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

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

METHOD 1 (Use Sorting)

Algorithm:

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

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

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

Sort the array
A = {-8, 1, 4, 6, 10, 45}

Initialize l = 0, r = 5
A[l] + A[r] (-8 + 45) > 16 => decrement r. Now r = 10
A[l] + A[r] (-8 + 10) < 2 => increment l. Now l = 1
A[l] + A[r] (1 + 10) < 16 => increment l. Now l = 2

A[l] + A[r] (4 + 10) < 14 => increment l. Now l = 3

A[l] + A[r] (6 + 10) == 16 => Found candidates (return 1)

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

Implementation:

```
# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

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

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

    if( hasArrayTwoCandidates(A, arr_size, n))
        printf("Array has two elements with sum 16");
    else
        printf("Array doesn't have two elements with sum 16 ");

    getchar();
    return 0;
}
```

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

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

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

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

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]]

Implementation:

```
#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
  int i, temp;
  bool binMap[MAX] = {0}; /*initialize hash map as 0*/

  for(i = 0; i < arr_size; i++)
  {
    temp = sum - arr[i];
    if(temp >= 0 && binMap[temp] == 1)
    {
      printf("Pair with given sum %d is (%d, %d) \n", sum, arr[i], temp);
    }
    binMap[arr[i]] = 1;
  }
}

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

    printPairs(A, arr_size, n);

    getchar();
    return 0;
}
```

Time Complexity: O(n)
Auxiliary Space: O(R) where R is range of integers.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is scale everything with reference to the smallest negative integer in the given range.

2.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)
Thanks to Sachin Midha for suggesting this solution.

Node of the Binary Search Tree (used in this approach) will be as follows.

```
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.

```
findCandidate(a[], size)
1.  Initialize index and count of majority element
     maj_index = 0, count = 1
2.  Loop for i = 1 to size – 1
    (a)If a[maj_index] == a[i]
       count++
    (b)Else
       count--;
    (c)If count == 0
       maj_index = i;
       count = 1
3.  Return a[maj_index]
```

Above algorithm loops through each element and maintains a count of a[maj_index], If next element is same then increments the count, if next element is not same then decrements the count, and if the count reaches 0 then changes the maj_index to the current element and sets count to 1.
First Phase algorithm gives us a candidate element. In second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

Example:
A[] = 2, 2, 3, 5, 2, 2, 6
Initialize:
maj_index = 0, count = 1 –> candidate ‘2?
2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 2 => maj_index = 4

2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

Finally candidate for majority element is 2.

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

2. *Check if the element obtained in step 1 is majority*

```
printMajority (a[], size)
1.  Find the candidate for majority
2.  If candidate is majority. i.e., appears more than n/2 times.
       Print the candidate
3.  Else
       Print "NONE"
```

Implementation of method 3:

```
/* Program for finding out majority element in an array */
# include<stdio.h>
# define bool int

int findCandidate(int *, int);
bool isMajority(int *, int, int);
```

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

  /* Print the candidate if it is Majority*/
  if(isMajority(a, size, cand))
    printf(" %d ", cand);
  else
    printf("NO 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;
}

/* Driver function to test above functions */
int main()
{
    int a[] = {1, 3, 3, 1, 2};
    printMajority(a, 5);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Auxiliary Space : O(1)

3. 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

Algorithm:

Do bitwise XOR of all the elements. Finally we get the number which has odd occurrences.

Program:

```
#include <stdio.h>

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;
}

/* Diver function to test above function */
int main()
{
     int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
     printf("%d", getOddOccurrence(ar, 13));
     getchar();
}
```

Time Complexity: O(n)

4. 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:

```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_ending_here < 0)
            max_ending_here = 0
  (c) if(max_so_far < max_ending_here)
            max_so_far = max_ending_here
return max_so_far
```

Explanation:
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

Lets take the example:
{-2, -3, 4, -1, -2, 1, 5, -3}

max_so_far = max_ending_here = 0

for i=0, a[0] = -2
max_ending_here = max_ending_here + (-2)
Set max_ending_here = 0 because max_ending_here < 0

for i=1, a[1] = -3
max_ending_here = max_ending_here + (-3)
Set max_ending_here = 0 because max_ending_here < 0

for i=2, a[2] = 4
max_ending_here = max_ending_here + (4)
max_ending_here = 4

max_so_far is updated to 4 because max_ending_here greater than max_so_far which was 0 till now

for i=3, a[3] = -1
max_ending_here = max_ending_here + (-1)
max_ending_here = 3

for i=4, a[4] = -2
max_ending_here = max_ending_here + (-2)
max_ending_here = 1

for i=5, a[5] = 1
max_ending_here = max_ending_here + (1)
max_ending_here = 2

for i=6, a[6] = 5
max_ending_here = max_ending_here + (5)
max_ending_here = 7
max_so_far is updated to 7 because max_ending_here is greater than max_so_far

for i=7, a[7] = -3
max_ending_here = max_ending_here + (-3)
max_ending_here = 4

Program:

```
#include<stdio.h>
int maxSubArraySum(int a[], int size)
{
   int max_so_far = 0, max_ending_here = 0;
   int i;
   for(i = 0; i < size; i++)
   {
     max_ending_here = max_ending_here + a[i];
     if(max_ending_here < 0)
        max_ending_here = 0;
     if(max_so_far < max_ending_here)
        max_so_far = max_ending_here;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
   int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
   int max_sum = maxSubArraySum(a, 8);
   printf("Maximum contiguous sum is %d\n", max_sum);
   getchar();
   return 0;
}
```

Notes:
Algorithm doesn't work for all negative numbers. It simply returns 0 if all numbers are negative. For handling this we can add an extra phase before actual implementation. The phase will look if all numbers are negative, if they are it will return maximum of them (or smallest in terms of absolute value). There may be other ways to handle it though.

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;
   int i;
   for(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

5. 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.
```

Program:

```
#include<stdio.h>

/* getMissingNo takes array and size of array as arguments*/
int getMissingNo (int a[], int n)
{
    int i, total;
    total  = (n+1)*(n+2)/2;
    for ( i = 0; i< n; i++)
       total -= a[i];
    return total;
}

/*program to test above function */
int main()
{
    int a[] = {1,2,4,5,6};
    int miss = getMissingNo(a,5);
    printf("%d", miss);
    getchar();
}
```

Time Complexity: O(n)

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.
```

```
#include<stdio.h>

/* getMissingNo takes array and size of array as arguments*/
int getMissingNo(int a[], int n)
{
    int i;
    int x1 = a[0]; /* For xor of all the elemets in arary */
    int x2 = 1; /* For xor of all the elemets 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);
}

/*program to test above function */
int main()
{
    int a[] = {1, 2, 4, 5, 6};
    int miss = getMissingNo(a, 5);
    printf("%d", miss);
    getchar();
}
```

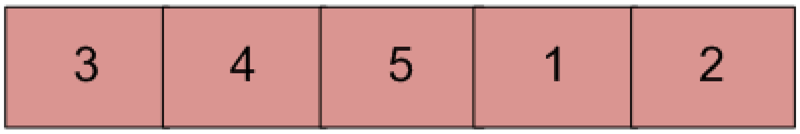
Time Complexity: O(n)

In method 1, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Method 2 has no such problems

6. Search an element in a sorted and pivoted array

Question:

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



Solution:

Algorithm:

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

```
Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
 1) Find out pivot point and divide the array in two
    sub-arrays. (pivot = 2) /*Index of 5*/
 2) Now call binary search for one of the two sub-arrays.
    (a) If element is greater than 0th element then
          search in left array
    (b) Else Search in right array
        (1 will go in else as 1 < 0th element(3))
 3) If element is found in selected sub-array then return index
    Else return -1.
```

Implementation:

```
/*Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element no in a pivoted sorted array arrp[]
   of size arr_size */
int pivotedBinarySearch(int arr[], int arr_size, int no)
{
   int pivot = findPivot(arr, 0, arr_size-1);
   if(arr[pivot] == no)
     return pivot;
   if(arr[0] <= no)
     return binarySearch(arr, 0, pivot-1, no);
   else
     return binarySearch(arr, pivot+1, arr_size-1, no);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2
   it will return 3 */
int findPivot(int arr[], int low, int high)
{
   int mid = (low + high)/2;   /*low + (high - low)/2;*/
   if(arr[mid] > arr[mid + 1])
     return mid;
   if(arr[low] > arr[mid])
     return findPivot(arr, low, mid-1);
   else
```

```
      return findPivot(arr, mid + 1, high);
}

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int no)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/

    if(no == arr[mid])
      return mid;
    if(no > arr[mid])
      return binarySearch(arr, (mid + 1), high, no);
    else
      return binarySearch(arr, low, (mid -1), no);
  }
  /*Return -1 if element is not found*/
  return -1;
}

/* Driver program to check above functions */
int main()
{
   int arr[10] = {3, 4, 5, 6, 7, 1, 2};
   int n = 5;
   printf("Index of the element is %d", pivotedBinarySearch(arr, 7, 5));
   getchar();
   return 0;
}
```

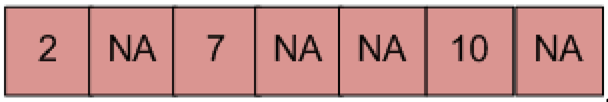
Time Complexity O(logn)

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

Question:

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.

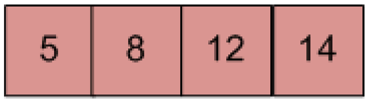
Input: array with m+n elements (mPlusN[]).



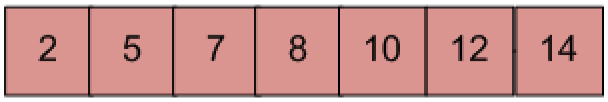
NA => Value is not filled/available in array mPlusN[].

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[].
```

Implementation:

```
/* Assuming -1 is filled for the places where element
   is not available */
#define NA -1

/* Function to move m elements at the end of array
   mPlusN[] */
void moveToEnd(int mPlusN[], int size)
{
  int i = 0, j = size - 1;
  for (i = size-1; i >= 0; i--)
    if(mPlusN[i] != NA)
    {
      mPlusN[j] = mPlusN[i];
      j--;
    }
}

/* Merges array N[] of size n into array mPlusN[]
   of size m+n*/
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 the element from mPlusN[] if
       a) its value 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
    {
      mPlusN[k] = N[j];
      k++;
      j++;
    }
  }
}

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

  printf("\n");
}

/* Driver function to test above functions */
int main()
{
  /* Initialize arrays */
  int mPlusN[9] = {2, 8, NA, NA, NA, 13, NA, 15, 20};
  int N[] = {5, 7, 9, 25};
  int m = 5, n = 4;

  /*Move the m elements at the end of mPlusN*/
  moveToEnd(mPlusN, 9);

  /*Merge N[] into mPlusN[] */
  merge(mPlusN, N, 5, 4);

  /* Print the resultant mPlusN */
  printArray(mPlusN, 9);
  getchar();
}
```

Time Complexity: O(m+n)

8. Median of two sorted arrays

Question: 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))

Median: In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half.
The median of a finite list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one.

For getting the median of input array { 12, 11, 15, 10, 20 }, first sort the array. We get { 10, 11, 12, 15, 20 } after sorting. Median is the middle element of the sorted array which is 12.

There are different conventions to take median of an array with even number of elements, one can take the mean of the two middle values, or first middle value, or second middle value.

Let us see different methods to get the median of two sorted arrays of size n each. Since size of the set for which we are looking for median is even (2n), we are taking average of two middle two numbers in all below solutions.

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. See the below implementation.

Implementation:

```
#include <stdio.h>

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
  int i = 0;  /* Current index of i/p array ar1[] */
  int j = 0; /* Current index of i/p array ar2[] */
  int count;
  int m1 = -1, m2 = -1;

  /* Since there are 2n elements, median will be average
```

```
    of elements at index n-1 and n in the array obtained after
    merging ar1 and ar2 */
   for(count = 0; count <= n; count++)
   {
     /*Below is to handle case where all elements of ar1[] are
       smaller than smallest(or first) element of ar2[]*/
     if(i == n)
     {
       m1 = m2;
       m2 = ar2[0];
       break;
     }

     /*Below is to handle case where all elements of ar2[] are
       smaller than smallest(or first) element of ar1[]*/
     else if(j == n)
     {
       m1 = m2;
       m2 = ar1[0];
       break;
     }

     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;
}

/* Driver program to test above function */
int main()
{
   int ar1[] = {1, 12, 15, 26, 38};
   int ar2[] = {2, 13, 17, 30, 45};

   printf("%d", getMedian(ar1, ar2, 5)) ;

   getchar();
   return 0;
}
```

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

Implementation:

```
#include<stdio.h>

int max(int, int); /* to get maximum of two integers */
int min(int, int); /* to get minimum of two integeres */
int median(int [], int); /* to get median of a single array */

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
  int m1; /* For median of ar1 */
  int m2; /* For median of ar2 */

  /* return -1  for invalid input */
  if(n <= 0)
    return -1;

  if(n == 1)
    return (ar1[0] + ar2[0])/2;

  if (n == 2)
    return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;

  m1 = median(ar1, n); /* get the median of the first array */
  m2 = median(ar2, n); /* get the median of the second array */

  /* If medians are equal then return either m1 or m2 */
  if(m1 == m2)
    return m1;

  /* if m1 < m2 then median must exist in ar1[m1....] and ar2[....m2] */
  if (m1 < m2)
    return getMedian(ar1 + n/2, ar2, n - n/2);

  /* if m1 > m2 then median must exist in ar1[....m1] and ar2[m2...] */
  return getMedian(ar2 + n/2, ar1, n - n/2);
}

/* Driver program to test above function */
int main()
{
  int ar1[] = {1, 12, 15, 26, 38};
  int ar2[] = {2, 13, 17, 30, 45};
  printf("%d", getMedian(ar1, ar2, 5)) ;

  getchar();
  return 0;
}

/* Utility functions */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x > y? y : x;
}

/* Function to get median of a single array */
int median(int arr[], int n)
{
  if(n%2 == 0)
    return (arr[n/2] + arr[n/2-1])/2;
  else
    return arr[n/2];
}
```

Time Complexity: O(logn)

Algorithmic Paradigm: Divide and Conquer

Method 3 (By doing binary search for the median):

The basic idea is that if you are given two arrays ar1[] and ar2[] and know the length of each, you can check whether an element ar1[i] is the median in constant time. Suppose that the median is ar1[i]. Since the array is sorted, it is greater than exactly i-1 values in array ar1[]. Then if it is the median, it is also greater than exactly j = n – i – 1 elements in ar2[].

It requires constant time to check if ar2[j] <= ar1[i] <= ar2[j + 1]. If ar1[i] is not the median, then depending on whether ar1[i] is greater or less than ar2[j] and ar2[j + 1], you know that ar1[i] is either greater than or less than the median. Thus you can binary search for median in O(lg n) worst-case time.

For two arrays ar1 and ar2, first do binary search in ar1[]. If you reach at the end (left or right) of the first array and don't find median, start searching in the second array ar2[].

```
1) Get the middle element of ar1[] using array indexes left and right.
  Let index of the middle element be i.
2) Calculate the corresponding index j of ar2[]
    j = n – i – 1
3) If ar1[i] >= ar2[j] and ar1[i] <= ar2[j+1] then ar1[i] and ar2[j]
  are the middle elements.
    return average of ar2[j] and ar1[i]
4) If ar1[i] is greater than both ar2[j] and ar2[j+1] then
    do binary search in left half  (i.e., arr[left ... i-1])
5) If ar1[i] is smaller than both ar2[j] and ar2[j+1] then
    do binary search in right half (i.e., arr[i+1....right])
6) If you reach at any corner of ar1[] then do binary search in ar2[]
```

Example:

```
  ar1[] = {1, 5, 7, 10, 13}
  ar2[] = {11, 15, 23, 30, 45}
```

Middle element of ar1[] is 7. Let us compare 7 with 23 and 30, since 7 smaller than both 23 and 30, move to right in ar1[]. Do binary search in {10, 13}, this step will pick 10. Now compare 10 with 15 and 23. Since 10 is smaller than both 15 and 23, again move to right. Only 13 is there in right side now. Since 13 is greater than 11 and smaller than 15, terminate here. We have got the median as 12 (average of 11 and 13)

Implementation:

```
#include<stdio.h>

int getMedianRec(int ar1[], int ar2[], int left, int right, int n);

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
  return getMedianRec(ar1, ar2, 0, n-1, n);
}
```

```
 /* A recursive function to get the median of ar1[] and ar2[]
    using binary search */
int getMedianRec(int ar1[], int ar2[], int left, int right, int n)
{
  int i, j;

  /* We have reached at the end (left or right) of ar1[] */
  if(left > right)
    return getMedianRec(ar2, ar1, 0, n-1, n);

  i = (left + right)/2;
  j = n - i - 1;  /* Index of ar2[] */

 /* Recursion terminates here.*/
  if(ar1[i] > ar2[j] && (j == n-1 || ar1[i] <= ar2[j+1]))
  {
    /*ar1[i] is decided as median 2, now select the median 1
       (element just before ar1[i] in merged array) to get the
       average of both*/
     if(ar2[j] > ar1[i-1] || i == 0)
       return (ar1[i] + ar2[j])/2;
     else
       return (ar1[i] + ar1[i-1])/2;
  }

  /*Search in left half of ar1[]*/
  else if (ar1[i] > ar2[j] && j != n-1 && ar1[i] > ar2[j+1])
    return getMedianRec(ar1, ar2, left, i-1, n);

  /*Search in right half of ar1[]*/
  else /* ar1[i] is smaller than both ar2[j] and ar2[j+1]*/
    return getMedianRec(ar1, ar2, i+1, right, n);
}

/* Driver program to test above function */
int main()
{
  int ar1[] = {1, 12, 15, 26, 38};
  int ar2[] = {2, 13, 17, 30, 45};
  printf("%d", getMedian(ar1, ar2, 5)) ;

  getchar();
  return 0;
}
```

Time Complexity: O(logn)

Algorithmic Paradigm: Divide and Conquer

9. Write a program to reverse an array

Iterative way:

1) Initialize start and end indexes.

start = 0, end = n-1

2) In a loop, swap arr[start] with arr[end] and change start and end as follows.

start = start +1; end = end – 1

```
/* Function to reverse arr[] from start to end*/
void rvereseArray(int arr[], int start, int end)
{
  int i;
  int temp;
  while(start < end)
  {
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    start++;
    end--;
  }
}

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

  printf("\n");
}

/* Driver function to test above functions */
int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6};
  printArray(arr, 6);
  rvereseArray(arr, 0, 5);
  printf("Reversed array is \n");
  printArray(arr, 6);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Recursive Way:

1) Initialize start and end indexes

start = 0, end = n-1

2) Swap arr[start] with arr[end]

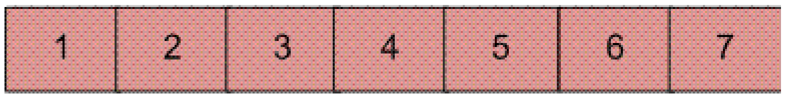
3) Recursively call reverse for rest of the array.

```
/* Function to reverse arr[] from start to end*/
void rvereseArray(int arr[], int start, int end)
{
   int temp;
   if(start >= end)
     return;
   temp = arr[start];
   arr[start] = arr[end];
   arr[end] = temp;
   rvereseArray(arr, start+1, end-1);
}

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

  printf("\n");
}

/* Driver function to test above functions */
int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  printArray(arr, 5);
  rvereseArray(arr, 0, 4);
  printf("Reversed array is \n");
  printArray(arr, 5);
  getchar();
  return 0;
}
```

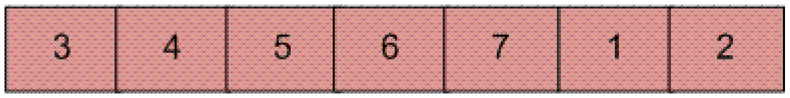
Time Complexity: O(n)

10. Program for array rotation

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



Rotation of the above array by 2 will make array



METHOD 1 (Use temp array)

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

Time complexity O(n)

Auxiliary Space: O(d)

METHOD 2 (Rotate one by one)

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

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

Let us take the same example arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2

Rotate arr[] by one 2 times

We get [2, 3, 4, 5, 6, 7, 1] after first rotation and [3, 4, 5, 6, 7, 1, 2] after second rotation.

```
/*Function to left Rotate arr[] of size n by 1*/
void leftRotatebyOne(int arr[], int n);

/*Function to left rotate arr[] of size n by d*/
```

```
void leftRotate(int arr[], int d, int n)
{
  int i;
  for (i = 0; i < d; i++)
    leftRotatebyOne(arr, n);
}

void leftRotatebyOne(int arr[], int n)
{
  int i, temp;
  temp = arr[0];
  for (i = 0; i < n-1; i++)
     arr[i] = arr[i+1];
  arr[i] = temp;
}

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

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

Time complexity: O(n*d)

Auxiliary Space: O(1)

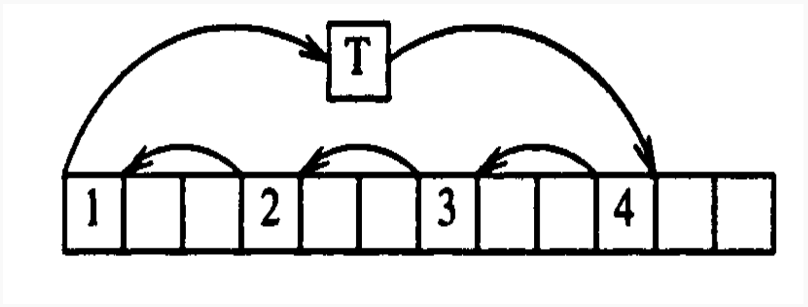
METHOD 3 (A Juggling Algorithm)

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

Here is an example for n =12 and d = 3. GCD is 3 and

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

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



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

b) Then in second set.

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

c) Finally in third set.

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

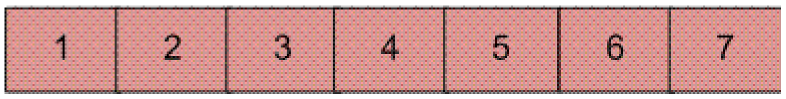
```
/* function to print an array */
void printArray(int arr[], int size);

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

/*Function to left rotate arr[] of siz n by d*/
void leftRotate(int arr[], int d, int n)
{
  int i, j, k, temp;
  for (i = 0; i < gcd(d, n); i++)
  {
    /* move i-th values of blocks */
    temp = arr[i];
    j = i;
    while(1)
    {
      k = j + d;
      if (k >= n)
        k = k - n;
      if (k == i)
        break;
      arr[j] = arr[k];
      j = k;
    }
    arr[j] = temp;
  }
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
  int i;
  for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
}

/*Fuction to get gcd of a and b*/
int gcd(int a,int b)
{
   if(b==0)
     return a;
   else
     return gcd(b, a%b);
}

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

Time complexity: O(n)

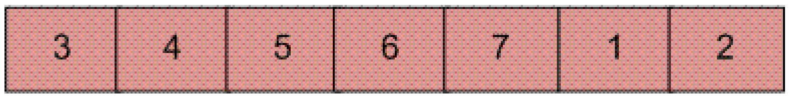
Auxiliary Space: O(1)

11. Reversal algorithm for array rotation

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



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);
```

Let AB are the two parts of the input array where A = arr[0..d-1] and B = arr[d..n-1]. The idea of the algorithm is:

Reverse A to get ArB. /* Ar is reverse of A */

Reverse B to get ArBr. /* Br is reverse of B */

Reverse all to get (ArBr) r = BA.

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

A = [1, 2] and B = [3, 4, 5, 6, 7]

Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7]

Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]

Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

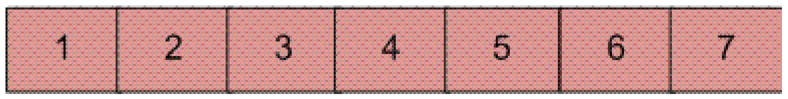
Implementation:

```
/*Utility function to print an array */
void printArray(int arr[], int size);

/* Utility function to reverse arr[] from start to end */
void rvereseArray(int arr[], int start, int end);

/* Function to left rotate arr[] of size n by d */
void leftRotate(int arr[], int d, int n)
{
  rvereseArray(arr, 0, d-1);
  rvereseArray(arr, d, n-1);
  rvereseArray(arr, 0, n-1);
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
  int i;
  for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("%\n ");
}

/*Function to reverse arr[] from index start to end*/
void rvereseArray(int arr[], int start, int end)
{
  int i;
  int temp;
  while(start < end)
  {
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    start++;
    end--;
  }
}

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

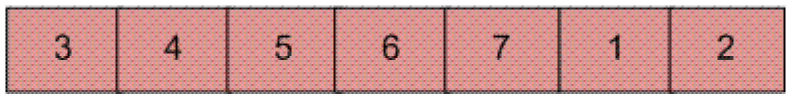
Time Complexity: O(n)

12. Block swap algorithm for array rotation

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



Rotation of the above array by 2 will make array



Algorithm:

Initialize A = arr[0..d-1] and B = arr[d..n-1]

1) Do following until size of A is equal to size of B

  a) If A is shorter, divide B into Bl and Br such that Br is of same length as A. Swap A and Br to change ABlBr into BrBlA. Now A is at its final place, so recur on pieces of B.

  b) If A is longer, divide A into Al and Ar such that Al is of same length as B Swap Al and B to change AlArB into BArAl. Now B is at its final place, so recur on pieces of A.

2) Finally when A and B are of equal size, block swap them.

Recursive Implementation:

```
#include<stdio.h>

/*Prototype for utility functions */
void printArray(int arr[], int size);
void swap(int arr[], int fi, int si, int d);

void leftRotate(int arr[], int d, int n)
{
  /* Return If number of elements to be rotated is
    zero or equal to array size */
  if(d == 0 || d == n)
    return;

  /*If number of elements to be rotated is exactly
    half of array size */
  if(n-d == d)
  {
    swap(arr, 0, n-d, d);
    return;
  }

 /* If A is shorter*/
  if(d < n-d)
  {
    swap(arr, 0, n-d, d);
    leftRotate(arr, d, n-d);
  }
  else /* If B is shorter*/
  {
    swap(arr, 0, d, n-d);
    leftRotate(arr+n-d, 2*d-n, d); /*This is tricky*/
  }
}
```

```
/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
  int i;
  for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("%\n ");
}

/*This function swaps d elements starting at index fi
  with d elements starting at index si */
void swap(int arr[], int fi, int si, int d)
{
   int i, temp;
   for(i = 0; i<d; i++)
   {
     temp = arr[fi + i];
     arr[fi + i] = arr[si + i];
     arr[si + i] = temp;
   }
}

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

Iterative Implementation:

Here is iterative implementation of the same algorithm. Same utility function swap() is used here.

```
void leftRotate(int arr[], int d, int n)
{
  int i, j;
  if(d == 0 || d == n)
    return;
  i = d;
  j = n - d;
  while (i != j)
  {
    if(i < j) /*A is shorter*/
    {
      swap(arr, d-i, d+j-i, i);
      j -= i;
    }
    else /*B is shorter*/
    {
      swap(arr, d-i, d, j);
      i -= j;
    }
    // printArray(arr, 7);
  }
  /*Finally, block swap A and B*/
  swap(arr, d-i, d, i);
}
```

Time Complexity: O(n)

13. Maximum sum such that no two elements are adjacent

Question: Given an array all of whose elements are 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.

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).

At the end of the loop return max of incl and excl.

Example:

```
arr[] = {5,  5, 10, 40, 50, 35}

inc = 5
exc = 0

For i = 1 (current element is 5)
incl =  (excl + arr[i])  = 5
excl =  max(5, 0) = 5

For i = 2 (current element is 10)
incl =  (excl + arr[i]) = 15
```

excl = max(5, 5) = 5

For i = 3 (current element is 40)

incl = (excl + arr[i]) = 45

excl = max(5, 15) = 15

For i = 4 (current element is 50)

incl = (excl + arr[i]) = 65

excl = max(45, 15) = 45

For i = 5 (current element is 35)

incl = (excl + arr[i]) = 80

excl = max(5, 15) = 65

And 35 is the last element. So, answer is max(incl, excl) = 80

Implementation:

```
#include&lt;stdio.h&gt;

/*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;
  int i;

  for (i = 1; i &lt; n; i++)
  {
     /* current max excluding i */
     excl_new = (incl &gt; excl)? incl: excl;

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

   /* return max of incl and excl */
   return ((incl &gt; excl)? incl : excl);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {5, 5, 10, 100, 10, 5};
  printf(&quot;%d \n&quot;, FindMaxSum(arr, 6));
  getchar();
  return 0;
}
```

Time Complexity: O(n)

14. 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.

```
/*Function to print leaders in an array */
void printLeaders(int arr[], int size)
{
  int i, j;

  for (i = 0; i < size; i++)
  {
    for (j = i+1; j < size; j++)
    {
        if(arr[i] < arr[j])
          break;
    }
    if(j == size) // the loop didn't break
    {
        printf("%d ", arr[i]);
    }
  }
}

/*Driver program to test above function*/
int main()
{
  int arr[] = {16, 17, 4, 3, 5, 2};
  printLeaders(arr, 6);
  getchar();
}
// Output:  17 5 2
```

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.

```
/*Function to print leaders in an array */
void printLeaders(int arr[], int size)
{
  int max_from_right =  arr[size-1];
  int i;

  /* Rightmost element is always leader */
  printf("%d ", max_from_right);

  for(i = size-2; i >= 0; i--)
  {
    if(max_from_right < arr[i])
    {
       printf("%d ", arr[i]);
       max_from_right = arr[i];
    }
  }
}

/*Driver program to test above function*/
int main()
{
  int arr[] = {16, 17, 4, 3, 5, 2};
  printLeaders(arr, 6);
  getchar();
}
// Output:  2 5 17
```

Time Complexity: O(n)

15. Sort elements by frequency

Question:

Print the elements of an array in the decreasing frequency if 2 numbers have same frequency then print the one which came 1st

E.g. 2 5 2 8 5 6 8 8 output: 8 8 8 2 2 5 5 6.

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
```

There is one issue with above approach (thanks to ankit for pointing this out). If we modify the input to 5 2 2 8 5 6 8 8, then we should get 8 8 8 5 5 2 2 6 and not 8 8 8 2 2 5 5 6 as will be the case.
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.

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.

16. Count Inversions in an array

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)

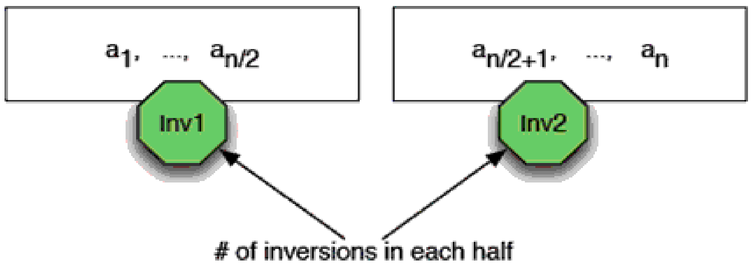
For each element, count number of elements which are on right side of it and are smaller than it.

```
int getInvCount(int arr[], int n)
{
  int inv_count = 0;
  int i, j;

  for(i = 0; i < n - 1; i++)
    for(j = i+1; j < n; j++)
      if(arr[i] > arr[j])
        inv_count++;

  return inv_count;
}

/* Driver progra to test above functions */
int main(int argv, char** args)
{
  int arr[] = {1, 20, 6, 4, 5};
  printf(" Number of inversions are %d \n", getInvCount(arr, 5));
  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

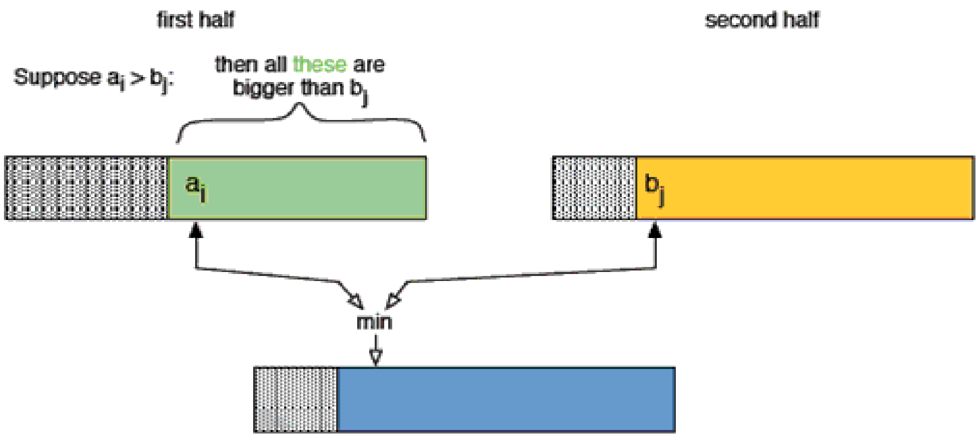
METHOD 2(Enhance Merge Sort)

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().

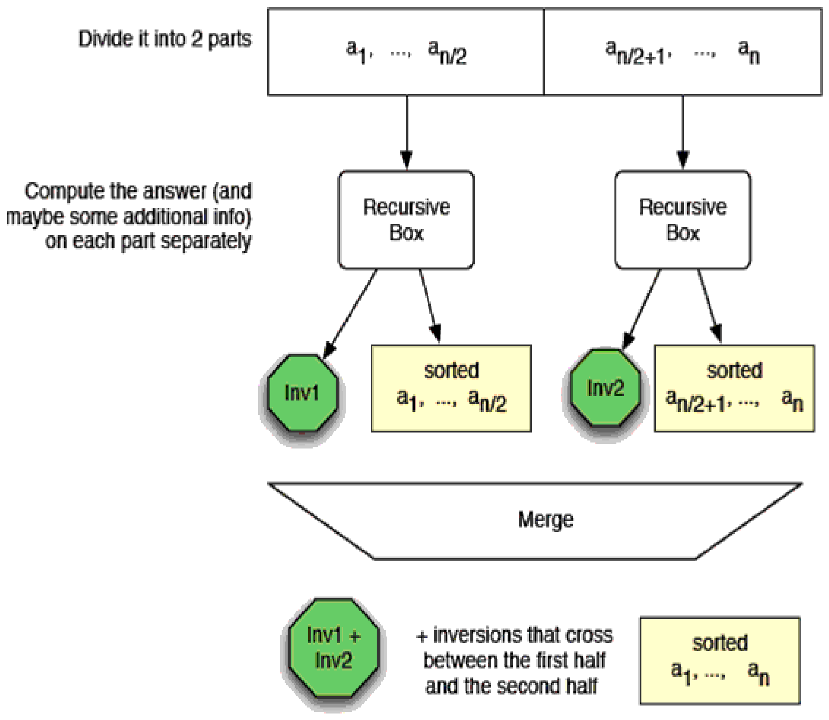


How to get number of inversions in merge()?

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



The complete picture:


Divide it into 2 parts
$a_1, \dots, a_{n/2}$
$a_{n/2+1}, \dots, a_n$
Compute the answer (and maybe some additional info) on each part separately
Recursive Box
Recursive Box
Inv1
sorted $a_1, \dots, a_{n/2}$
Inv2
sorted $a_{n/2+1}, \dots, a_n$
Merge
Inv1 + Inv2
+ inversions that cross between the first half and the second half
sorted $a_1, \dots, a_n$

Implementation:

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

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

/* This function sorts the input array and returns the
   number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
  returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
  int mid, inv_count = 0;
  if (right > left)
  {
    /* Divide the array into two parts and call _mergeSortAndCountInv()
       for each of the parts */
    mid = (right + left)/2;

    /* Inversion count will be sum of inversions in left-part, right-part
      and number of inversions in merging */
    inv_count  = _mergeSort(arr, temp, left, mid);
    inv_count += _mergeSort(arr, temp, mid+1, right);

    /*Merge the two parts*/
    inv_count += merge(arr, temp, left, mid+1, right);
  }
  return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
   the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
  int i, j, k;
  int inv_count = 0;

  i = left; /* i is index for left subarray*/
  j = mid;  /* i is index for right subarray*/
  k = left; /* i is index for resultant merged subarray*/
  while ((i <= mid - 1) && (j <= right))
  {
    if (arr[i] <= arr[j])
    {
      temp[k++] = arr[i++];
    }
    else
    {
      temp[k++] = arr[j++];

     /*this is tricky -- see above explanation/diagram for merge()*/
      inv_count = inv_count + (mid - i);
    }
  }

  /* Copy the remaining elements of left subarray
   (if there are any) to temp*/
  while (i <= mid - 1)
    temp[k++] = arr[i++];

  /* Copy the remaining elements of right subarray
   (if there are any) to temp*/
  while (j <= right)
    temp[k++] = arr[j++];

  /*Copy back the merged elements to original array*/
  for (i=left; i <= right; i++)
    arr[i] = temp[i];

  return inv_count;
}

/* Driver progra to test above functions */
int main(int argv, char** args)
{
  int arr[] = {1, 20, 6, 4, 5};
  printf(" Number of inversions are %d \n", mergeSort(arr, 5));
  getchar();
  return 0;
}
```

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.

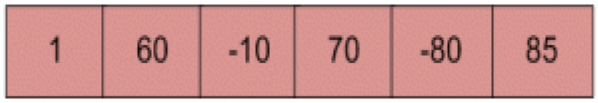
Time Complexity: O(nlogn)

Algorithmic Paradigm: Divide and Conquer

17. Two elements whose sum is closest to zero

Question: An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

For the below array, program should print -80 and 85.



1	60	-10	70	-80	85

METHOD 1 (Simple)

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

Implementation

```
# include <stdio.h>
# include <stdlib.h> /* for abs() */
# include <math.h>
void minAbsSumPair(int arr[], int arr_size)
{
  int inv_count = 0;
  int l, r, min_sum, sum, min_l, min_r;

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

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

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

  printf(" The two elements whose sum is minimum are %d and %d",
          arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 60, -10, 70, -80, 85};
  minAbsSumPair(arr, 6);
  getchar();
  return 0;
}
```

Time complexity: O(n^2)

METHOD 2 (Use Sorting)

Thanks to baskin for suggesting this approach. We recommend to read this post for background of this approach.

Algorithm

1) Sort all the elements of the input array.

2) Use two index variables l and r to traverse from left and right ends respectively. Initialize l as 0 and r as n-1.

3) sum = a[l] + a[r]

4) If sum is -ve, then l++

5) If sum is +ve, then r–

6) Keep track of abs min sum.

7) Repeat steps 3, 4, 5 and 6 while l < r

Implementation

```
# include <stdio.h>
# include <math.h>
# include <limits.h>

void quickSort(int *, int, int);

/* Function to print pair of elements having minimum sum */
void minAbsSumPair(int arr[], int n)
{
  // Variables to keep track of current sum and minimum sum
  int sum, min_sum = INT_MAX;

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

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

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

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

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

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

  printf(" The two elements whose sum is minimum are %d and %d",
          arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 60, -10, 70, -80, 85};
  int n = sizeof(arr)/sizeof(arr[0]);
  minAbsSumPair(arr, n);
  getchar();
  return 0;
}

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

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

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

  exchange (&arr[i + 1], &arr[ei]);
  return (i + 1);
}

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

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

18. Find the smallest and second smallest element in an array

Question: Write an efficient C program to find smallest and second smallest element in an array.

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
```

Implementation:

```c
#include <stdio.h>
#include <limits.h> /* For INT_MAX */

/* Function to print first smallest and second smallest elements */
void print2Smallest(int arr[], int arr_size)
{
  int i, first, second;

  /* There should be atleast two elements*/
  if(arr_size < 2)
  {
    printf(" Invalid Input ");
    return;
  }

  first = second = INT_MAX;
  for(i = 0; i < arr_size ; i ++)
  {

    /*If current element is smaller than first then update both
      first and second */
    if(arr[i] < first)
    {
      second = first;
      first = arr[i];
    }

    /* If arr[i] is in between first and second then update second  */
    else if (arr[i] < second)
    {
      second = arr[i];
    }
  }
  printf("The smallest element is %d and second Smallest element is %d",
         first, second);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {12, 13, 15, 10, 35, 1};
  print2Smallest(arr, 6);
  getchar();
  return 0;
}
```

The same approach can be used to find the largest and second largest elements in an array.

Time Complexity: O(n)

19. Check for Majority Element in a sorted array

Question: Write a C function to find if a given integer appears more than n/2 times in a sorted array of n integers.

Basically, 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).

In the following programs array is assumed to be sorted in non-decreasing order.

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.

```
/* Program to check for majority element in a sorted array */
# include <stdio.h>
# define bool int

bool isMajority(int arr[], int n, int x)
{
  int i;

 /* get last index according to n (even or odd) */
  int last_index = n%2? n/2: (n/2 + 1);

 /* search for first occurrence of x in arr[]*/
  for(i = 0; i < last_index; i++)
  {

    /* check if x is present and is present more than n/2 times */
    if( arr[i] == x && arr[i+n/2] == x)
       return 1;
  }
  return 0;
}

/* Driver program to check above function */
int main()
{
   int arr[10] = {1, 2, 3, 3, 3, 3, 10};
   int n = 7;
   int x = 3;
   if(isMajority(arr, n, x))
     printf("%d appears more than %d times in arr[]", x, n/2);
   else
    printf("%d does not appear more than %d times in arr[]", x, n/2);
   getchar();
   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.

```
/* Program to check for majority element in a sorted array */
# include <stdio.h>;
# define bool int

/* If x is present in arr[low...high] then returns the index of
  first occurance of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x, int n);

/* This function returns true if the x is present more than n/2
   times in arr[] of size n */
bool isMajority(int arr[], int n, int x)
{
   int i = _binarySearch(arr, 0, n-1, x, n);

   /* check if the element is present more than n/2 times */
   if(((i + n/2) <= (n -1)) && arr[i + n/2] == x)
     return 1;
   else
     return 0;
}

/* Standard Binary Search function*/
int _binarySearch(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/

    /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i)  mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
     */
    if(( mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
      return mid;
    else if(x > arr[mid])
      return _binarySearch(arr, (mid + 1), high, x, n);
    else
      return _binarySearch(arr, low, (mid -1), x, n);
  }
  return -1;
}

/* Driver program to check above functions */
int main()
{
   int arr[10] = {1, 2, 3, 3, 3, 3, 10};
   int n = 7;
   int x = 3;
   if(isMajority(arr, n, x))
     printf("%d appears more than %d times in arr[]", x, n/2);
   else
    printf("%d does not appear more than %d times in arr[]", x, n/2);

   getchar();
   return 0;
}
```

Time Complexity: O(Logn)

Algorithmic Paradigm: Divide and Conquer

20. Maximum and minimum of an array using minimum number of comparisons

Write a C function to return minimum and maximum in an array. You program should make minimum number of comparisons.

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)

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int n)
{
  struct pair minmax;
  int i;

  /*If there is only one element then return it as min and max both*/
  if(n == 1)
  {
     minmax.max = arr[0];
     minmax.min = arr[0];
     return minmax;
  }

  /* If there are more than one elements, then initialize min
      and max*/
  if(arr[0] > arr[1])
  {
      minmax.max = arr[0];
      minmax.min = arr[1];
  }
  else
  {
      minmax.max = arr[0];
      minmax.min = arr[1];
  }

  for(i = 2; i<n; i++)
  {
    if(arr[i] >  minmax.max)
      minmax.max = arr[i];

    else if(arr[i] <  minmax.min)
      minmax.min = arr[i];
  }

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax (arr, arr_size);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

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
```

Implementation

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int low, int high)
{
  struct pair minmax, mml, mmr;
  int mid;

  /* If there is only on element */
  if(low == high)
  {
     minmax.max = arr[low];
     minmax.min = arr[low];
     return minmax;
  }

  /* If there are two elements */
  if(high == low + 1)
  {
     if(arr[low] > arr[high])
     {
        minmax.max = arr[low];
        minmax.min = arr[high];
     }
     else
     {
        minmax.max = arr[high];
        minmax.min = arr[low];
     }
     return minmax;
  }

  /* If there are more than 2 elements */
  mid = (low + high)/2;
  mml = getMinMax(arr, low, mid);
  mmr = getMinMax(arr, mid+1, high);

  /* compare minimums of two parts*/
  if(mml.min < mmr.min)
    minmax.min = mml.min;
  else
    minmax.min = mmr.min;

  /* compare maximums of two parts*/
  if(mml.max > mmr.max)
    minmax.max = mml.max;
  else
    minmax.max = mmr.max;

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax(arr, 0, arr_size-1);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

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.

```
#include<stdio.h>

/* structure is used to return two values from minMax() */
struct pair
{
  int min;
  int max;
};

struct pair getMinMax(int arr[], int n)
{
  struct pair minmax;
  int i;

  /* If array has even number of elements then
    initialize the first two elements as minimum and
    maximum */
  if(n%2 == 0)
  {
    if(arr[0] > arr[1])
    {
      minmax.max = arr[0];
      minmax.min = arr[1];
    }
    else
    {
      minmax.min = arr[0];
      minmax.max = arr[1];
    }
    i = 2;  /* set the startung index for loop */
  }

   /* If array has odd number of elements then
    initialize the first element as minimum and
    maximum */
  else
  {
    minmax.min = arr[0];
    minmax.max = arr[0];
    i = 1;  /* set the startung index for loop */
  }

  /* In the while loop, pick elements in pair and
     compare the pair with max and min so far */
  while(i < n-1)
  {
    if(arr[i] > arr[i+1])
    {
      if(arr[i] > minmax.max)
        minmax.max = arr[i];
      if(arr[i+1] < minmax.min)
        minmax.min = arr[i+1];
    }
    else
    {
      if(arr[i+1] > minmax.max)
        minmax.max = arr[i+1];
      if(arr[i] < minmax.min)
        minmax.min = arr[i];
    }
    i += 2; /* Increment the index by 2 as two
               elements are processed in loop */
  }

  return minmax;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1000, 11, 445, 1, 330, 3000};
  int arr_size = 6;
  struct pair minmax = getMinMax (arr, arr_size);
  printf("\nMinimum element is %d", minmax.min);
  printf("\nMaximum element is %d", minmax.max);
  getchar();
}
```

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.

21. Segregate 0s and 1s in an array

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

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

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

Time Complexity: O(n)

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

Method 2 (Use two indexes to traverse)

Maintain two indexes. Initialize first index *left* as 0 and second index *right* as n-1.

Do following while *left* < *right*

a) Keep incrementing index *left* while there are 0s at it

b) Keep decrementing index *right* while there are 1s at it

c) If left < right then exchange arr[left] and arr[right]

Implementation:

```
#include<stdio.h>

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

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

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

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

/* driver program to test */
int main()
{
  int arr[] = {0, 1, 0, 1, 1, 1};
  int arr_size = 6, i = 0;

  segregate0and1(arr, arr_size);

  printf("array after segregation ");
  for(i = 0; i < 6; i++)
```

```
    printf("%d ", arr[i]);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

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

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

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

Method 1 (Use Bubble k times)
Thanks to Shailendra for suggesting this approach.
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)

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.
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.

23. Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

```
0  1  1  0  1
1  1  0  1  0
0  1  1  1  0
1  1  1  1  0
1  1  1  1  1
0  0  0  0  0
```

The maximum square sub-matrix with all set bits is

```
1  1  1
1  1  1
1  1  1
```

Algorithm:
Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents size of the square sub-matrix with all 1s including M[i][j] and M[i][j] is the rightmost and bottommost entry in sub-matrix.

```
1) Construct a sum matrix S[R][C] for the given M[R][C].
   a)     Copy first row and first columns as it is from M[][] to S[][]
   b)     For other entries, use following expressions to construct S[][]
      If M[i][j] is 1 then
        S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1
      Else /*If M[i][j] is 0*/
        S[i][j] = 0
2) Find the maximum entry in S[R][C]
3) Using the value and coordinates of maximum entry in S[i], print
  sub-matrix of M[][]
```

For the given M[R][C] in above example, constructed S[R][C] would be:

```
0 1 1 0 1
1 1 0 1 0
0 1 1 1 0
1 1 2 2 0
1 2 2 3 1
0 0 0 0 0
```

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```
#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
  int i,j;
  int S[R][C];
  int max_of_s, max_i, max_j;

  /* Set first column of S[][]*/
  for(i = 0; i < R; i++)
     S[i][0] = M[i][0];

  /* Set first row of S[][]*/
  for(j = 0; j < C; j++)
     S[0][j] = M[0][j];

  /* Construct other entries of S[][]*/
  for(i = 1; i < R; i++)
  {
    for(j = 1; j < C; j++)
    {
      if(M[i][j] == 1)
        S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
      else
        S[i][j] = 0;
    }
  }

  /* Find the maximum entry, and indexes of maximum entry
     in S[][] */
  max_of_s = S[0][0]; max_i = 0; max_j = 0;
  for(i = 0; i < R; i++)
  {
    for(j = 0; j < C; j++)
    {
      if(max_of_s < S[i][j])
      {
        max_of_s = S[i][j];
        max_i = i;
        max_j = j;
      }
    }
  }

  printf("\n Maximum size sub-matrix is: \n");
  for(i = max_i; i > max_i - max_of_s; i--)
  {
    for(j = max_j; j > max_j - max_of_s; j--)
    {
      printf("%d ", M[i][j]);
    }
    printf("\n");
  }
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{
  int m = a;
  if (m > b)
    m = b;
  if (m > c)
    m = c;
  return m;
}

/* Driver function to test above functions */
int main()
{
  bool M[R][C] =  {{0, 1, 1, 0, 1},
                   {1, 1, 0, 1, 0},
                   {0, 1, 1, 1, 0},
                   {1, 1, 1, 1, 0},
                   {1, 1, 1, 1, 1},
                   {0, 0, 0, 0, 0}};

  printMaxSubSquare(M);
  getchar();
}
```

Time Complexity: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Auxiliary Space: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Algorithmic Paradigm: Dynamic Programming

24. 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 numbe**r 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.

```
#include<stdio.h>

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

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 90, 10, 110};
  printf("Maximum difference is %d",  maxDiff(arr, 5));
  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

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).

```
#include<stdio.h>

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

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 6, 80, 100};
  int size = sizeof(arr)/sizeof(arr[0]);
  printf("Maximum difference is %d",  maxDiff(arr, size));
  getchar();
  return 0;
}
```

Time Complexity: O(n)

25. Union and Intersection of two sorted arrays

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**.**

```
#include<stdio.h>

/* Function prints union of arr1[] and arr2[]
   m is the number of elements in arr1[]
   n is the number of elements in arr2[] */
int printUnion(int arr1[], int arr2[], int m, int n)
{
  int i = 0, j = 0;
  while(i < m && j < n)
  {
    if(arr1[i] < arr2[j])
      printf(" %d ", arr1[i++]);
    else if(arr2[j] < arr1[i])
      printf(" %d ", arr2[j++]);
    else
    {
      printf(" %d ", arr2[j++]);
      i++;
    }
  }

  /* Print remaining elements of the larger array */
  while(i < m)
   printf(" %d ", arr1[i++]);
  while(j < n)
   printf(" %d ", arr2[j++]);
}

/* Driver program to test above function */
int main()
{
  int arr1[] = {1, 2, 4, 5, 6};
  int arr2[] = {2, 3, 5, 7};
  int m = sizeof(arr1)/sizeof(arr1[0]);
  int n = sizeof(arr2)/sizeof(arr2[0]);
  printUnion(arr1, arr2, m, n);
  getchar();
  return 0;
}
```

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.

```
#include<stdio.h>

/* Function prints Intersection of arr1[] and arr2[]
   m is the number of elements in arr1[]
   n is the number of elements in arr2[] */
int printIntersection(int arr1[], int arr2[], int m, int n)
{
  int i = 0, j = 0;
  while(i < m && j < n)
  {
    if(arr1[i] < arr2[j])
      i++;
    else if(arr2[j] < arr1[i])
      j++;
    else /* if arr1[i] == arr2[j] */
    {
      printf(" %d ", arr2[j++]);
      i++;
    }
  }
}

/* Driver program to test above function */
int main()
{
  int arr1[] = {1, 2, 4, 5, 6};
  int arr2[] = {2, 3, 5, 7};
  int m = sizeof(arr1)/sizeof(arr1[0]);
  int n = sizeof(arr2)/sizeof(arr2[0]);
  printIntersection(arr1, arr2, m, n);
  getchar();
  return 0;
}
```

Time Complexity: O(m+n)

26. Floor and 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 = 1:    floor  = 1,  ceil  = 1
For x = 5:    floor  = 2,  ceil  = 8
For x = 20:   floor  = 19,  ceil doesn't exist in array
```

In below methods, we have implemented only ceiling search functions. Floor search can be implemented in the same way.

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

```
#include<stdio.h>

/* Function to get index of ceiling of x in arr[low..high] */
int ceilSearch(int arr[], int low, int high, int x)
{
  int i;

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

  /* Otherwise, linearly search for ceil value */
  for(i = low; i < high; i++)
  {
    if(arr[i] == x)
      return i;

    /* if x lies between arr[i] and arr[i+1] including
       arr[i+1], then return arr[i+1] */
    if(arr[i] < x && arr[i+1] >= x)
       return i+1;
  }

  /* If we reach here then x is greater than the last element
```

```
    of the array,  return -1 in this case */
  return -1;
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 2, 8, 10, 10, 12, 19};
   int n = sizeof(arr)/sizeof(arr[0]);
   int x = 3;
   int index = ceilSearch(arr, 0, n-1, x);
   if(index == -1)
     printf("Ceiling of %d doesn't exist in array ", x);
   else
     printf("ceiling of %d is %d", x, arr[index]);
   getchar();
   return 0;
}
```

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).

```
#include<stdio.h>

/* Function to get index of ceiling of x in arr[low..high]*/
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);
  }
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 2, 8, 10, 10, 12, 19};
   int n = sizeof(arr)/sizeof(arr[0]);
   int x = 20;
   int index = ceilSearch(arr, 0, n-1, x);
   if(index == -1)
     printf("Ceiling of %d doesn't exist in array ", x);
   else
     printf("ceiling of %d is %d", x, arr[index]);
   getchar();
   return 0;
}
```

Time Complexity: O(Logn)

27. 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[].

Implementation:

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

/* Function to print product array for a given array
 arr[] of size n */
void productArray(int arr[], int n)
{
  /* Allocate memory for temporary arrays left[] and right[] */
  int *left = (int *)malloc(sizeof(int)*n);
  int *right = (int *)malloc(sizeof(int)*n);

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

  int i, j;

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

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

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

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

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

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

  return;
}

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

```
{
  int arr[] = {10, 3, 5, 6, 2};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("The product array is: \n");
  productArray(arr, n);
  getchar();
}
```

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];
  }

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

  return;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

Auxiliary Space: O(1)

28. 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]
```

Implementation:

```
#include<stdio.h>

/* Function to swap *a and *b */
void swap(int *a, int *b);

void segregateEvenOdd(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]%2 == 0 && left < right)
        left++;

     /* Decrement right index while we see 1 at right */
     while(arr[right]%2 == 1 && left < right)
        right--;

     if(left < right)
     {
       /* Swap arr[left] and arr[right]*/
       swap(&arr[left], &arr[right]);
       left++;
       right--;
     }
  }
}

/* UTILITY FUNCTIONS */
void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}

/* driver program to test */
int main()
{
  int arr[] = {12, 34, 45, 9, 8, 90, 3};
  int arr_size = 7, i = 0;

  segregateEvenOdd(arr, arr_size);

  printf("array after segregation ");
  for(i = 0; i < arr_size; i++)
    printf("%d ", arr[i]);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

29. 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.

```
#include<stdio.h>
#include<stdlib.h>
void printRepeating(int arr[], int size)
{
  int i, j;
  printf(" Repeating elements are ");
  for(i = 0; i < size; i++)
    for(j = i+1; j < size; j++)
      if(arr[i] == arr[j])
        printf(" %d ", arr[i]);
}

int main()
{
  int arr[] = {4, 2, 4, 5, 2, 3, 1};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printRepeating(arr, arr_size);
  getchar();
  return 0;
}
```

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.

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

```
void printRepeating(int arr[], int size)
{
  int *count = (int *)calloc(sizeof(int), (size - 2));
  int i;

  printf(" Repeating elements are ");
  for(i = 0; i < size; i++)
  {
    if(count[arr[i]] == 1)
      printf(" %d ", arr[i]);
    else
     count[arr[i]]++;
  }
}

int main()
{
  int arr[] = {4, 2, 4, 5, 2, 3, 1};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printRepeating(arr, arr_size);
  getchar();
  return 0;
}
```

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

The methods 3 and 4 use all useful information given in the question

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

/* function to get factorial of n */
int fact(int n);

void printRepeating(int arr[], int size)
{
  int S = 0;  /* S is for sum of elements in arr[] */
  int P = 1;  /* P is for product of elements in arr[] */
  int x,  y;   /* x and y are two repeating elements */
  int D;      /* D is for difference of x and y, i.e., x-y*/
  int n = size - 2,  i;

  /* Calculate Sum and Product of all elements in arr[] */
  for(i = 0; i < size; i++)
  {
    S = S + arr[i];
    P = P*arr[i];
  }

  S = S - n*(n+1)/2;  /* S is x + y now */
  P = P/fact(n);         /* P is x*y now */

  D = sqrt(S*S - 4*P); /* D is x - y now */

  x = (D + S)/2;
  y = (S - D)/2;

  printf("The two Repeating elements are %d & %d", x, y);
}

int fact(int n)
{
   return (n == 0)? 1 : n*fact(n-1);
}

int main()
{
  int arr[] = {4, 2, 4, 5, 2, 3, 1};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printRepeating(arr, arr_size);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Method 4 (Use XOR)

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);
}
```

```
int main()
{
  int arr[] = {4, 2, 4, 5, 2, 3, 1};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printRepeating(arr, arr_size);
  getchar();
  return 0;
}
```

Method 5 (Use array elements as index)

traverse the list for i= 1st to n+2 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
}
```

Example: A[] = {1,1,2,3,2}

i=1 ; A[abs(A[1])] i.e,A[1] is positive ; so make it negative ;

so list now is {-1,1,2,3,2}

i=2 ; A[abs(A[2])] i.e., A[1] is negative ; so A[i] i.e., A[2] is repetition,

now list is {-1,1,2,3,2}

i=3 ; list now becomes {-1,-1,2,3,2} and A[3] is not repeated

now list becomes {-1,-1,-2,3,2}

i=4 ;

and A[4]=3 is not repeated

i=5 ; we find A[abs(A[i])] = A[2] is negative so A[i]= 2 is a repetition,

This method modifies the original array.

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

```
void printRepeating(int arr[], int size)
{
  int i;

  printf("\n The repeating elements are");

  for(i = 0; i < size; i++)
  {
    if(arr[abs(arr[i])] > 0)
      arr[abs(arr[i])] = -arr[abs(arr[i])];
    else
      printf(" %d ", abs(arr[i]));
  }
}

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

Method 6:

Traverse through the array, and do the following:
1) if (arr[index] == index), then do nothing
2) if (arr[index] != index), put arr[index] in its correct place by doing a swap.

Repeat the above process once again(these steps have to be followed twice). Now the last 2 elements of the array are the ones which are repeated.

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

30. 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:

1. a[1..Lo-1] zeroes (red)
2. a[Lo..Mid-] ones (white)
3. a[Mid..Hi] unknown
4. a[Hi+1..N] twos (blue)

The unknown region is shrunk while maintaining these conditions

1. Lo := 1; Mid := 1; Hi := N;
2. while Mid <= Hi do
    1. Invariant: a[1..Lo-1]=0 and a[Lo..Mid-1]=1 and a[Hi+1..N]=2; a[Mid..Hi] are unknown.
    2. case a[Mid] in
    - 0: swap a[Lo] and a[Mid]; Lo++; Mid++
    - 1: Mid++
    - 2: swap a[Mid] and a[Hi]; Hi–

— Dutch National Flag Algorithm, or 3-way Partitioning —

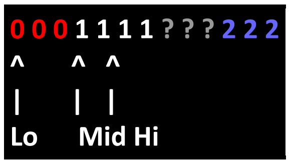
Part way through the process, some red, white and blue elements are known and are in the “right” place. The section of unknown elements, a[Mid..Hi], is shrunk by examining a[Mid]:

```
|
0 0 0 1 1 1 ? ? ? ? 2 2 2
^   ^   ^
|   |   |
Lo  Mid Hi
```

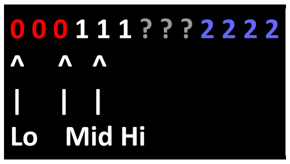
Examine a[Mid]. There are three possibilities: a[Mid] is (0) red, (1) white or (2) blue.
Case (0) a[Mid] is red, swap a[Lo] and a[Mid]; Lo++; Mid++

```
0 0 0 0 1 1 1 ? ? ? 2 2 2
^   ^ ^
|   | |
Lo  Mid Hi
```

Case (1) a[Mid] is white, Mid++



Case (2) a[Mid] is blue, swap a[Mid] and a[Hi]; Hi–



Continue until Mid>Hi.

```
#include<stdio.h>

/* Function to swap *a and *b */
void swap(int *a, int *b);

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;
      }
   }
}

/* UTILITY FUNCTIONS */
void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

```
/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size)
{
  int i;
  for (i = 0; i < arr_size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

/* driver program to test */
int main()
{
  int arr[] = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  int i;

  sort012(arr, arr_size);

  printf("array after segregation ");
  printArray(arr, arr_size);

  getchar();
  return 0;
}
```

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 put following loop before switch condition to make sure that hi is pointing to location which is not 2 so that it would eliminate unnecessary swaps of 2.

```
while ((a[hi]==2) && hi >= mid)
    –hi;
if (hi < mid)
    break;
```

31. Find the 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*.

Implementation:

```
#include<stdio.h>

void printUnsorted(int arr[], int n)
{
  int s = 0, e = n-1, i, max, min;

  // step 1(a) of above algo
  for (s = 0; s < n-1; s++)
  {
    if (arr[s] > arr[s+1])
      break;
  }
  if (s == n-1)
  {
    printf("The complete array is sorted");
    return;
  }

  // step 1(b) of above algo
  for(e = n - 1; e > 0; e--)
  {
    if(arr[e] < arr[e-1])
      break;
  }

  // step 2(a) of above algo
  max = arr[s]; min = arr[s];
  for(i = s + 1; i <= e; i++)
  {
    if(arr[i] > max)
      max = arr[i];
    if(arr[i] < min)
      min = arr[i];
  }

  // step 2(b) of above algo
  for( i = 0; i < s; i++)
  {
    if(arr[i] > min)
    {
      s = i;
      break;
    }
  }

  // step 2(c) of above algo
  for( i = n -1; i >= e+1; i--)
  {
    if(arr[i] < max)
    {
      e = i;
      break;
    }
  }

  // step 3 of above algo
  printf(" The unsorted subarray which makes the given array "
         " sorted lies between the indees %d and %d", s, e);
  return;
}

int main()
{
  int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printUnsorted(arr, arr_size);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

32. Find duplicates in O(n) time and O(1) 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, 0, 6}, the answer should be 1 & 3.

This problem is an extended version of following problem.

Find the two repeating elements in a given array

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

Algorithm:

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

Implementation:

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

void printRepeating(int arr[], int size)
{
  int i;
  printf("The repeating elements are: \n");
  for(i = 0; i < size; i++)
  {
    if(arr[abs(arr[i])] >= 0)
      arr[abs(arr[i])] = -arr[abs(arr[i])];
    else
      printf(" %d ", abs(arr[i]));
  }
}

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

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

Output:

The repeating elements are:
1 3

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

33. 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).

```
#include <stdio.h>

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

  /* Check for indexes one by one until an equilibrium
    index is found */
  for ( i = 0; i < n; ++i)
  {
    leftsum = 0;  // initialize left sum for current index i
    rightsum = 0; // initialize right sum for current index i

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

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

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

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

int main()
{
  int arr[] = {-7, 1, 5, 2, -4, 3, 0};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printf("%d\n", equilibrium(arr, arr_size));

  getchar();
  return 0;
}
```

Time Complexity: 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
          // there is no equilibrium index
```

```c
#include <stdio.h>

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

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

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

      if(leftsum == sum)
        return i;

      leftsum += arr[i];
   }

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

int main()
{
  int arr[] = {-7, 1, 5, 2, -4, 3, 0};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printf("First equilibrium index is %d\n", equilibrium(arr, arr_size));
  getchar();
  return 0;
}
```

Time Complexity: O(n)

we can remove the return statement and add a print statement to print all equilibrium indexes instead of returning only one.

34. Turn an image by 90 degree

Given an image, how will you turn it by 90 degrees? A vague question. Minimize the browser and try your solution before going further.

An image can be treated as 2D matrix which can be stored in a buffer. We are provided with matrix dimensions and it's base address. How can we turn it?

For example see the below picture,

```
* * * ^ * * *
* * * | * * *
* * * | * * *
* * * | * * *
```

After rotating right, it appears (observe arrow direction)

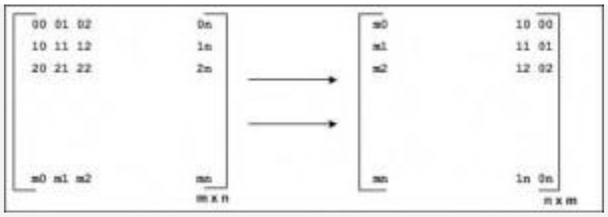
```
* * * *
* * * *
* * * *
-- - - >
* * * *
* * * *
* * * *
```

The idea is simple. Transform each row of source matrix into required column of final image. We will use an auxiliary buffer to transform the image.

From the above picture, we can observe that

```
first row of source ------> last column of destination
second row of source ------> last but-one column of destination
so ... on
last row of source ------> first column of destination
```

In pictorial form, we can represent the above transformations of an (m x n) matrix into (n x m) matrix,



Transformations

If you have not attempted, atleast try your pseudo code now.

It will be easy to write our pseudo code. In C/C++ we will usually traverse matrix on row major order. Each row is transformed into different column of final image. We need to construct columns of final image. See the following algorithm (transformation)

```
for(r = 0; r < m; r++)
{
  for(c = 0; c < n; c++)
  {
    // Hint: Map each source element indices into
    // indices of destination matrix element.
     dest_buffer [ c ] [ m - r - 1 ] = source_buffer [ r ] [ c ];
  }
}
```

Note that there are various ways to implement the algorithm based on traversal of matrix, row major or column major order. We have two matrices and two ways (row and column major) to traverse each matrix. Hence, there can atleast be 4 different ways of transformation of source matrix into final matrix.

Code:

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

```
void displayMatrix(unsigned int const *p, unsigned int row, unsigned int col);
void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int
col);

int main()
{
    // declarations
    unsigned int image[][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
    unsigned int *pSource;
    unsigned int *pDestination;
    unsigned int m, n;

    // setting initial values and memory allocation
    m = 3, n = 4, pSource = (unsigned int *)image;
    pDestination = (unsigned int *)malloc(sizeof(int)*m*n);

    // process each buffer
    displayMatrix(pSource, m, n);

    rotate(pSource, pDestination, m, n);

    displayMatrix(pDestination, n, m);

    free(pDestination);

    getchar();
    return 0;
}

void displayMatrix(unsigned int const *p, unsigned int r, unsigned int c)
{
    unsigned int row, col;
    printf("\n\n");

    for(row = 0; row < r; row++)
    {
        for(col = 0; col < c; col++)
        {
            printf("%d\t", *(p + row * c + col));
        }
        printf("\n");
    }

    printf("\n\n");
}

void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int
col)
{
    unsigned int r, c;
    for(r = 0; r < row; r++)
    {
        for(c = 0; c < col; c++)
        {
            *(pD + c * row + (row - r - 1)) = *(pS + r * col + c);
        }
    }
}
```

35. Search in a 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

Implementation:

```
#include<stdio.h>

/* Searches the element x in mat[][]. If the element is found,
    then prints its position and returns true, otherwise prints
    "not found" and returns false */
int search(int mat[4][4], int n, int x)
{
   int i = 0, j = n-1;  //set indexes for top right element
   while ( i < n && j >= 0 )
   {
      if ( mat[i][j] == x )
      {
         printf("\n Found at %d, %d", i, j);
         return 1;
      }
      if ( mat[i][j] > x )
        j--;
      else //  if mat[i][j] < x
        i++;
   }

   printf("\n Element not found");
   return 0;  // if ( i==n || j== -1 )
}

// driver program to test above function
int main()
{
  int mat[4][4] = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {27, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
  search(mat, 4, 29);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

The above approach will also work for m x n matrix (not only for n x n). Complexity would be O(m + n).

36. Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

a) For any array, rightmost element always has next greater element as -1.

b) For an array which is sorted in decreasing order, all elements 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
```

d) For the input array [13, 7, 6, 12}, the next greater elements for each element are as follows.

```
 Element      NGE
  13     -->   -1
  7      -->   12
  6      -->   12
  12    -->   -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.

```
#include<stdio.h>
/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next = -1;
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -> %d\n", arr[i], next);
    }
}

int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Output:

```
11 --> 13
13 --> 21
21 --> -1
3 --> -1
```

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 fot the popped element.

....d) Keep poppoing 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.

```
#include<stdio.h>
#include<stdlib.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};

// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        ps->items1 = x;
    }
}

bool isEmpty(struct stack *ps)
{
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        temp = ps->items1;
        ps->top -= 1;
        return temp;
    }
}

/* 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);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Output:

```
11 --> 13
13 --> 21
3 --> -1
21 --> -1
```

Time Complexity: O(n). The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

a) Initialy pushed to the stack.

b) Popped from the stack when next element is being processed.

c) Pushed back to the stack because next element is smaller.

d) Popped from the stack in step 3 of algo.

37. Check if array elements are consecutive | Added Method 3

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[].

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

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
  If elements are consecutive, then returns true, else returns
  false */
bool areConsecutive(int arr[], int n)
{
  if ( n <  1 )
    return false;

  /* 1) Get the minimum element in array */
  int min = getMin(arr, n);

  /* 2) Get the maximum element in array */
  int max = getMax(arr, n);

  /* 3) max - min + 1 is equal to n,  then only check all elements */
  if (max - min  + 1 == n)
  {
      /* Create a temp array to hold visited flag of all elements.
         Note that, calloc is used here so that all values are initialized
         as false */
      bool *visited = (bool *)calloc(sizeof(bool), n);
      int i;
      for (i = 0; i < n; i++)
      {
         /* If we see an element again, then return false */
         if ( visited[arr[i] - min] != false )
           return false;

         /* If visited first time, then mark the element as visited */
         visited[arr[i] - min] = true;
      }

      /* If all elements occur once, then return true */
      return true;
  }

  return false; // if (max - min  + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
{
  int min = arr[0];
  for (int i = 1; i < n; i++)
   if (arr[i] < min)
     min = arr[i];
  return min;
}

int getMax(int arr[], int n)
{
  int max = arr[0];
  for (int i = 1; i < n; i++)
   if (arr[i] > max)
     max = arr[i];
  return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {5, 4, 2, 3, 1, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}
```

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.

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

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
  If elements are consecutive, then returns true, else returns
  false */
bool areConsecutive(int arr[], int n)
{

    if ( n <  1 )
        return false;

    /* 1) Get the minimum element in array */
    int min = getMin(arr, n);

    /* 2) Get the maximum element in array */
    int max = getMax(arr, n);

    /* 3) max - min + 1 is equal to n then only check all elements */
    if (max - min  + 1 == n)
    {
        int i;
        for(i = 0; i < n; i++)
        {
            int j;

            if (arr[i] < 0)
                j = -arr[i] - min;
            else
                j = arr[i] - min;

            // if the value at index j is negative then
            // there is repitition
            if (arr[j] > 0)
                arr[j] = -arr[j];
            else
                return false;
        }

        /* If we do not see a negative value then all elements
           are distinct */
        return true;
    }

    return false; // if (max - min  + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
{
    int min = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] < min)
            min = arr[i];
    return min;
}

int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {1, 4, 5, 3, 2, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}
```

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)

38. Find the smallest missing number

Given a **sorted** array of n 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.

Examples
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)
Traverse the input array 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.

```
#include<stdio.h>

int findFirstMissing(int array[], int start, int end) {

    if(start  > end)
      return end + 1;

    if (start != array[start])
      return start;

    int mid = (start + end) / 2;

    if (array[mid] > mid)
      return findFirstMissing(array, start, mid);
    else
      return findFirstMissing(array, mid + 1, end);
}

// driver program to test above function
int main()
{
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 10};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf(" First missing element is %d",
           findFirstMissing(arr, 0, n-1));
  getchar();
  return 0;
}
```

Note: This method doesn't work if there are duplicate elements in the array.

Time Complexity: O(Logn)

39. Count the 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 = 2
Output: 4 // x (or 2) occurs 4 times in arr[]

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

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

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);

```
/* if x is present in arr[] then returns the count of occurrences of x,
   otherwise returns -1. */
int count(int arr[], int x, int n)
{
  int i; // index of first occurrence of x in arr[0..n-1]
  int j; // index of last occurrence of x in arr[0..n-1]

  /* get the index of first occurrence of x */
  i = first(arr, 0, n-1, x, n);
  /* If x doesn't exist in arr[] then return -1 */
  if(i == -1)
    return i;

  /* Else get the index of last occurrence of x. Note that we
      are only looking in the subarray after first occurrence */
  j = last(arr, i, n-1, x, n);

  /* return count */
  return j-i+1;
}

/* if x is present in arr[] then returns the index of FIRST occurrence
   of x in arr[0..n-1], otherwise returns -1 */
int first(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/
    if( ( mid == 0 || x > arr[mid-1]) && arr[mid] == x)
      return mid;
    else if(x > arr[mid])
      return first(arr, (mid + 1), high, x, n);
    else
      return first(arr, low, (mid -1), x, n);
  }
  return -1;
}

/* if x is present in arr[] then returns the index of LAST occurrence
   of x in arr[0..n-1], otherwise returns -1 */
int last(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/
    if( ( mid == n-1 || x < arr[mid+1]) && arr[mid] == x )
      return mid;
    else if(x < arr[mid])
      return last(arr, low, (mid -1), x, n);
    else
      return last(arr, (mid + 1), high, x, n);
  }
  return -1;
}

/* driver program to test above functions */
int main()
{
  int arr[] = {1, 2, 2, 3, 3, 3, 3};
  int x =  3;  // Element to be counted in arr[]
  int n = sizeof(arr)/sizeof(arr[0]);
  int c = count(arr, x, n);
  printf(" %d occurs %d times ", x, c);
  getchar();
  return 0;
}
```

Time Complexity: O(Logn)

Programming Paradigm: Divide & Conquer

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

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.

```
#include <stdio.h>
/* For a given array arr[], returns the maximum j – i such that
    arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff = -1;
    int i, j;
    for (i = 0; i < n; ++i)
    {
        for (j = n-1; j > i; --j)
        {
            if(arr[j] > arr[i] && maxDiff < (j - i))
                maxDiff = j - i;
        }
    }

    return maxDiff;
}

int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}
```

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.

```
#include <stdio.h>

/* Utility Functions to get max and minimum of two integers */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x < y? x : y;
}
```

```
/* 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;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Auxiliary Space: O(n)

41. Maximum of all subarrays of size k

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

Examples:

Input :

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

k = 3

Output :

3 3 4 5 5 5 6

Input :

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

k = 4

Output :

10 10 10 15 15 90 90

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.

```
#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
    int j, max;

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

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

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

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).

42. Find whether an array is subset of another array | Added Method 3

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.

Examples:
Input: arr1[] = {11, 1, 13, 21, 3, 7}, arr2[] = {11, 3, 7, 1}
Output: arr2[] is a subset of arr1[]

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.

```
#include<stdio.h>

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        for (j = 0; j<m; j++)
        {
           if(arr2[i] == arr1[j])
              break;
        }

        /* If the above inner loop was not broken at all then
           arr2[i] is not present in arr1[] */
        if (j == m)
           return 0;
    }

    /* If we reach here then all elements of arr2[]
      are present in arr1[] */
    return 1;
}

int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
      printf("arr2[] is subset of arr1[] ");
    else
      printf("arr2[] is not a subset of arr1[]");

    getchar();
    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.

```
#include<stdio.h>

/* Fucntion prototypes */
void quickSort(int *arr, int si, int ei);
int binarySearch(int arr[], int low, int high, int x);

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;

    quickSort(arr1, 0, m-1);
    for (i=0; i<n; i++)
    {
        if (binarySearch(arr1, 0, m-1, arr2[i]) == -1)
           return 0;
    }

    /* If we reach here then all elements of arr2[]
      are present in arr1[] */
    return 1;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SEARCHING AND SORTING PURPOSE */
/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int x)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/

    /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i)  mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
     */
    if(( mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
      return mid;
    else if(x > arr[mid])
      return binarySearch(arr, (mid + 1), high, x);
    else
      return binarySearch(arr, low, (mid -1), x);
  }
 return -1;
}
```

```
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a   = *b;
    *b   = temp;
}

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

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

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

/*Driver program to test above functions */
int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
      printf("arr2[] is subset of arr1[] ");
    else
      printf("arr2[] is not a subset of arr1[] ");
    getchar();
    return 0;
}
```

Time Complexity: O(mLogm + nLogm) 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[].

```
/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;

    if(m < n)
       return 0;

    quickSort(arr1, 0, m-1);
    quickSort(arr2, 0, n-1);
    while( i < n && j < m )
    {
        if( arr1[j] <arr2[i] )
            j++;
        else if( arr1[j] == arr2[i] )
        {
            j++;
            i++;
        }
        else if( arr1[j] > arr2[i] )
            return 0;
    }

    if( i < n )
        return 0;
    else
        return 1;
}
```

Time Complexity: O(mLogm + nLogn)

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.

43. 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.

```
#include <stdio.h>
#include <stdlib.h> // for abs()
#include <limits.h> // for INT_MAX

int minDist(int arr[], int n, int x, int y)
{
   int i, j;
   int min_dist = INT_MAX;
   for (i = 0; i < n; i++)
   {
     for (j = i+1; j < n; j++)
     {
         if( (x == arr[i] && y == arr[j] ||
                y == arr[i] && x == arr[j]) && min_dist > abs(i-j))
          {
                min_dist = abs(i-j);
          }
      }
   }
   return min_dist;
}

/* Driver program to test above fnction */
int main()
{
    int arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
              minDist(arr, n, x, y));
    return 0;
}
```

Output: *Minimum distance between 3 and 6 is 4*

Time Complexity: O(n^2)

Method 2 (Tricky)

1) Traverse array from left side and stop if either *x* or *y* are found. Store index of this first occurrrence 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*.

```
#include <stdio.h>
#include <limits.h>  // For INT_MAX

int minDist(int arr[], int n, int x, int y)
{
   int i = 0;
   int min_dist = INT_MAX;
   int prev;

   // Find the first occurence of any of the two numbers (x or y)
   // and store the index of this occurence in prev
   for (i = 0; i < n; i++)
   {
     if (arr[i] == x || arr[i] == y)
     {
       prev = i;
       break;
```

```
      }
   }

   // Traverse after the first occurence
   for ( ; i < n; i++)
   {
      if (arr[i] == x || arr[i] == y)
      {
          // If the current element matches with any of the two then
          // check if current element and prev element are different
          // Also check if this value is smaller than minimm distance so
far
          if ( arr[prev] != arr[i] && (i - prev) < min_dist )
          {
             min_dist = i - prev;
             prev = i;
          }
          else
             prev = i;
      }
   }

   return min_dist;
}

/* Driver program to test above fnction */
int main()
{
    int arr[] ={3, 5, 4, 2, 6, 3, 0, 0, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
              minDist(arr, n, x, y));
    return 0;
}
```

Output: *Minimum distance between 3 and 6 is 4*

Time Complexity: O(n)

44. Find the repeating and the missing | Added 3 new methods

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.

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

void printTwoElements(int arr[], int size)
{
    int i;
    printf("\n The repeating element is");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])-1] > 0)
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];
        else
            printf(" %d ", abs(arr[i]));
    }
    printf("\nand the missing element is ");
    for(i=0; i<size; i++)
    {
        if(arr[i]>0)
            printf("%d",i+1);
    }
}

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

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.

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.

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

/* 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 */
}
```

```
/* Driver program to test above function */
int main()
{
  int arr[] = {1, 3, 4, 5, 5, 6, 2};
  int *x = (int *)malloc(sizeof(int));
  int *y = (int *)malloc(sizeof(int));
  int n = sizeof(arr)/sizeof(arr[0]);
  getTwoElements(arr, n, x, y);
  printf(" The two elements are %d and %d", *x, *y);
  getchar();
}
```

Time Complexity: O(n)

This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing

45. 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
```

Implementation:

```
int  arr[MAX_X][MAX_Y];
void spiral(int x, int y, int a[MAX_X][MAX_Y])
```

```
{
        while(x <= MAX_X/2 || y <= MAX_Y/2 )
        {
                int i,j;
                i=x; j=y;
                while(j < MAX_Y-y-1)
                        printf("%d,", a[i][j++]);
                while(i < MAX_X-x-1)
                        printf("%d,", a[i++][j]);
                while(j > y)
                        printf("%d,", a[i][j--]);
                while(i > x)
                        printf("%d,", a[i--][j]);
                x++; y++;
        }
}
```

46. 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
1 0
0 0
should be changed to following
1 1
1 0

Example 2
The matrix
0 0 0
0 0 1
should be changed to following
0 0 1
1 1 1
```

Example 3

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.

```
#include <stdio.h>
#define R 3
#define C 4

void modifyMatrix(bool mat[R][C])
{
    bool row[R];
    bool col[C];

    int i, j;

    /* Initialize all values of row[] as 0 */
    for (i = 0; i < R; i++)
    {
       row[i] = 0;
    }

    /* Initialize all values of col[] as 0 */
    for (i = 0; i < C; i++)
    {
       col[i] = 0;
    }

    /* Store the rows and columns to be marked as 1 in row[] and col[]
       arrays respectively */
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            if (mat[i][j] == 1)
            {
                row[i] = 1;
                col[j] = 1;
            }
        }
    }

    /* Modify the input matrix mat[] using the above constructed row[] and
       col[] arrays */
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            if ( row[i] == 1 || col[j] == 1 )
            {
                mat[i][j] = 1;
            }
        }
    }
}

/* A utility function to print a 2D matrix */
void printMatrix(bool mat[R][C])
{
    int i, j;
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { {1, 0, 0, 1},
        {0, 0, 1, 0},
        {0, 0, 0, 0},
    };

    printf("Input Matrix \n");
    printMatrix(mat);

    modifyMatrix(mat);

    printf("Matrix after modification \n");
    printMatrix(mat);
    return 0;
}
```

Output:

```
Input Matrix
1 0 0 1
0 0 1 0
0 0 0 0
Matrix after modification
1 1 1 1
1 1 1 1
1 0 1 1
```

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)

47. Median in a stream of integers (running integers)

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

```
After reading 1st element of stream - 5 -> median - 5
After reading 2nd element of stream - 5, 15 -> median - 10
After reading 3rd element of stream - 5, 15, 1 -> median - 5
After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so on...
```

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

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

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(n^2) 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.

Given below is implementation of above method. For algorithm to build these heaps, please read the highlighted code.

```
#include <iostream>
using namespace std;

// Heap capacity
#define MAX_HEAP_SIZE (128)
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

//// Utility functions

// exchange a and b
inline
void Exch(int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}

// Greater and Smaller are used as comparators
bool Greater(int a, int b)
{
    return a > b;
}
```

```
bool Smaller(int a, int b)
{
    return a < b;
}

int Average(int a, int b)
{
    return (a + b) / 2;
}

// Signum function
// = 0  if a == b  - heaps are balanced
// = -1 if a < b   - left contains less elements than right
// = 1  if a > b   - left contains more elements than right
int Signum(int a, int b)
{
    if( a == b )
        return 0;

    return a < b ? -1 : 1;
}

// Heap implementation
// The functionality is embedded into
// Heap abstract class to avoid code duplication
class Heap
{
public:
    // Initializes heap array and comparator required
    // in heapification
    Heap(int *b, bool (*c)(int, int)) : A(b), comp(c)
    {
        heapSize = -1;
    }

    // Frees up dynamic memory
    virtual ~Heap()
    {
        if( A )
        {
            delete[] A;
        }
    }

    // We need only these four interfaces of Heap ADT
    virtual bool Insert(int e) = 0;
    virtual int GetTop() = 0;
    virtual int ExtractTop() = 0;
    virtual int GetCount() = 0;

protected:

    // We are also using location 0 of array
```

```
int left(int i)
{
    return 2 * i + 1;
}

int right(int i)
{
    return 2 * (i + 1);
}

int parent(int i)
{
    if( i <= 0 )
    {
        return -1;
    }

    return (i - 1)/2;
}

// Heap array
int  *A;
// Comparator
bool  (*comp)(int, int);
// Heap size
int  heapSize;

// Returns top element of heap data structure
int top(void)
{
    int max = -1;

    if( heapSize >= 0 )
    {
        max = A[0];
    }

    return max;
}

// Returns number of elements in heap
int count()
{
    return heapSize + 1;
}

// Heapification
// Note that, for the current median tracing problem
// we need to heapify only towards root, always
void heapify(int i)
{
    int p = parent(i);

    // comp - differentiate MaxHeap and MinHeap
```

```
        // percolates up
        if( p >= 0 && comp(A[i], A[p]) )
        {
            Exch(A[i], A[p]);
            heapify(p);
        }
    }

    // Deletes root of heap
    int deleteTop()
    {
        int del = -1;

        if( heapSize > -1)
        {
            del = A[0];

            Exch(A[0], A[heapSize]);
            heapSize--;
            heapify(parent(heapSize+1));
        }

        return del;
    }

    // Helper to insert key into Heap
    bool insertHelper(int key)
    {
        bool ret = false;

        if( heapSize < MAX_HEAP_SIZE )
        {
            ret = true;
            heapSize++;
            A[heapSize] = key;
            heapify(heapSize);
        }

        return ret;
    }
};

// Specilization of Heap to define MaxHeap
class MaxHeap : public Heap
{
private:

public:
    MaxHeap() : Heap(new int[MAX_HEAP_SIZE], &Greater)  {  }

    ~MaxHeap()  { }

    // Wrapper to return root of Max Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Max Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Max Heap
    int GetCount()
    {
        return count();
    }

    // Wrapper to insert into Max Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Specilization of Heap to define MinHeap
class MinHeap : public Heap
{
private:

public:

    MinHeap() : Heap(new int[MAX_HEAP_SIZE], &Smaller) { }

    ~MinHeap() { }

    // Wrapper to return root of Min Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Min Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Min Heap
    int GetCount()
    {
        return count();
    }

    // Wrapper to insert into Min Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// 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;
}

void printMedian(int A[], int size)
{
    int m = 0; // effective median
    Heap *left  = new MaxHeap();
    Heap *right = new MinHeap();

    for(int i = 0; i < size; i++)
    {
        m = getMedian(A[i], m, *left, *right);

        cout << m << endl;
    }

    // C++ more flexible, ensure no leaks
    delete left;
    delete right;
}
// Driver code
int main()
{
    int A[] = {5, 15, 1, 3, 2, 8, 7, 9, 10, 6, 11, 4};
    int size = ARRAY_SIZE(A);

    // In lieu of A, we can also use data read from a stream
    printMedian(A, size);

    return 0;
}
```

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.

48. 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
```

Asked by rajk

Method 1 (Linear Search)
Linearly search for an index i such that arr[i] == i. Return the first such index found. Thanks to pm for suggesting this solution.

```
int linearSearch(int arr[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(arr[i] == i)
            return i;
    }

    /* If no fixed point present then return -1 */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[10] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", linearSearch(arr, n));
    getchar();
    return 0;
}
```

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.

```
int binarySearch(int arr[], int low, int high)
{
    if(high >= low)
    {
        int mid = (low + high)/2;  /*low + (high - low)/2;*/
        if(mid == arr[mid])
            return mid;
        if(mid > arr[mid])
            return binarySearch(arr, (mid + 1), high);
        else
            return binarySearch(arr, low, (mid -1));
    }

    /* Return -1 if there is no Fixed Point */
    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[10] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Fixed Point is %d", binarySearch(arr, 0, n-1));
    getchar();
    return 0;
}
```

Algorithmic Paradigm: Divide & Conquer

Time Complexity: O(Logn)

49. 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.

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

int bitonic(int arr[], int n)
{
    int i;
    int *inc = new int[n];
    int *dec = new int[n];
    int max;
    inc[0] = 1; // The length of increasing sequence ending at first index is 1
    dec[n-1] = 1; // The length of increasing sequence starting at first index 
is 1

    // Step 1) Construct increasing sequence array
    for(i = 1; i < n; i++)
    {
        if (arr[i] > arr[i-1])
            inc[i] = inc[i-1] + 1;
        else
            inc[i] = 1;
    }

    // Step 2) Construct decreasing sequence array
    for (i = n-2; i >= 0; i--)
    {
        if (arr[i] > arr[i+1])
            dec[i] = dec[i+1] + 1;
        else
            dec[i] = 1;
    }

    // Step 3) Find the length of maximum length bitonic sequence
    max = inc[0] + dec[0] - 1;
    for (i = 1; i < n; i++)
    {
        if (inc[i] + dec[i] - 1 > max)
        {
            max = inc[i] + dec[i] - 1;
        }
    }

    // free dynamically allocated memory
    delete [] inc;
    delete [] dec;

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {12, 4, 78, 90, 45, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("\n Length of max length Bitnoic Subarray is %d", bitonic(arr, n));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Auxiliary Space: O(n)

50. 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.

```
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
   int max = arr[low];
   int i;
   for (i = low; i <= high; i++)
   {
       if (arr[i] > max)
          max = arr[i];
   }
   return max;
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

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}

```
#include <stdio.h>

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);
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 3, 50, 10, 9, 7, 6};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

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}.

51. 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, 1}
Output:  countSmaller[]  =  {6, 1, 2, 2, 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[].

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

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

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

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

  printf("\n");
}

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

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Method 2 (Use BST)

A Self Balancing Binary Search Tree (AVL, Red Black,.. etc) can be used to get the solution in O(nLogn) time complexity. A Self Balancing Binary Search Tree(BST) can be augmented to contain count of smaller values in every node. We can store this info by storing count of nodes in left subtree. The BST insert function must be augmented in such a way that it increments the count for those ancestors for which, the node being inserted is in left subtree. The BST delete function must also be augmented in such a way that decrements the count for those ancestors for which, the node being deleted is in left subtree. Both of these operations can still be done in O(Logn) complexity.

Following is the algorithm that constructs countSmaller[] using the above Augmented Self Balancing BST.

1) Traverse the array from left to right and construct a BST such that every node of BST also contains count of nodes in left subtree. O(nLogn)
2) Traverse the array again from left to right and do following for every array element arr[i] .
…..a) Search the element arr[i] in BST, get the count of nodes in left subtree and store it in countSmaller[i]
…..b) Delete the node (which contains arr[i]) from BST

The step 2 b) is necessary to make sure that the smaller nodes on left side (in array) are deleted before we consider count for a node.

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

52. Minimum number of jumps to reach end

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

Example:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 ->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.

Method 1 (Naive Recursive Approach)
A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

minJumps(start, end) = Min (minJumps(k, end)) for all k reachable from start

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

// Returns minimum number of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int l, int h)
{
   // Base case: when source and destination are same
   if (h == l)
     return 0;

   // When nothing is reachable from the given source
   if (arr[l] == 0)
     return INT_MAX;

   // Traverse through all the points reachable from arr[l]. Recursively
   // get the minimum number of jumps needed to reach arr[h] from these
   // reachable points.
   int min = INT_MAX;
   for (int i = l+1; i <= h && i <= l + arr[l]; i++)
   {
       int jumps = minJumps(arr, i, h);
       if(jumps != INT_MAX && jumps + 1 < min)
           min = jumps + 1;
   }

   return min;
}

// Driver program to test above function
int main()
{
  int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-
1));
  return 0;
}
```

If we trace the execution of this method, we can see that there will be overlapping subproblems. For example, minJumps(3, 9) will be called two times as arr[3] is reachable from arr[1] and arr[2]. So this problem has both properties (optimal substructure and overlapping subproblems) of Dynamic Programming.

Method 2 (Dynamic Programming)

In this method, we build a jumps[] array from left to right such that jumps[i] indicates the minimum number of jumps needed to reach arr[i] from arr[0]. Finally, we return jumps[n-1].

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

```
#include <limits.h>

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    int *jumps = new int[n];  // jumps[n-1] will hold the result
    int i, j;

    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++)
    {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++)
        {
            if (i <= j + arr[j] && jumps[j] != INT_MAX)
            {
                 jumps[i] = jumps[j] + 1;
                 break;
            }
        }
    }
    return jumps[n-1];
}

// Driver program to test above function
int main()
{
    int arr[]= {1, 3, 6, 1, 0, 9};
    int size=sizeof(arr)/sizeof(int);
    printf("Minimum number of jumps to reach end is %d ",
minJumps(arr,size));
    return 0;
}
```

Time Complexity: O(n^2)

Method 3 (Dynamic Programming)

In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return arr[0].

```
int minJumps(int arr[], int n)
{
    int *jumps = new int[n];  // jumps[0] will hold the result
    int min;

    // Minimum number of jumps needed to reach last element
    // from last elements itself is always 0
    jumps[n-1] = 0;

    int i, j;

    // Start from the second element, move from right to left
    // and construct the jumps[] array where jumps[i] represents
    // minimum number of jumps needed to reach arr[m-1] from arr[i]
    for (i = n-2; i >=0; i--)
    {
        // If arr[i] is 0 then arr[n-1] can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can direcly reach to the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum number of jumps needed
        // to reach arr[n-1], check all the points reachable from here
        // and jumps[] value for those points
        else
        {
            min = INT_MAX;  // initialize min value

            // following loop checks with all reachable points and
            // takes the minimum
            for (j = i+1; j < n && j <= arr[i] + i; j++)
            {
                if (min > jumps[j])
                    min = jumps[j];
            }

            // Handle overflow
            if (min != INT_MAX)
              jumps[i] = min + 1;
            else
              jumps[i] = min; // or INT_MAX
        }
    }

    return jumps[0];
}
```

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