0 X
X X X X X

Approach.
1) Use the code for Printing Matrix in Spiral form.

2) Instead of printing the array, inserted the element 'X' or '0' alternatively in the array.

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

134 Print Matrix Diagonally

http://www.geeksforgeeks.org/print-matrix-diagonally/

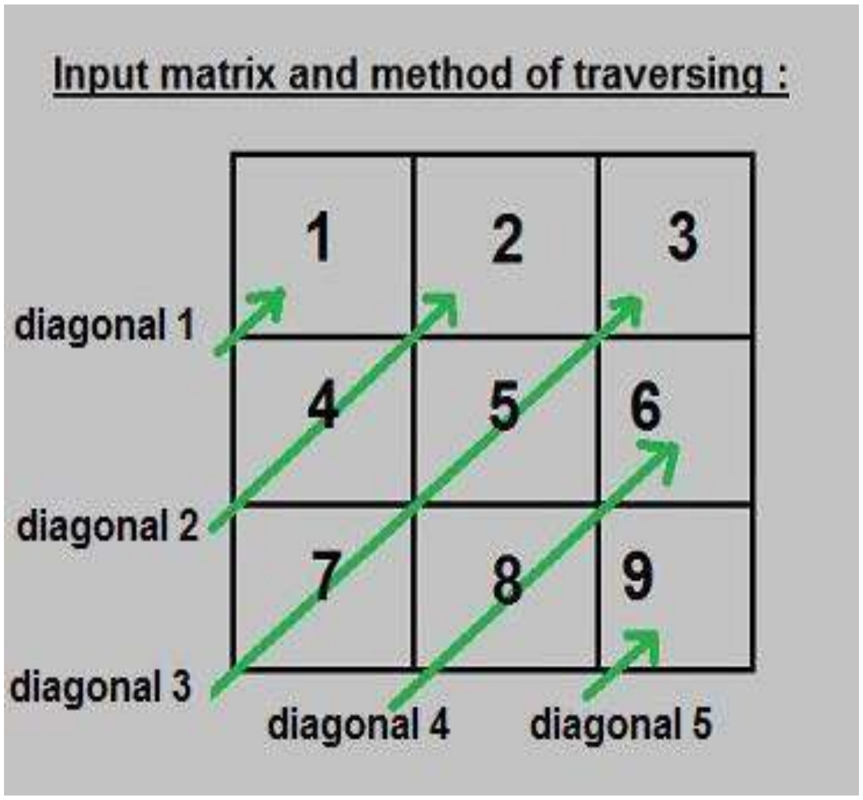
Given a 2D matrix, print all elements of the given matrix in diagonal order. For example, consider the following 5 X 4 input matrix.

```
   1     2     3     4
   5     6     7     8
   9    10    11    12
  13    14    15    16
  17    18    19    20
```

Diagonal printing of the above matrix is

```
   1
   5     2
   9     6     3
  13    10     7     4
  17    14    11     8
  18    15    12
  19    16
  20
```

Another example:



```
// The main function that prints given matrix in diagonal order
void diagonalOrder(int matrix[][COL])
{
    // There will be ROW+COL-1 lines in the output
    for (int line=1; line<=(ROW + COL -1); line++)
    {
        /* Get column index of the first element in this line of output.
           The index is 0 for first ROW lines and line - ROW for remaining
           lines  */
        int start_col =  max(0, line-ROW);

        /* Get count of elements in this line. The count of elements is
           equal to minimum of line number, COL-start_col and ROW */
         int count = min(line, (COL-start_col), ROW);

        /* Print elements of this line */
        for (int j=0; j<count; j++)
            printf("%5d ", matrix[min(ROW, line)-j-1][start_col+j]);

        /* Ptint elements of next diagonal on next line */
        printf("\n");
    }
}
```

Below is an Alternate Method to solve the above problem.

```
Matrix =>    1   2   3   4
             5   6   7   8
             9   10  11  12
             13  14  15  16
             17  18  19  20
```

Observe the sequence

```
 1 /  2 /  3 /  4
   / 5 /  6 /  7 /  8
      /  9 / 10 / 11 / 12
         / 13 / 14 / 15 / 16
            / 17 / 18 / 19 / 20
```

```
void diagonalOrder(int arr[][C])
{
    /* through this for loop we choose each element of first column
    as starting point and print diagonal starting at it.
    arr[0][0], arr[1][0]....arr[R-1][0] are all starting points */
    for (int k = 0; k < R; k++)
    {
        cout << arr[k][0] << " ";
        int i = k-1;    // set row index for next point in diagonal
        int j = 1;      //  set column index for next point in diagonal

        /* Print Diagonally upward */
        while (isvalid(i,j))
        {
            cout << arr[i][j] << " ";
            i--;
            j++;    // move in upright direction
        }
        cout << endl;
    }

    /* through this for loop we choose each element of last row
       as starting point (except the [0][c-1] it has already been
       processed in previous for loop) and print diagonal starting at it.
       arr[R-1][0], arr[R-1][1]....arr[R-1][c-1] are all starting points
*/

    //Note : we start from k = 1 to C-1;
    for (int k = 1; k < C; k++)
    {
        cout << arr[R-1][k] << " ";
        int i = R-2; // set row index for next point in diagonal
        int j = k+1; // set column index for next point in diagonal

        /* Print Diagonally upward */
        while (isvalid(i,j))
        {
            cout << arr[i][j] << " ";
            i--;
            j++; // move in upright direction
        }
        cout << endl;
    }
}
```

135 Find the row with maximum number of 1s

http://www.geeksforgeeks.org/find-the-row-with-maximum-number-1s/

Given a boolean 2D array, where each row is sorted. Find the row with the maximum number of 1s.
Example
Input matrix
0 1 1 1
0 0 1 1
1 1 1 1 // this row has maximum 1s
0 0 0 0

A **simple method** is to do a row wise traversal of the matrix, count the number of 1s in each row and compare the count with max. Finally, return the index of row with maximum 1s. The **time complexity** of this method is O(m*n) where m is number of rows and n is number of columns in matrix.

We can do better. Since each row is sorted, we can use Binary Search to count of 1s in each row. We find the index of first instance of 1 in each row. The count of 1s will be equal to total number of columns minus the index of first 1.
Time Complexity: O(mLogn) where m is number of rows and n is number of columns in matrix.

The **above solution** can be **optimized further**. Instead of doing binary search in every row, we first check whether the row has more 1s than max so far. If the row has more 1s, then only count 1s in the row. Also, to count 1s in a row, we don't do binary search in complete row, we do search in before the index of last max.
The **worst case time complexity** of the above optimized version is also O(mLogn), the will solution work better on average.

The worst case of the above solution occurs for a matrix like following.
0 0 0 ... 0 1
0 0 0 ..0 1 1
0 ... 0 1 1 1
....0 1 1 1 1
Following method works in O(m+n) time complexity in worst case.

```
Step1: Get the index of first (or leftmost) 1 in the first row.
Step2: Do following for every row after the first row
...IF the element on left of previous leftmost 1 is 0, ignore this row.
...ELSE Move left until a 0 is found. Update the leftmost index to this index and max_row_index to be the current row.
```

The **time complexity** is O(m+n) because we can possibly go as far left as we came ahead in the first step.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    // Initialize first row as row with max 1s
    int max_row_index = 0;

    // The function first() returns index of first 1 in row 0.
    // Use this index to initialize the index of leftmost 1 seen so far
    int j = first(mat[0], 0, C-1);
    if (j == -1) // if 1 is not present in first row
      j = C - 1;

    for (int i = 1; i < R; i++)
    {
        // Move left until a 0 is found
        while (j >= 0 && mat[i][j] == 1)
        {
           j = j-1;  // Update the index of leftmost 1 seen so far
           max_row_index = i;  // Update max_row_index
        }
    }
    return max_row_index;
}
```

136 Print all elements in sorted order from row and column wise sorted matrix

http://www.geeksforgeeks.org/print-elements-sorted-order-row-column-wise-sorted-matrix/

Given an n x n matrix, where every row and column is sorted in non-decreasing order. Print all elements of matrix in sorted order.

Example:

```
Input: mat[][]  = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {27, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
Output:
Elements of matrix in sorted order
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

We can **use Young Tableau** to solve the above problem. The idea is to consider given 2D array as Young Tableau and call extract minimum O(N)
Time complexity of extract minimum is O(N) and it is called O(N2) times. Therefore the overall **time complexity** is O(N3).

A **better solution** is to use the **approach** used for **merging k sorted arrays**. The idea is to use a Min Heap of size N which stores elements of first column. The do extract minimum. In extract minimum, replace the minimum element with the next element of the row from which the element is extracted. **Time complexity** of this solution is O(N2LogN).

```
// This function prints elements of a given matrix in non-decreasing
//  order. It assumes that ma[][] is sorted row wise sorted.
void printSorted(int mat[][N])
{    // Create a min heap with k heap nodes.  Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[N];
    for (int i = 0; i < N; i++)
    {
        harr[i].element = mat[i][0]; // Store the first element
        harr[i].i = i;  // index of row
        harr[i].j = 1;  // Index of next element to be stored from row
    }
    MinHeap hp(harr, N); // Create the min heap
     // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < N*N; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();
         cout << root.element << " ";
         // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < N)
        {
            root.element = mat[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element =  INT_MAX; //INT_MAX is for infinite
         // Replace root with next element of array
        hp.replaceMin(root);
    }}
```

Exercise:
Above solutions work for a square matrix. Extend the above solutions to work for an M*N rectangular matrix.

137 Given an n x n square matrix, find sum of all sub-squares of size k x k

http://www.geeksforgeeks.org/given-n-x-n-square-matrix-find-sum-sub-squares-size-k-x-k/

Given an n x n square matrix, find sum of all sub-squares of size k x k where k is smaller than or equal to n.
Examples
Input:
n = 5, k = 3

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

Output:

```
    18  18  18
    27  27  27
    36  36  36
```

A **Simple Solution** is to one by one pick starting point (leftmost-topmost corner) of all possible sub-squares. Once the starting point is picked, calculate sum of sub-square starting with the picked starting point.
Time complexity of above solution is O(k2n2).

We can solve this problem in O(n2) time using a **Tricky Solution**. The idea is to **preprocess the given square matrix**. In the preprocessing step, calculate sum of all vertical strips of size k x 1 in a temporary square matrix stripSum[][].

Once we have sum of all vertical strips, we can calculate sum of first sub-square in a row as sum of first k strips in that row, and for remaining sub-squares, we can calculate sum in O(1) time by removing the leftmost strip of previous subsquare and adding the rightmost strip of new square.

```
// A O(n^2) function to find sum of all sub-squares of size k x k
// in a given square matrix of size n x n
void printSumTricky(int mat[][n], int k)
{
   // k must be smaller than or equal to n
   if (k > n) return;
```

```
    // 1: PREPROCESSING
    // To store sums of all strips of size k x 1
    int stripSum[n][n];

    // Go column by column
    for (int j=0; j<n; j++)
    {
        // Calculate sum of first k x 1 rectangle in this column
        int sum = 0;
        for (int i=0; i<k; i++)
           sum += mat[i][j];
        stripSum[0][j] = sum;

        // Calculate sum of remaining rectangles
        for (int i=1; i<n-k+1; i++)
        {
             sum += (mat[i+k-1][j] - mat[i-1][j]);
             stripSum[i][j] = sum;
        }
    }

    // 2: CALCULATE SUM of Sub-Squares using stripSum[][]
    for (int i=0; i<n-k+1; i++)
    {
       // Calculate and print sum of first subsquare in this row
       int sum = 0;
       for (int j = 0; j<k; j++)
            sum += stripSum[i][j];
       cout << sum << "  ";

       // Calculate sum of remaining squares in current row by
       // removing the leftmost strip of previous sub-square and
       // adding a new strip
       for (int j=1; j<n-k+1; j++)
       {
          sum += (stripSum[i][j+k-1] - stripSum[i][j-1]);
          cout << sum << "  ";
       }

       cout << endl;
    }
}
```

138 Count number of islands where every island is row-wise and column-wise separated

http://www.geeksforgeeks.org/count-number-islands-every-island-separated-line/

Given a rectangular matrix which has only two possible values 'X' and 'O'. The values 'X' always appear in form of rectangular islands and these islands are always row-wise and column-wise separated by at least one line of 'O's. Note that islands can only be diagonally adjacent. Count the number of islands in the given matrix.
Examples:

```
mat[M][N] =  {{'O', 'O', 'O'},
              {'X', 'X', 'O'},
              {'X', 'X', 'O'},
              {'O', 'O', 'X'},
              {'O', 'O', 'X'},
              {'X', 'X', 'O'}
             };
Output: Number of islands is 3

mat[M][N] =  {{'X', 'O', 'O', 'O', 'O', 'O'},
              {'X', 'O', 'X', 'X', 'X', 'X'},
              {'O', 'O', 'O', 'O', 'O', 'O'},
              {'X', 'X', 'X', 'O', 'X', 'X'},
              {'X', 'X', 'X', 'O', 'X', 'X'},
              {'O', 'O', 'O', 'O', 'X', 'X'},
             };
Output: Number of islands is 4
```

The idea is to count all top-leftmost corners of given matrix. We can check if a 'X' is top left or not by checking following conditions.
1) A 'X' is top of rectangle if the cell just above it is a 'O'
2) A 'X' is leftmost of rectangle if the cell just left of it is a 'O'

Note that we must check for both conditions as there may be more than one top cells and more than one leftmost cells in a rectangular island

```
// This function takes a matrix of 'X' and 'O'
// and returns the number of rectangular islands
// of 'X' where no two islands are row-wise or
// column-wise adjacent, the islands may be diagonaly
// adjacent
int countIslands(int mat[][N])
{
    int count = 0; // Initialize result

    // Traverse the input matrix
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
        {
            // If current cell is 'X', then check
            // whether this is top-leftmost of a
            // rectangle. If yes, then increment count
            if (mat[i][j] == 'X')
            {
                if ((i == 0 || mat[i-1][j] == 'O') &&
                    (j == 0 || mat[i][j-1] == 'O'))
                    count++;
            }}}
    return count;
}
```

Time complexity of this solution is O(MN).

139 Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'

http://www.geeksforgeeks.org/given-matrix-o-x-replace-o-x-surrounded-x/

Given a matrix where every element is either 'O' or 'X', replace 'O' with 'X' if surrounded by 'X'. A 'O' (or a set of 'O') is considered to be by surrounded by 'X' if there are 'X' at locations just below, just above, just left and just right of it.

Examples:

```
Input: mat[M][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                {'X', 'O', 'X', 'X', 'O', 'X'},
                {'X', 'X', 'X', 'O', 'O', 'X'},
                {'O', 'X', 'X', 'X', 'X', 'X'},
                {'X', 'X', 'X', 'O', 'X', 'O'},
                {'O', 'O', 'X', 'O', 'O', 'O'},
               };
Output: mat[M][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                {'X', 'O', 'X', 'X', 'X', 'X'},
                {'X', 'X', 'X', 'X', 'X', 'X'},
                {'O', 'X', 'X', 'X', 'X', 'X'},
                {'X', 'X', 'X', 'O', 'X', 'O'},
                {'O', 'O', 'X', 'O', 'O', 'O'},
               };
```

This is mainly an application of Flood-Fill algorithm. The main difference here is that a ‘O’ is not replaced by ‘X’ if it lies in region that ends on a boundary. Following are simple steps to do this special flood fill.

1) Traverse the given matrix and replace all ‘O’ with a special character ‘-‘.
2) Traverse four edges of given matrix and call floodFill(‘-‘, ‘O’) for every ‘-‘ on edges. The remaining ‘-‘ are the characters that indicate ‘O’s (in the original matrix) to be replaced by ‘X’.
3) Traverse the matrix and replace all ‘-‘s with ‘X’s.

Let us see steps of above algorithm with an example. Let following be the input matrix.

```
    mat[M][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                  {'X', 'O', 'X', 'X', 'O', 'X'},
                  {'X', 'X', 'X', 'O', 'O', 'X'},
                  {'O', 'X', 'X', 'X', 'X', 'X'},
                  {'X', 'X', 'X', 'O', 'X', 'O'},
                  {'O', 'O', 'X', 'O', 'O', 'O'},
                 };
```

Step 1: Replace all ‘O’ with ‘-‘.

```
    mat[M][N] =  {{'X', '-', 'X', 'X', 'X', 'X'},
                  {'X', '-', 'X', 'X', '-', 'X'},
                  {'X', 'X', 'X', '-', '-', 'X'},
                  {'-', 'X', 'X', 'X', 'X', 'X'},
                  {'X', 'X', 'X', '-', 'X', '-'},
                  {'-', '-', 'X', '-', '-', '-'},
                 };
```

Step 2: Call floodFill(‘-‘, ‘O’) for all edge elements with value equals to ‘-‘

```
    mat[M][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                  {'X', 'O', 'X', 'X', '-', 'X'},
                  {'X', 'X', 'X', '-', '-', 'X'},
                  {'O', 'X', 'X', 'X', 'X', 'X'},
                  {'X', 'X', 'X', 'O', 'X', 'O'},
                  {'O', 'O', 'X', 'O', 'O', 'O'},
                 };
```

Step 3: Replace all ‘-‘ with ‘X’.

```
    mat[M][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                  {'X', 'O', 'X', 'X', 'X', 'X'},
                  {'X', 'X', 'X', 'X', 'X', 'X'},
                  {'O', 'X', 'X', 'X', 'X', 'X'},
                  {'X', 'X', 'X', 'O', 'X', 'O'},
                  {'O', 'O', 'X', 'O', 'O', 'O'},
                 };
```

```
// Returns size of maximum size subsquare matrix
// surrounded by 'X'
int replaceSurrounded(char mat[][N])
{
   // Step 1: Replace all 'O'  with '-'
   for (int i=0; i<M; i++)
      for (int j=0; j<N; j++)
          if (mat[i][j] == 'O')
             mat[i][j] = '-';

   // Call floodFill for all '-' lying on edges
   for (int i=0; i<M; i++)    // Left side
      if (mat[i][0] == '-')
        floodFillUtil(mat, i, 0, '-', 'O');
   for (int i=0; i<M; i++)  //  Right side
      if (mat[i][N-1] == '-')
        floodFillUtil(mat, i, N-1, '-', 'O');
   for (int i=0; i<N; i++)    // Top side
      if (mat[0][i] == '-')
        floodFillUtil(mat, 0, i, '-', 'O');
   for (int i=0; i<N; i++)  // Bottom side
      if (mat[M-1][i] == '-')
        floodFillUtil(mat, M-1, i, '-', 'O');

   // Step 3: Replace all '-' with 'X'
   for (int i=0; i<M; i++)
      for (int j=0; j<N; j++)
         if (mat[i][j] == '-')
             mat[i][j] = 'X';

}
```

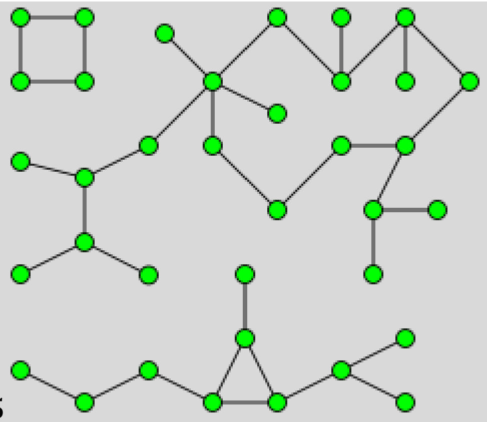
Time Complexity of the above solution is O(MN). Note that every element of matrix is processed at most three times.

140 Find the number of islands | Set 1 (Using DFS)

http://www.geeksforgeeks.org/find-number-of-islands/

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

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



Output : 5

This is an variation of the standard problem: "Counting number of connected components in a undirected graph".

Before we go to the problem, let us understand what is a connected component. A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.

For example, the graph shown below has three connected components.
A graph where all vertices are connected with each other, has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.
What is an island?
A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},
 {0, 1, 0, 0, 1},
 {1, 0, 0, 1, 1},
 {0, 0, 0, 0, 0},
 {1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbors only. We keep track of the visited 1s so that they are not visited again.

```
// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL];
    memset(visited, 0, sizeof(visited));
     // Initialize count as 0 and travese through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j] && !visited[i][j])// If a cell with value 1 is not
            {  // visited yet, then new island found
                DFS(M, i, j, visited); // Visit all cells in this island.
                ++count;               // and increment island count
            }
     return count;
}
```

Time complexity: O(ROW x COL)

141 Find the number of Islands | Set 2 (Using Disjoint Set)

http://www.geeksforgeeks.org/find-the-number-of-islands-set-2-using-disjoint-set/

Given a boolean 2D matrix, find the number of islands.
A group of connected 1s forms an island. For example, the below matrix contains 5 islands
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
A cell in 2D matrix can be connected to 8 neighbors.

This is an variation of the standard problem: "Counting number of connected components in a undirected graph". We have discussed a DFS based solution in below set 1.
Find the number of islands
We can also solve the question using disjoint set data structure explained here. The idea is to consider all 1 values as individual sets. Traverse the matrix and do union of all adjacent 1 vertices. Below are detailed steps.

Approach:
1) Initialize result (count of islands) as 0
2) Traverse each index of the 2D matrix.
3) If value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take union of index and its neighbour.
4) Now define an array of size row*column to store frequencies of all sets.
5) Now traverse the matrix again.
6) If value at index is 1, find its set.
7) If frequency of the set in the above array is 0, increment the result be 1.

142 Find the longest path in a matrix with given constraints

http://www.geeksforgeeks.org/find-the-longest-path-in-a-matrix-with-given-constraints/

Given a n*n matrix where numbers all numbers are distinct and are distributed from range 1 to n2, find the maximum length path (starting from any cell) such that all cells along the path are increasing order with a difference of 1.
We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacen
Example:
Input: mat[][] = {{1, 2, 9}
              {5, 3, 8}
              {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.

The **idea is simple**, **we calculate longest path beginning with every cell**. Once we have computed longest for all cells, we return maximum of all longest paths.
One important observation in this approach is many overlapping subproblems. Therefore this problem can be optimally solved using Dynamic Programming.

Below is Dynamic Programming based C implementation that uses a lookup table dp[][] to check if a problem is already solved or not.

```
// Returns length of the longest path beginning with mat[i][j].
// This function mainly uses lookup table dp[n][n]
int findLongestFromACell(int i, int j, int mat[n][n], int dp[n][n])
{
    // Base case
    if (i<0 || i>=n || j<0 || j>=n)
        return 0;

    // If this subproblem is already solved
    if (dp[i][j] != -1)
        return dp[i][j];

    // Since all numbers are unique and in range from 1 to n*n,
    // there is atmost one possible direction from any cell
    if (j<n-1 && ((mat[i][j] +1) == mat[i][j+1]))
       return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);

    if (j>0 && (mat[i][j] +1 == mat[i][j-1]))
       return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);

    if (i>0 && (mat[i][j] +1 == mat[i-1][j]))
```

```
        return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);

    if (i<n-1 && (mat[i][j] +1 == mat[i+1][j]))
        return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);

    // If none of the adjacent fours is one greater
    return dp[i][j] = 1;
}

// Returns length of the longest path beginning with any cell
int finLongestOverAll(int mat[n][n])
{
    int result = 1;  // Initialize result

    // Create a lookup table and fill all entries in it as -1
    int dp[n][n];
    memset(dp, -1, sizeof dp);

    // Compute longest path beginning from all cells
    for (int i=0; i<n; i++)
    {
      for (int j=0; j<n; j++)
       {
          if (dp[i][j] == -1)
             findLongestFromACell(i, j, mat, dp);

          //  Update result if needed
          result = max(result, dp[i][j]);
       }
     }

     return result;
}
```

Time complexity of the above solution is O(n^2). It may seem more at first look. If we take a closer look, we can notice that all values of dp[i][j] are computed only once.

143 Given a Boolean Matrix, find k such that all elements in k'th row are 0 and k'th column are 1.

http://www.geeksforgeeks.org/find-k-such-that-all-elements-in-kth-row-are-0-and-kth-column-are-1-in-a-boolean-matrix/

Given a square boolean matrix mat[n][n], find k such that all elements in k'th row are 0 and all elements in k'th column are 1. The value of mat[k][k] can be anything (either 0 or 1). If no such k exists, return -1.

```
Examples:
Input: bool mat[n][n] = { {1, 0, 0, 0},
                          {1, 1, 1, 0},
                          {1, 1, 0, 0},
                          {1, 1, 1, 0},
                        };
Output: 0
All elements in 0'th row are 0 and all elements in
0'th column are 1.  mat[0][0] is 1 (can be any value)


Input: bool mat[n][n] = {{0, 1, 1, 0, 1},
                         {0, 0, 0, 0, 0},
                         {1, 1, 1, 0, 0},
                         {1, 1, 1, 1, 0},
                         {1, 1, 1, 1, 1}};
Output: 1
All elements in 1'st row are 0 and all elements in
1'st column are 1.  mat[1][1] is 0 (can be any value)
Expected time complexity is O(n)
```

A **Simple Solution** is check all rows one by one. If we find a row 'i' such that all elements of this row are 0 except mat[i][i] which may be either 0 or 1, then we check all values in column 'i'. If all values are 1 in the column, then we return i.
Time complexity of this solution is O(n2).

An **Efficient Solution** can solve this problem in O(n) time. The solution is based on below facts.
1) There can be at most one k that can be qualified to be an answer (Why? Note that if k'th row has all 0's probably except mat[k][k], then no column can have all 1')s.
2) If we traverse the given matrix from a corner (preferably from top right and bottom left), we can quickly discard complete row or complete column based on below rules.
....a) If mat[i][j] is 0 and i != j, then column j cannot be the solution.
....b) If mat[i][j] is 1 and i != j, then row i cannot be the solution.

144 Find the largest rectangle of 1's with swapping of columns allowed

http://www.geeksforgeeks.org/find-the-largest-rectangle-of-1s-with-swapping-of-columns-allowed/

Given a matrix with 0 and 1's, find the largest rectangle of all 1's in the matrix. The rectangle can be formed by swapping any pair of columns of given matrix.

```
Input: bool mat[][] = { {0, 1, 0, 1, 0},
                        {0, 1, 0, 1, 1},
                        {1, 1, 0, 1, 0}
                      };
Output: 6
The largest rectangle's area is 6. The rectangle
can be formed by swapping column 2 with 3
The matrix after swapping will be
    0 0 1 1 0
    0 0 1 1 1
    1 0 1 1 0
Input: bool mat[R][C] = { {0, 1, 0, 1, 0},
                          {0, 1, 1, 1, 1},
                          {1, 1, 1, 0, 1},
                          {1, 1, 1, 1, 1} };
Output: 9
```

The **idea is to use an auxiliary matrix to store count of consecutive 1's in every column**. Once we have these counts, we sort all rows of auxiliary matrix in non-increasing order of counts. Finally traverse the sorted rows to find the maximum area.
Below are detailed steps for first example mentioned above.

Step 1: First of all, calculate no. of consecutive 1's in every column. An auxiliary array hist[][] is used to store the counts of consecutive 1's. So for the above first example, contents of hist[R][C] would be

```
    0 1 0 1 0
    0 2 0 2 1
    1 3 0 3 0
```

Time complexity of this step is O(R*C)

Step 2: Sort the rows in non-increasing fashion. After sorting step the matrix hist[][] would be

```
    1 1 0 0 0
    2 2 1 0 0
    3 3 1 0 0
```

This step can be done in O(R * (R + C)). Since we know that the values are in range from 0 to R, we can use counting sort for every row.
Step 3: Traverse each row of hist[][] and check for the max area. Since every row is sorted by count of 1's, current area can be calculated by multiplying column number with value in hist[i][j]. This step also takes O(R * C) time.
Time complexity of above solution is O(R * (R + C)) where R is number of rows and C is number of columns in input matrix.
Extra space: O(R * C)

145 Find a common element in all rows of a given row-wise sorted matrix

http://www.geeksforgeeks.org/find-common-element-rows-row-wise-sorted-matrix/

```
Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a
common element in all rows. If there is no common element, then returns -1.
Example:

Input: mat[4][5] = { {1, 2, 3, 4, 5},
                     {2, 4, 5, 8, 10},
                     {3, 5, 7, 9, 11},
                     {1, 3, 5, 7, 9},
                   };
Output: 5
```

A O(m*n*n) **simple solution is to take every element of first row and search it in all other rows, till we find a common element**. Time complexity of this solution is O(m*n*n) where m is number of rows and n is number of columns in given matrix. This can be improved to O(m*n*Logn) if we use Binary Search instead of linear search.

We can **solve** this problem **in O(mn) time** using the approach similar to merge of Merge Sort.

The idea is to start from the last column of every row. If elements at all last columns are same, then we found the common element. Otherwise we find the minimum of all last columns. Once we find a minimum element, we know that all other elements in last columns cannot be a common element, so we reduce last column index for all rows except for the row which has minimum value. We keep repeating these steps till either all elements at current last column don't become same, or a last column index reaches 0.

Explanation for working of above code
Let us understand working of above code for following example.

```
Initially entries in last column array are N-1, i.e., {4, 4, 4, 4}
   {1, 2, 3, 4, 5},
   {2, 4, 5, 8, 10},
   {3, 5, 7, 9, 11},
   {1, 3, 5, 7, 9},
The value of min_row is 0, so values of last column index for rows with value greater than 5 is
reduced by one. So column[] becomes {4, 3, 3, 3}.
   {1, 2, 3, 4, 5},
   {2, 4, 5, 8, 10},
   {3, 5, 7, 9, 11},
   {1, 3, 5, 7, 9},
```

The value of min_row remains 0 and and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 2, 2}.

```
  {1, 2, 3, 4, 5},
  {2, 4, 5, 8, 10},
  {3, 5, 7, 9, 11},
  {1, 3, 5, 7, 9},
```

The value of min_row remains 0 and value of last column index for rows with value greater than 5 is reduced by one. So colomun[] becomes {4, 2, 1, 2}.

```
  {1, 2, 3, 4, 5},
  {2, 4, 5, 8, 10},
  {3, 5, 7, 9, 11},
  {1, 3, 5, 7, 9},
```

Now all values in current last columns of all rows is same, so 5 is returned.

A Hashing Based Solution

We can also use hashing. This solution works even if the rows are not sorted. It can be used to print all common elements.

```
Step1:  Create a Hash Table with all key as distinct elements
     of row1. Value for all these will be 0.

Step2:
For i = 1 to M-1
 For j = 0 to N-1
  If (mat[i][j] is already present in Hash Table)
   If (And this is not a repetition in current row.
     This can be checked by comparing HashTable value with
     row number)
       Update the value of this key in HashTable with current
       row number

Step3: Iterate over HashTable and print all those keys for
     which value = M
```

Time complexity of the above hashing based solution is O(MN) under the assumption that search and insert in HashTable take O(1) time. Thanks to Nishant for suggesting this solution in a comment below.

Exercise: Given n sorted arrays of size m each, find all common elements in all arrays in O(mn) time.

146 Number of paths with exactly k coins

http://www.geeksforgeeks.org/number-of-paths-with-exactly-k-coins/

Given a matrix where every cell has some number of coins. Count number of ways to reach bottom right from top left with exactly k coins. We can move to (i+1, j) and (i, j+1) from a cell (i, j).

```
Example:
Input:  k = 12
      mat[][] = { {1, 2, 3},
                  {4, 6, 5},
                  {3, 2, 1}
                };
Output:  2
There are two paths with 12 coins
1 -> 2 -> 6 -> 2 -> 1
1 -> 2 -> 3 -> 5 -> 1
```

<u>The above problem can be recursively defined as below</u>:

```
pathCount(m, n, k):   Number of paths to reach mat[m][n] from mat[0][0]
                      with exactly k coins

If (m == 0 and n == 0)
   return 1 if mat[0][0] == k else return 0
Else:
    pathCount(m, n, k) = pathCount(m-1, n, k - mat[m][n]) +
                         pathCount(m, n-1, k - mat[m][n])
```

```
// Recursive function to count paths with sum k from
// (0, 0) to (m, n)
int pathCountRec(int mat[][C], int m, int n, int k)
{
    // Base cases
    if (m < 0 || n < 0) return 0;
    if (m==0 && n==0) return (k == mat[m][n]);

    // (m, n) can be reached either through (m-1, n) or
    // through (m, n-1)
    return pathCountRec(mat, m-1, n, k-mat[m][n]) +
           pathCountRec(mat, m, n-1, k-mat[m][n]);
}
```

The **time complexity** of above solution recursive is exponential.

We can solve this problem in Pseudo Polynomial Time (time complexity is dependent on numeric value of input) using Dynamic Programming. The idea is to use a 3 dimensional table dp[m][n][k] where m is row number, n is column number and k is number of coins. Below is Dynamic Programming based C++ implementation.

```
int pathCountDPRecDP(int mat[][C], int m, int n, int k)
{
    // Base cases
    if (m < 0 || n < 0) return 0;
    if (m==0 && n==0) return (k == mat[m][n]);

    // If this subproblem is already solved
    if (dp[m][n][k] != -1) return dp[m][n][k];

    // (m, n) can be reached either through (m-1, n) or
    // through (m, n-1)
    dp[m][n][k] = pathCountDPRecDP(mat, m-1, n, k-mat[m][n]) +
                  pathCountDPRecDP(mat, m, n-1, k-mat[m][n]);

    return dp[m][n][k];
}

// This function mainly initializes dp[][][] and calls
// pathCountDPRecDP()
int pathCountDP(int mat[][C], int k)
{
    memset(dp, -1, sizeof dp);
    return pathCountDPRecDP(mat, R-1, C-1, k);
}
```

Time complexity of this solution is O(m*n*k).

147 Collect maximum coins before hitting a dead end

http://www.geeksforgeeks.org/collect-maximum-coins-before-hitting-a-dead-end/

Given a character matrix where every cell has one of the following values.
'C' --> This cell has coin

'#' --> This cell is a blocking cell.
    We can not go anywhere from this.

'E' --> This cell is empty. We don't get
    a coin, but we can move from here.
Initial position is cell (0, 0) and initial direction is right.
Following are rules for movements across cells.

If face is Right, then we can move to below cells
Move one step ahead, i.e., cell (i, j+1) and direction remains right.
Move one step down and face left, i.e., cell (i+1, j) and direction becomes left.
If face is Left, then we can move to below cells
Move one step ahead, i.e., cell (i, j-1) and direction remains left.
Move one step down and face right, i.e., cell (i+1, j) and direction becomes right.
Final position can be anywhere and final direction can also be anything. The target is to collect maximum coins.

Example:

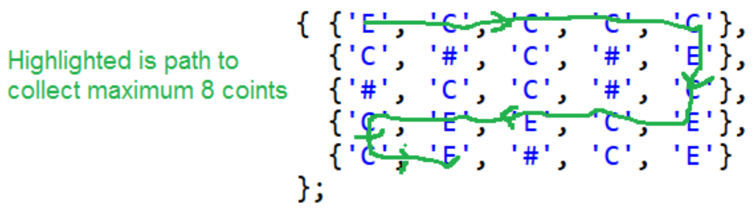
Input:

```
    char arr[R][C] = { {'E', 'C', 'C', 'C', 'C'},
                       {'C', '#', 'C', '#', 'E'},
                       {'#', 'C', 'C', '#', 'C'},
                       {'C', 'E', 'E', 'C', 'E'},
                       {'C', 'E', '#', 'C', 'E'}
                     };
```

Output:

```
     Maximum number of collected coins is 8
```

Explanation:



```
{ {'E', 'C', 'C', 'C', 'C'},
  {'C', '#', 'C', '#', 'E'},
  {'#', 'C', 'C', '#', 'C'},
  {'C', 'E', 'E', 'C', 'E'},
  {'C', 'E', '#', 'C', 'E'}
};
```

```
// to check whether current cell is out of the grid or not
bool isValid(int i, int j)
{
    return (i >=0 && i < R && j >=0 && j < C);
}

// dir = 0 for left, dir = 1 for facing right.  This function returns
// number of maximum coins that can be collected starting from (i, j).
int maxCoinsRec(char arr[R][C],  int i, int j, int dir)
{
    // If this is a invalid cell or if cell is a blocking cell
    if (isValid(i,j) == false || arr[i][j] == '#')
        return 0;

    // Check if this cell contains the coin 'C' or if its empty 'E'.
    int result = (arr[i][j] == 'C')? 1: 0;
    // Get the maximum of two cases when you are facing right in this cell
    if (dir == 1) // Direction is right
       return result + max(maxCoinsRec(arr, i+1, j, 0),      // Down
                           maxCoinsRec(arr, i, j+1, 1));   // Ahead in
right

    // Direction is left
    // Get the maximum of two cases when you are facing left in this cell
     return  result + max(maxCoinsRec(arr, i+1, j, 1),     // Down
                         maxCoinsRec(arr, i, j-1, 0));   // Ahead in left
}
```

The **time complexity** of above solution recursive is exponential. We can solve this problem in Polynomial Time using Dynamic Programming. The idea is to use a 3 dimensional table dp[R][C][k] where R is number of rows, C is number of columns and d is direction.

```
// dir = 0 for left, dir = 1 for right.  This function returns
// number of maximum coins that can be collected starting from
// (i, j).
int maxCoinsUtil(char arr[R][C],  int i, int j, int dir,
                 int dp[R][C][2])
{
    // If this is a invalid cell or if cell is a blocking cell
    if (isValid(i,j) == false || arr[i][j] == '#')
        return 0;

    // If this subproblem is already solved than return the
    // already evaluated answer.
    if (dp[i][j][dir] != -1)
       return dp[i][j][dir];

    // Check if this cell contains the coin 'C' or if its 'E'.
    dp[i][j][dir] = (arr[i][j] == 'C')? 1: 0;

    // Get the maximum of two cases when you are facing right
    // in this cell
    if (dir == 1) // Direction is right
       dp[i][j][dir] += max(maxCoinsUtil(arr, i+1, j, 0, dp), // Down
                            maxCoinsUtil(arr, i, j+1, 1, dp)); // Ahead in
rught

    // Get the maximum of two cases when you are facing left
    // in this cell
    if (dir == 0) // Direction is left
```

```
        dp[i][j][dir] += max(maxCoinsUtil(arr, i+1, j, 1, dp),  // Down
                             maxCoinsUtil(arr, i, j-1, 0, dp)); // Ahead in
left

    // return the answer
    return dp[i][j][dir];
}

// This function mainly creates a lookup table and calls
// maxCoinsUtil()
int maxCoins(char arr[R][C])
{
    // Create lookup table and initialize all values as -1
    int dp[R][C][2];
    memset(dp, -1, sizeof dp);

    // As per the question initial cell is (0, 0) and direction
    // is right
    return maxCoinsUtil(arr, 0, 0, 1, dp);
}
```

Time Complexity of above solution is O(R x C x d). Since d is 2, time complexity can be written as O(R x C).

148 Submatrix Sum Queries

http://www.geeksforgeeks.org/submatrix-sum-queries/

Given a matrix of size M x N, there are large number of queries to find submatrix sums. Inputs to queries are left top and right bottom indexes of submatrix whose sum is to find out.
How to preprocess the matrix so that submatrix sum queries can be performed in O(1) time.
Example:
tli : Row number of top left of query submatrix
tlj : Column number of top left of query submatrix
rbi : Row number of bottom right of query submatrix
rbj : Column number of bottom right of query submatrix

```
Input: mat[M][N] = {{1, 2, 3, 4, 6},
                    {5, 3, 8, 1, 2},
                    {4, 6, 7, 5, 5},
                    {2, 4, 8, 9, 4} };
Query1: tli = 0, tlj = 0, rbi = 1, rbj = 1
Query2: tli = 2, tlj = 2, rbi = 3, rbj = 4
Query3: tli = 1, tlj = 2, rbi = 3, rbj = 3;
```

```
Output:
Query1: 11  // Sum between (0, 0) and (1, 1)
Query2: 38  // Sum between (2, 2) and (3, 4)
Query3: 38  // Sum between (1, 2) and (3, 3)
```

The idea is to first create an auxiliary matrix aux[M][N] **such that aux[i][j] stores sum of elements in submatrix from (0,0) to (i,j)**. Once aux[][] is constructed, we can compute sum of submatrix between (tli, tlj) and (rbi, rbj) in O(1) time. We need to consider aux[rbi][rbj] and subtract all unncessary elements. Below is complete expression to compute submatrix sum in O(1) time.

```
Sum between (tli, tlj) and (rbi, rbj) is,
  aux[rbi][rbj] - aux[tli-1][rbj] -
  aux[rbi][tlj-1] + aux[tli-1][tlj-1]

The submatrix aux[tli-1][tlj-1] is added because
elements of it are subtracted twice.
Illustration:
mat[M][N] = {{1, 2, 3, 4, 6},
           {5, 3, 8, 1, 2},
           {4, 6, 7, 5, 5},
           {2, 4, 8, 9, 4} };

We first preprocess the matrix and build
following aux[M][N]
aux[M][N] = {1,  3,   6,  10, 16}
          {6,  11,  22, 27,  35},
          {10, 21,  39, 49,  62},
          {12, 27,  53, 72,  89} }

Query : tli = 2, tlj = 2, rbi = 3, rbj = 4

Sum between (2, 2) and (3, 4) = 89 - 35 - 27 + 11
                    = 38
```

```
How to build aux[M][N]?
1. Copy first row of mat[][] to aux[][]
2. Do column wise sum of the matrix and store it.
3. Do the row wise sum of updated matrix aux[][] in step 2.
```

```
// Function to preprcess input mat[M][N].  This function
// mainly fills aux[M][N] such that aux[i][j] stores sum
// of elements from (0,0) to (i,j)
int preProcess(int mat[M][N], int aux[M][N])
{
   // Copy first row of mat[][] to aux[][]
   for (int i=0; i<N; i++)
      aux[0][i] = mat[0][i];

   // Do column wise sum
   for (int i=1; i<M; i++)
      for (int j=0; j<N; j++)
         aux[i][j] = mat[i][j] + aux[i-1][j];

   // Do row wise sum
   for (int i=0; i<M; i++)
      for (int j=1; j<N; j++)
         aux[i][j] += aux[i][j-1];
}

// A O(1) time function to compute sum of submatrix
// between (tli, tlj) and (rbi, rbj) using aux[][]
// which is built by the preprocess function
int sumQuery(int aux[M][N], int tli, int tlj, int rbi,
                                              int rbj)
{
    // result is now sum of elements between (0, 0) and
    // (rbi, rbj)
    int res = aux[rbi][rbj];

    // Remove elements between (0, 0) and (tli-1, rbj)
    if (tli > 0)
       res = res - aux[tli-1][rbj];

    // Remove elements between (0, 0) and (rbi, tlj-1)
    if (tlj > 0)
       res = res - aux[rbi][tlj-1];

    // Add aux[tli-1][tlj-1] as elements between (0, 0)
    // and (tli-1, tlj-1) are subtracted twice
    if (tli > 0 && tlj > 0)
       res = res + aux[tli-1][tlj-1];

    return res;
}
```

149 Count Negative Numbers in a Column-Wise and Row-Wise Sorted Matrix

http://www.geeksforgeeks.org/count-negative-numbers-in-a-column-wise-row-wise-sorted-matrix/

Find the number of negative numbers in a column-wise / row-wise sorted matrix M[][]. Suppose M has n rows and m columns.
Example:
```
Input:  M = [-3, -2, -1,  1]
            [-2,  2,  3,  4]
            [4,   5,  7,  8]
Output : 4
We have 4 negative numbers in this matrix
```

<u>Here's a naive, non-optimal solution.</u>

We start from the top left corner and count the number of negative numbers one by one, from left to right and top to bottom.
With the given example:
```
[-3, -2, -1,  1]
[-2,  2,  3,  4]
[4,   5,  7,  8]
```

Evaluation process

```
[→,  →,  →,  1]
[→,  2,  3,  4]
[4,  5,  7,  8]
```
Optimal Solution

<u>Here's a more efficient solution:</u>

We start from the top right corner and find the position of the last negative number in the first row.

Using this information, we find the position of the last negative number in the second row.
We keep repeating this process until we either run out of negative numbers or we get to the last row.

150 Construct Ancestor Matrix from a Given Binary Tree

http://www.geeksforgeeks.org/construct-ancestor-matrix-from-a-given-binary-tree/

```
Given a Binary Tree where all values are from 0 to n-1. Construct an ancestor matrix mat[n][n].
Ancestor matrix is defined as below.
mat[i][j] = 1 if i is ancestor of j
mat[i][j] = 0, otherwise
Examples:
Input: Root of below Binary Tree.
      0
    /  \
   1    2
Output: 0 1 1
        0 0 0
        0 0 0

Input: Root of below Binary Tree.
       5
     /   \
    1     2
   / \   /
  0   4 3
Output: 0 0 0 0 0 0
        1 0 0 0 1 0
        0 0 0 1 0 0
        0 0 0 0 0 0
        0 0 0 0 0 0
        1 1 1 1 1 0
```

The **idea** is to traverse the tree. While traversing, keep track of ancestors in an array. When we visit a node, we add it to ancestor array and consider corresponding row in adjacency matrix.

We mark all ancestors in its row as 1. Once a node and all its children are processed, we remove the node from ancestor array.

```
// Creating a global boolean matrix for simplicity
bool mat[MAX][MAX];

// anc[] stores all ancestors of current node.  This
// function fills ancestors for all nodes.
// It also returns size of tree.  Size of tree is
// used to print ancestor matrix.
```

```
int ancestorMatrixRec(Node *root, vector<int> &anc)
{
    /* base case */
    if (root == NULL) return 0;;

    // Update all ancestors of current node
    int data = root->data;
    for (int i=0; i<anc.size(); i++)
       mat[anc[i]][data] = true;

    // Push data to list of ancestors
    anc.push_back(data);

    // Traverse left and right subtrees
    int l = ancestorMatrixRec(root->left, anc);
    int r = ancestorMatrixRec(root->right, anc);

    // Remove data from list the list of ancestors
    // as all descendants of it are processed now.
    anc.pop_back();

    return l+r+1;
}


// This function mainly calls ancestorMatrixRec()
void ancestorMatrix(Node *root)
{
    // Create an empty ancestor array
    vector<int> anc;

    // Fill ancestor matrix and find size of
    // tree.
    int n = ancestorMatrixRec(root, anc);

    // Print the filled values
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<n; j++)
            cout << mat[i][j] << " ";
        cout << endl;
    }
}
```

Time complexity of above solution is O(n^2).

151 Construct tree from ancestor matrix

http://www.geeksforgeeks.org/construct-tree-from-ancestor-matrix/

```
Given an ancestor matrix mat[n][n] where Ancestor matrix is defined as below.
  mat[i][j] = 1 if i is ancestor of j
  mat[i][j] = 0, otherwise
Construct a Binary Tree from given ancestor matrix where all its values of nodes are from 0 to
n-1.
It may be assumed that the input provided the program is valid and tree can be constructed out
of it.
Many Binary trees can be constructed from one input. The program will construct any one of
them.
Examples:
Input: 0 1 1
       0 0 0
       0 0 0
Output: Root of one of the below trees.
    0          0
  /   \   OR  /   \
 1     2     2     1

Input: 0 0 0 0 0 0
       1 0 0 0 1 0
       0 0 0 1 0 0
       0 0 0 0 0 0
       0 0 0 0 0 0
       1 1 1 1 1 0
Output: Root of one of the below trees.
     5          5          5
   /   \       / \        /  \
  1     2  OR  2   1  OR  1    2  OR ....
 / \   /      /   / \    / \   /
0   4 3      3   0   4  4   0 3
```

There are different possible outputs because ancestor
matrix doesn't store that which child is left and which
is right.

Observations used in the solution:
The rows that correspond to leaves have all 0's
The row that corresponds to root has maximum number of 1's.
Count of 1's in i'th row indicates number of descendants of node i.

The idea is to construct the tree in bottom up manner.
1) Create an array of node pointers node[].
2) Store row numbers that correspond to a given count. We have used multimap for this purpose.
3) Process all entries of multimap from smallest count to largest (Note that entries in map and multimap can be traversed in sorted order). Do following for every entry.
…….a) Create a new node for current row number.
…….b) If this node is not a leaf node, consider all those descendants of it whose parent is not set, make current node as its parent.
4) The last processed node (node with maximum sum) is root of tree.

```
// Constructs tree from ancestor matrix
Node* ancestorTree(int mat[][N])
{
    // Binary array to determine weather
    // parent is set for node i or not
    int parent[N] = {0};

    // Root will store the root of the constructed tree
    Node* root = NULL;

    // Create a multimap, sum is used as key and row
    // numbers are used as values
    multimap<int, int> mm;

    for (int i = 0; i < N; i++)
    {
        int sum = 0; // Initialize sum of this row
        for (int j = 0; j < N; j++)
            sum += mat[i][j];

        // insert(sum, i) pairs into the multimap
        mm.insert(pair<int, int>(sum, i));
    }

    // node[i] will store node for i in constructed tree
    Node* node[N];

    // Traverse all entries of multimap.  Note that values
    // are accessed in increasing order of sum
    for (auto it = mm.begin(); it != mm.end(); ++it)
    {
      // create a new node for every value
      node[it->second] = newNode(it->second);

      // To store last processed node. This node will be
```

```
        // root after loop terminates
        root = node[it->second];

        // if non-leaf node
        if (it->first != 0)
        {
          // traverse row 'it->second' in the matrix
          for (int i = 0; i < N; i++)
          {
             // if parent is not set and ancestor exits
             if (!parent[i] && mat[it->second][i])
             {
               // check for unoccupied left/right node
               // and set parent of node i
               if (!node[it->second]->left)
                 node[it->second]->left = node[i];
               else
                 node[it->second]->right = node[i];

               parent[i] = 1;
             }
          }
        }
      }
      return root;
}
```

Note that we can also use an array of vectors in place of multimap. We have used multimap for simplicity. Array of vectors would improve performance as inserting and accessing elements would take O(1) time.

152 Common elements in all rows of a given matrix

http://www.geeksforgeeks.org/common-elements-in-all-rows-of-a-given-matrix/

Given an m x n matrix, find all common elements present in all rows in O(mn) time and one traversal of matrix.

Example:

Input:

```
mat[4][5] = {{1, 2, 1, 4, 8},
          {3, 7, 8, 5, 1},
          {8, 7, 7, 3, 1},
          {8, 1, 2, 7, 9},
         };
```

Output:

1 8 or 8 1

8 and 1 are present in all rows.

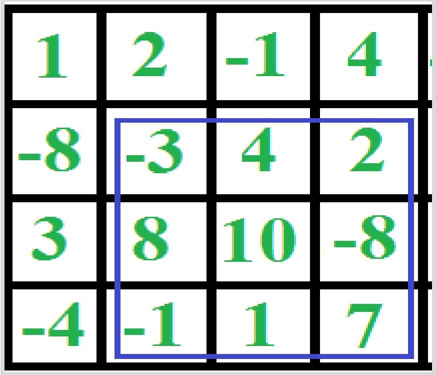
A **better solution** is to sort all rows in the matrix and use similar approach as discussed here. Sorting will take O(mnlogn) time and finding common elements will take O(mn) time. So overall **time complexity** of this solution is O(mnlogn)

<u>Can we do better?</u>
The idea is to use maps. We initially insert all elements of the first row in an map. For every other element in remaining rows, we check if it is present in the map. If it is present in the map and is not duplicated in current row, we increment count of the element in map by 1, else we ignore the element. If the currently traversed row is the last row, we print the element if it has appeared m-1 times before.
The **time complexity** of this solution is O(m * n) and we are doing only one traversal of the matrix.

153 Print maximum sum square sub-matrix of given size

http://www.geeksforgeeks.org/print-maximum-sum-square-sub-matrix-of-given-size/

Given an N x N matrix, find a k x k submatrix where k <= N and k >= 1, such that sum of all the elements in submatrix is maximum. The input matrix can contain zero, positive and negative numbers.
For example consider below matrix, if k = 3, then output should print the sub-matrix enclosed in blue.



1	2	-1	4
-8	-3	4	2
3	8	10	-8
-4	-1	1	7

A **Simple Solution is to consider all possible sub-squares of size k x k in our input matrix and find the one which has maximum sum**. Time complexity of above solution is O(N2k2).

<u>We can solve this problem in O(N2) time.</u>
This problem is mainly an extension of this problem of printing all sums. The idea is to preprocess the given square matrix. In the preprocessing step, calculate sum of all vertical strips of size k x 1 in a temporary square matrix stripSum[][]. Once we have sum of all vertical strips, we can calculate sum of first sub-square in a row as sum of first k strips in that row, and for remaining sub-squares, we can calculate sum in O(1) time by removing the leftmost strip of previous subsquare and adding the rightmost strip of new square.

154 Find a specific pair in Matrix

http://www.geeksforgeeks.org/find-a-specific-pair-in-matrix/

Given an n x n matrix mat[n][n] of integers, find the maximum value of mat(c, d) – mat(a, b) over all choices of indexes such that both c > a and d > b.
Example:
Input:
mat[N][N] = {{ 1, 2, -1, -4, -20 },
{ -8, -3, 4, 2, 1 },
{ 3, 8, 6, 1, 3 },
{ -4, -1, 1, 7, -6 },
{ 0, -4, 10, -5, 1 }};
Output: 18
The maximum value is 18 as mat[4][2]
- mat[1][0] = 18 has maximum difference.
The program should do only ONE traversal of the matrix. i.e. expected time complexity is O(n2)

A **simple solution** would be to apply **Brute-Force**. For all values mat(a, b) in the matrix, we find mat(c, d) that has maximum value such that c > a and d > b and keeps on updating maximum value found so far. We finally return the maximum value.
The above program runs in O(n^4) time which is nowhere close to expected **time complexity** of O(n^2)

An **efficient solution** uses extra space. We pre-process the matrix such that index(i, j) stores max of elements in matrix from (i, j) to (N-1, N-1) and in the process keeps on updating maximum value found so far. We finally return the maximum value.

```
// The function returns maximum value A(c,d) - A(a,b)
// over all choices of indexes such that both c > a
// and d > b.
int findMaxValue(int mat[][N])
{
    //stores maximum value
    int maxValue = INT_MIN;

    // maxArr[i][j] stores max of elements in matrix
    // from (i, j) to (N-1, N-1)
    int maxArr[N][N];

    // last element of maxArr will be same's as of
    // the input matrix
    maxArr[N-1][N-1] = mat[N-1][N-1];

    // preprocess last row
    int maxv = mat[N-1][N-1];  // Initialize max
    for (int j = N - 2; j >= 0; j--)
    {
        if (mat[N-1][j] > maxv)
            maxv = mat[N - 1][j];
        maxArr[N-1][j] = maxv;
    }

    // preprocess last column
    maxv = mat[N - 1][N - 1];  // Initialize max
    for (int i = N - 2; i >= 0; i--)
    {
        if (mat[i][N - 1] > maxv)
            maxv = mat[i][N - 1];
        maxArr[i][N - 1] = maxv;
    }

    // preprocess rest of the matrix from bottom
    for (int i = N-2; i >= 0; i--)
    {
        for (int j = N-2; j >= 0; j--)
        {
            // Update maxValue
            if (maxArr[i+1][j+1] - mat[i][j] >
                                            maxValue)
                maxValue = maxArr[i + 1][j + 1] - mat[i][j];

            // set maxArr (i, j)
            maxArr[i][j] = max(mat[i][j],
                               max(maxArr[i][j + 1],
                                   maxArr[i + 1][j]) );
        }
    }

    return maxValue;
}
```

155 Find orientation of a pattern in a matrix

http://www.geeksforgeeks.org/find-orientation-of-a-pattern-in-a-matrix/

Given a matrix of characters and a pattern, find the orientation of pattern in the matrix. In other words, find if pattern appears in matrix in horizontal or vertical direction. Achieve this in minimum time possible.

Input:

```
mat[N][N] = { {'a', 'b', 'c', 'd', 'e'},
          {'f', 'g', 'h', 'i', 'j'},
          {'k', 'l', 'm', 'n', 'o'},
          {'p', 'q', 'r', 's', 't'},
          {'u', 'v', 'w', 'x', 'y'}};
pattern = "pqrs";

Output: Horizontal
```

A **simple solution** is for each row and column, use Naive pattern searching algorithm to find the orientation of pattern in the matrix. The time complexity of Naive pattern searching algorithm for every row is O(NM) where N is size of the matrix and M is length of the pattern. So, the **time complexity** of this solution will be O(N*(NM)) as each of N rows and N columns takes O(NM) time.

Can we do better?

The idea is to use KMP pattern matching algorithm for each row and column. The KMP matching algorithm improves the worst case to O(N + M). The total cost of a KMP search is linear in the number of characters of string and pattern. For a N x N matrix and pattern of length M, complexity of this solution will be O(N*(N+M)) as each of N rows and N columns will take O(N + M) time.

156 Inplace rotate square matrix by 90 degrees | Set 1

http://www.geeksforgeeks.org/inplace-rotate-square-matrix-by-90-degrees/

```
Given an square matrix, turn it by 90 degrees in anti-clockwise direction without using any
extra space.
Examples:
Input
 1  2  3
 4  5  6
 7  8  9

Output:
 3  6  9
 2  5  8
 1  4  7
```

An **approach** that requires **extra space** is already discussed here.

How to do without extra space?

<u>Below are some important observations</u>.
First row of source –> First column of destination, elements filled in opposite order
Second row of source –> Second column of destination, elements filled in opposite order
so ... on
Last row of source –> Last column of destination, elements filled in opposite order.
An N x N matrix will have floor(N/2) square cycles. For example, a 4 X 4 matrix will have 2 cycles. The first cycle is formed by its 1st row, last column, last row and 1st column. The second cycle is formed by 2nd row, second-last column, second-last row and 2nd column.

The idea is for each square cycle, we swap the elements involved with the corresponding cell in the matrix in anti-clockwise direction i.e. from top to left, left to bottom, bottom to right and from right to top one at a time. We use nothing but a temporary variable to achieve this.

```
// An Inplace function to rotate a N x N matrix
// by 90 degrees in anti-clockwise direction
void rotateMatrix(int mat[][N])
{
    // Consider all squares one by one
    for (int x = 0; x < N / 2; x++)
    {
        // Consider elements in group of 4 in
        // current square
        for (int y = x; y < N-x-1; y++)
        {
            // store current cell in temp variable
            int temp = mat[x][y];

            // move values from right to top
            mat[x][y] = mat[y][N-1-x];

            // move values from bottom to right
            mat[y][N-1-x] = mat[N-1-x][N-1-y];

            // move values from left to bottom
            mat[N-1-x][N-1-y] = mat[N-1-y][x];

            // assign temp to left
            mat[N-1-y][x] = temp;
        }
    }
}
```

Exercise: Turn 2D matrix by 90 degrees in clockwise direction without using extra space.

157 Rotate a matrix by 90 degree without using any extra space | Set 2

http://www.geeksforgeeks.org/rotate-matrix-90-degree-without-using-extra-space-set-2/

Given a square matrix, turn it by 90 degrees in anti-clockwise direction without using any extra space.

In this post another approach is discussed which is much simpler than the above approach. There are two steps :

Find transpose of matrix.
Reverse columns of the transpose.

Illustration of above steps :

```
Let the given matrix be
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16

First we find transpose.
1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16
Then we reverse elements of every column.
4 8 12 16
3 7 11 15
2 6 10 14
1 5  9 13
```

Time complexity :O(R*C)
Space complexity :O(1)

The above steps/program do left (or anticlockwise) rotation, how to right (or clockwise) rotate?

To right rotate, we do following steps.

Find transpose of matrix.
Reverse rows of the transpose.

Illustration of above steps :

```
Let the given matrix be
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
First we find transpose.
1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16
Then we reverse elements of every row.
13  9 5 1
14 10 6 2
15 11 7 3
16 12 8 4
```

158 Minimum Initial Points to Reach Destination

http://www.geeksforgeeks.org/minimum-positive-points-to-reach-destination/

Given a grid with each cell consisting of positive, negative or no points i.e, zero points. We can move across a cell only if we have positive points (> 0). Whenever we pass through a cell, points in that cell are added to our overall points. We need to find minimum initial points to reach cell (m-1, n-1) from (0, 0).
Constraints :
From a cell (i, j) we can move to (i+1, j) or (i, j+1).
We cannot move from (i, j) if your overall points at (i, j) is <= 0.
We have to reach at (n-1, m-1) with minimum positive points i.e., > 0.
Example:

```
Input: points[m][n] = { {-2, -3,   3},
                  {-5, -10,  1},
                  {10,  30, -5}
                 };
Output: 7
Explanation:
7 is the minimum value to reach destination with
positive throughout the path. Below is the path.

(0,0) -> (0,1) -> (0,2) -> (1, 2) -> (2, 2)
```

We start from (0, 0) with 7, we reach(0, 1)
with 5, (0, 2) with 2, (1, 2) with 5, (2, 2)
with and finally we have 1 point (we needed
greater than 0 points at the end).
At the **first look**, this problem **looks similar Max/Min Cost Path**, but maximum overall points gained will not guarantee the minimum initial points. Also, it is compulsory in the current problem that the points never drops to zero or below. For instance, Suppose following two paths exists from source to destination cell.

We can solve this problem through bottom-up table filling dynamic programing technique.
To begin with, we should maintain a 2D array dp of the same size as the grid, where dp[i][j] represents the minimum points that guarantees the continuation of the journey to destination before entering the cell (i, j). It's but obvious that dp[0][0] is our final solution. Hence, for this problem, we need to fill the table from the bottom right corner to left top.

Now, let us decide minimum points needed to leave cell (i, j) (remember we are moving from bottom to up). There are only two paths to choose: (i+1, j) and (i, j+1). Of course we will choose the cell that the player can finish the rest of his journey with a smaller initial points. Therefore we have: min_Points_on_exit = min(dp[i+1][j], dp[i][j+1])
Now we know how to compute min_Points_on_exit, but we need to fill the table dp[][] to get the solution in dp[0][0].

How to compute dp[i][j]?
The value of dp[i][j] can be written as below.
dp[i][j] = max(min_Points_on_exit – points[i][j], 1)

Let us see how above expression covers all cases.
If points[i][j] == 0, then nothing is gained in this cell; the player can leave the cell with the same points as he enters the room with, i.e. dp[i][j] = min_Points_on_exit.
If dp[i][j] < 0, then the player must have points greater than min_Points_on_exit before entering (i, j) in order to compensate for the points lost in this cell. The minimum amount of compensation is " - points[i][j] ", so we have dp[i][j] = min_Points_on_exit - points[i][j].
If dp[i][j] > 0, then the player could enter (i, j) with points as little as min_Points_on_exit – points[i][j].

since he could gain "points[i][j]" points in this cell. However, the value of min_Points_on_exit – points[i][j] might drop to 0 or below in this situation. When this happens, we must clip the value to 1 in order to make sure dp[i][j] stays positive:
dp[i][j] = max(min_Points_on_exit – points[i][j], 1).

Finally return dp[0][0] which is our answer.

```
int minInitialPoints(int points[][C])
{
    // dp[i][j] represents the minimum initial points player
    // should have so that when starts with cell(i, j) successfully
    // reaches the destination cell(m-1, n-1)
    int dp[R][C];
    int m = R, n = C;

    // Base case
    dp[m-1][n-1] = points[m-1][n-1] > 0? 1:
                   abs(points[m-1][n-1]) + 1;

    // Fill last row and last column as base to fill
    // entire table
    for (int i = m-2; i >= 0; i--)
         dp[i][n-1] = max(dp[i+1][n-1] - points[i][n-1], 1);
    for (int j = n-2; j >= 0; j--)
         dp[m-1][j] = max(dp[m-1][j+1] - points[m-1][j], 1);

    // fill the table in bottom-up fashion
    for (int i=m-2; i>=0; i--)
    {
        for (int j=n-2; j>=0; j--)
        {
            int min_points_on_exit = min(dp[i+1][j], dp[i][j+1]);
            dp[i][j] = max(min_points_on_exit - points[i][j], 1);
        }
     }

     return dp[0][0];
}
```

159 Inplace (Fixed space) M x N size matrix transpose | Updated

http://www.geeksforgeeks.org/inplace-m-x-n-size-matrix-transpose/

About four months of gap (missing GFG), a new post. Given an M x N matrix, transpose the matrix without auxiliary memory.It is easy to transpose matrix using an auxiliary array. If the matrix is symmetric in size, we can transpose the matrix inplace by mirroring the 2D array across it's diagonal (try yourself). How to transpose an arbitrary size matrix inplace? See the following matrix,

```
a b c     a d g j
d e f ==> b e h k
g h i     c f i l
j k l
```

160 Shortest path in a Binary Maze

http://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/

Given a MxN matrix where each element can either be 0 or 1. We need to find the shortest path between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.
Expected time complexity is O(MN).
For example –
Input:

```
mat[ROW][COL]  = {{1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
                  {1, 0, 1, 0, 1, 1, 1, 0, 1, 1 },
                  {1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },
                  {0, 0, 0, 0, 1, 0, 0, 0, 0, 1 },
                  {1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },
                  {1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },
                  {1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
                  {1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
                  {1, 1, 0, 0, 0, 0, 1, 0, 0, 1 }};
Source = {0, 0};
Destination = {3, 4};

Output:
Shortest Path is 11
```

The **idea** is inspired from Lee algorithm and uses BFS.

We start from the source cell and calls BFS procedure.
We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.
We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.
We LOOP till queue is not empty
Dequeue front cell from the queue
Return if the destination coordinates have reached.
For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited.

```
// function to find the shortest path between
// a given source cell to a destination cell.
int BFS(int mat[][COL], Point src, Point dest)
{
    // check source and destination cell
    // of the matrix have value 1
    if (!mat[src.x][src.y] || !mat[dest.x][dest.y])
        return INT_MAX;

    bool visited[ROW][COL];
    memset(visited, false, sizeof visited);

    // Mark the source cell as visited
    visited[src.x][src.y] = true;

    // Create a queue for BFS
    queue<queueNode> q;

    // distance of source cell is 0
    queueNode s = {src, 0};
    q.push(s);  // Enqueue source cell

    // Do a BFS starting from source cell
    while (!q.empty())
    {
        queueNode curr = q.front();
        Point pt = curr.pt;

        // If we have reached the destination cell,
        // we are done
        if (pt.x == dest.x && pt.y == dest.y)
            return curr.dist;

        // Otherwise dequeue the front cell in the queue
        // and enqueue its adjacent cells
        q.pop();

        for (int i = 0; i < 4; i++)
        {
            int row = pt.x + rowNum[i];
            int col = pt.y + colNum[i];

            // if adjacent cell is valid, has path and
            // not visited yet, enqueue it.
            if (isValid(row, col) && mat[row][col] &&
               !visited[row][col])
            {
                // mark cell as visited and enqueue it
                visited[row][col] = true;
                queueNode Adjcell = { {row, col},
                                      curr.dist + 1 };
                q.push(Adjcell);
            }
        }
    }

    //return -1 if destination cannot be reached
    return INT_MAX;
}
```

Reference: GeeksforGeeks