- Existence of such number indicates that we are able to find the indexes.
- Otherwise proceed to the next input element.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-13 Given an array A of n elements. Find three elements, i, j and k in the array such that $A[i]^2 + A[j]^2 = A[k]^2$?

Solution:

Algorithm

- For each array index i compute $A[i]^2$ and store in array.
- Now, the problem reduces to finding three indexes , i, j and k such that $A[i] + A[j] = A[k]$. This is same as that of Problem-9.

Problem-14 Two elements whose sum is closest to zero

Given an array with both positive and negative numbers. We need to find the two elements such that their sum is closest to zero. For the below array, algorithm should give -80 and 85.

Example: $1\ 60\ -10\ 70\ -80\ 85$

Solution: Brute Force Solution.

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

```
void TwoElementsWithMinSum(int A[], int n)
{
        int inv_count = 0;
        int i, j, min_sum, sum, min_i, min_j;

        if(n < 2)
        {
                printf("Invalid Input");
                return;
        }

        /* Initialization of values */
        min_i = 0;
        min_j = 1;
        min_sum = A[0] + A[1];
```

```
        for(i= 0; i < n - 1; i ++)
        {
                for(j = i + 1; j < n; j++)
                {
                        sum = A[i] + A[j];
                        if(abs(min_sum) > abs(sum))
                        {
                                min_sum = sum;
                                min_i = i;
                                min_j = j;
                        }
                }
        }
        printf(" The two elements are %d and %d", arr[min_i], arr[min_j]);
}
```

Time complexity: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-15 Can we improve the time complexity of Problem-14?

Solution: Use Sorting.

Algorithm

1) Sort all the elements of the given input array.
2) Find the two elements on either side of zero (if they are all positive or all negative then we are done with the solution)
3) If the one is positive and other is negative then add the two values at those positions. If the total is positive then increment the negative index, if it is negative then increment the positive index. If it is zero then stop.
4) loop step (3) until we hit a zero total or reached the end of array. Store the best total as you go.

Time Complexity: $O(nlogn)$, for sorting.

Problem-16 Given an array of n elements. Find three elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach.

The default solution to this is, for each pair of input elements check whether there is any element whose sum is K. This we can solve just by using three simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int data)
{
        int i =  0, j = 0, k = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        for(k = j+1; k < n; k++)
                        {
                                if(A[i] + A[j] + A[k]== data)
                                {
                                        printf("Items Found:%d %d %d", i, j, k);
                                        return;
                                }
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^3)$. This is because of three nested for loops.
Space Complexity: $O(1)$.

Problem-17 Does the solution of Problem-16 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we can find three numbers whose sum is K if they exist.

Problem-18 Can we use sorting technique for solving Problem-16*?*

Solution: Yes.

```
void Search[int A[], int n, int data)
{
        int i, j;
        Sort(A, n);
        for(k = 0; k < n; k++)
        {
                for(i =  k + 1, j = n-1; i < j;  )
                {
                        if (A[k] + A[i] + A[j]  == data)
                        {
                                printf("Items Found:%d %d %d", i, j, k);
```

```
                        return;
                    }
                    else if (A[k] + A[i] + A[j] < data)
                        i = i + 1;
                    else
                        j = j - 1;
                }
            }
            return;
}
```

Time Complexity: Time for sorting + Time for searching in sorted list = $O(nlogn) + O(n^2) \approx O(n^2)$. This is because of two nested *for* loops.
Space Complexity: $O(1)$.

Problem-19 Can we use hashing technique for solving Problem-16*?*

Solution: Yes. Since our objective is to find three indexes of the array whose sum is K. Let us say those indexes are X, Y and Z. That means, $A[X] + A[Y] + A[Z] = K$.

Let us assume that we have kept all possible sums along with their pairs in hash table. That means the key to hash table is $K - A[X]$ and values for $K - A[X]$ are all possible pairs of input whose sum is $K - A[X]$.

Algorithm

- Before starting the searching, insert all possible sums with pairs of elements into the hash table.
- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: Time for storing all possible pairs in Hash table + searching = $O(n^2) + O(n^2) \approx O(n^2)$.
Space Complexity: $O(n)$.

Problem-20 Given an array of n integers, the $3 - sum\ problem$ is to determine find three integers whose sum is closest to $zero$.

Solution: This is same as that of Problem-16. In this case, K value is zero.

Problem-21 Given an array of n numbers. Give an algorithm for finding the element which appears maximum number of times in the array?

Solution: Brute Force.

One simple solution to this is, for each input element check whether there is any element with same value and for each such occurrence, increment the counter. Each time, check the current counter with the max counter and update it if this its value is greater than max counter. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
int CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
        int counter =0, max=0;
        for(i = 0; i < n; i++)
        {
                counter=0;
                for(j = 0; j < n; j++)
                {
                        if(A[i] == A[j])
                                counter++;
                }
                if (counter > max)
                        max = counter;
        }
        return max;
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-22 Can we improve the complexity of Problem-21 solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see which element is appearing maximum number of times.

Time Complexity: $O(nlogn)$. (for sorting).
Space Complexity: $O(1)$.

Problem-23 Is there any other way of solving Problem-21?

Solution: Yes, using hash table. For each element of the input keep track of how many times that element appeared in the input. That means the counter value represents the number of occurrences for that element.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-24 For Problem-21, can we improve the time complexity? Assume that the elements range is o to $n-1$. That means all the elements are within this range only.

Solution: Yes. We solve this problem in two scans. We *cannot* use the negation technique of Problem-3 for this problem because of number of repetitions.

In the first scan, instead of negating we add the value n. That means for each of occurrence of an element we add the array size to that element.

In the second scan we check the element value by dividing it with n and we return the element whichever gives the maximum value. The code based on this method is given below.

```
void MaxRepititions(int A[], int n)
{
        int i =  0;
        int max = 0;
        for(i = 0; i < n; i++)
        {
                A[A[i]%n] +=n;
        }
        for(i = 0; i < n; i++)
        {
                if(A[i]/n > max)
                {
                        max = A[i]/n;
                        max =i;
                }
        }
        return max;
}
```

Note:

- This solution does not work if the given array is read only.
- This solution will work only if the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

Time Complexity: $O(n)$. Since no nested for loops are required.

Space Complexity: $O(1)$.

Problem-25 Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $1 \leq k \leq n$ such that $A[1], \ldots, A[k]$ is an increasing sequence and $A[k+1], \ldots, A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding k.

Similar question:

Lets us assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing function]. In this array find the starting index of the positive numbers. Let us assume that we know the length of the input array. Design a $O(logn)$ algorithm.

Solution: We use a variant of the binary search.

```
int Search (int A[], int first, int last)
{
        int first = 0;
        int last = n-1;
        int mid;

        while(first <= last)
        {
                // if the current array has size 1
                if(first == last)
                        return A[first];
                // if the current array has size 2
                else if(first == last-1)
                        return max(A[first], A[last]);
                // if the current array has size 3 or more
                else
                {
                        mid = first + (last-first)/2;

                        if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                                return A[mid];
                        else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                                first = mid+1;
                        else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                                last = mid-1;
                        else
                                return INT_MIN ;
                } // end of else
        } // end of while
```

}

The recursion equation is $T(n) = 2T(n/2) + c$. Using master theorem, we get $O(logn)$.

Problem-26 If we don't know n, how do we solve the Problem-25?

Solution: Repeatedly compute $A[1], A[2], A[4], A[8], A[16]$, and so on until we find a value of n such that $A[n] > 0$.

Time Complexity: $O(logn)$, since we are moving at the rate of 2.
Refer *Introduction to Analysis of Algorithms* chapter for details on this.

Problem-27 Given an input array of size unknown with all $1's$ in the beginning and $0's$ in the end. Find the index in the array from where $0's$ start. Consider there are millions of $1's$ and $0's$ in the array. E.g. array contents 1111111.....1100000.....0000000.

Solution: This problem is almost similar to Problem-26. Check the bits at the rate of $2^K$where $k = 0, 1, 2$ ....

Since we are moving at the rate of 2, the complexity is $O(logn)$.

Problem-28 Given a sorted array of n integers that has been rotated an unknown number of times, give a $O(log\ n)$ algorithm that finds an element in the array.
Example: Find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14)
Output: 8 (the index of 5 in the array)

Solution: Let us assume that the given array is $A[]$. Using solution of Problem-25, with extension. The below function $FindPivot$ returns the k value (let us assume that this function return the index instead of value). 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 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

Algorithm

1) Find out pivot point and divide the array in two sub-arrays.
2) Now call binary search for one of the two sub-arrays.
    a. if element is greater than first element then search in left subarray
    b. else search in right subarray
3) If element is found in selected sub-array then return index *else* return -1.

```
int FindPivot(int A[], int start, int finish)
{
```

```
        if(finish - start == 0)
                return start;
        else if(start) == finish - 1)
        {
                if (A[start] >= A[finish])
                        return start;
                else
                        return finish;
        }
        else
        {
                mid = start + (finish-start)/2;
                if (A[mid] >= A[mid + 1])
                        return FindPivot(A, start, mid);
                else
                        return FindPivot(A, mid, finish);
        }
}

int Search(int A[], int n, int x)
{
        int pivot = FindPivot(A, 0, n-1);
        if(A[pivot] == x)
                return pivot;
        if(A[pivot] <= x)
                return BinarySearch(A, 0, pivot-1, x);
        else
                return BinarySearch(A, pivot+1, n-1, x);
}

int BinarySearch(int A[], int low, int high, int x)
{
        if(high >= low)
        {
                int mid = low + (high - low)/2;

                if(x == A[mid])
                        return mid;
                if(x > A[mid])
                        return BinarySearch(A, (mid + 1), high, x);
                else
                        return BinarySearch(A, low, (mid -1), x);
        }
        /*Return -1 if element is not found*/
```

```
        return -1;
}
```

Time complexity:$O(logn)$.

Problem-29 For Problem-28, can we solve in one scan?

Solution: Yes.

```
int BinarySearchRotated(int A[], int start, int finish, int data)
{
        int mid;
        if (start > finish)
                return -1;

        mid = start + (finish - start) / 2;

        if (data == A[mid])
                return mid;
        else if (A[start] <= A[mid])
        {       // start half is in sorted order.
                if (data >= A[start] && data < A[mid])
                        return BinarySearchRotated(A, start, mid - 1, data);
                else
                        return BinarySearchRotated(A, mid + 1, finish, data);
        }
        else
        {       // A[mid] <= A[finish], finish half is in sorted order.
                if (data > A[mid] && data <= A[finish])
                        return BinarySearchRotated(A, mid + 1, finish, data);
                else
                        return BinarySearchRotated(A, start, mid - 1, data);
        }
}
```

Time complexity:$O(logn)$.

Problem-30 Bitonic search.

An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array A of n distinct integers, describe how to determine whether a given integer is in the array in $O(logn)$ steps.

Solution: This is same as Problem-25.

Problem-31 Yet, other way of asking Problem-25?
Let $A[]$ be an array that starts out increasing, reaches a maximum, and then decreases. Design an $O(logn)$ algorithm to find the index of the maximum value.

Problem-32 Give an $O(nlogn)$ algorithm for computing the median of a sequence of n integers.

Solution: Sort and return element at $n/2$.

Problem-33 Given two sorted lists of size m and n, find the median of all elements in $O(log\ (m + n))$ time.

Solution: Refer *Divide and Conquer* chapter.

Problem-34 Give a sorted array A of n elements, possibly with duplicates, find the index of the first occurrence of a number in $O(logn)$ time.

Solution: To find the first occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == low && A[mid] == data || A[mid] == data && A[mid-1] < data

```
int BinarySearchFirstOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
                        return mid;

                // Give preference to left half of the array
                else if (A[mid] >= data)
                        return BinarySearchFirstOccurrence (A, n, low, mid - 1, data);
                else
                        return BinarySearchFirstOccurrence (A, n, mid + 1, high, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-35 Give a sorted array A of n elements, possibly with duplicates, find the index of the last occurrence of a number in $O(logn)$ time.

Solution: To find the last occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == high && A[mid] == data || A[mid] == data && A[mid+1] > data

```
int BinarySearchLastOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
                        return mid;

                // Give preference to right half of the array
                else if (A[mid] <= data)
                        return BinarySearchLastOccurrence (A, n, mid + 1, high, data);
                else
                        return BinarySearchLastOccurrence (A, n, low, mod - 1, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-36 Give a sorted array of n elements, possibly with duplicates, find the number of occurrences of a number.

Solution: Brute For Approach.

Do a linear search over the array and increment count as and when we find the element data in the array.

```
int LinearSearchCount(int A[], int n, int data)
{
        int count = 0;

        for (int i = 0; i < n; i++)
        {
                if (a[i] == k)
                        count++;
        }
        return count;
```

}

Time Complexity: $O(n)$.

Problem-37 Can we improve the time complexity of Problem-36?

Solution: Yes. We can solve this by using one binary search call followed by another small scan.

Algorithm

- Do a binary search for the $data$ in the array. Let us assume its position be K.
- Now traverse towards left from K and count the number of occurrences of $data$. Let this count be $leftCount$.
- Similarly, traverse towards right and count the number of occurrences of $data$. Let this count be $rightCount$.
- Total number of occurrences $= leftCount + 1 + rightCount$

Time Complexity – $O(logn + S)$ where S is the number of occurrences of $data$.

Problem-38 Is there any alternative way of solving the Problem-36*?*

Solution:

Algorithm

- Find the first occurrence of $data$ and call its index as $firstOccurrence$ (for algorithm refer Problem-34)
- Find the last occurrence of data and call its index as $lastOccurrence$ (for algorithm refer Problem-35)
- Return $lastOccurrence - firstOccurrence + 1$

Time Complexity $= O(logn + logn) = O(logn)$.

Problem-39 What is the next number in the sequence $1, 11, 21$ and why?

Solution: Read the given number loudly. This is just a fun problem.

One one
Two Ones
One two, one one→ 1211

So answer is, the next number is the representation of previous number by reading it loudly.

Problem-40 Finding second smallest number efficiently.

Solution: We can construct a heap of the given elements using up just less than n comparisons (Refer *Priority Queues* chapter for algorithm). Then we find the second smallest using $logn$ comparisons for the GetMax() operation. Overall, we get $n + logn + constant$.

Problem-41 Is there any other solution for Problem-40?

Solution: Alternatively, split the n numbers into groups of 2, perform $n/2$ comparisons successively to find the largest using a tournament-like method. The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones the maximum popped. This will yield $logn - 1$ comparisons for a total of $n + lg\, n - 2$. The above solution is called as *tournament problem*.

Problem-42 An element is a majority if it appears more than $n/2$ times. Give and algorithm takes an array of n element as argument and identifies a majority (if it exists).

Solution: 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: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-43 Can we improve the Problem-42 time complexity to $O(nlogn)$?

Solution: Using binary search we can achieve this.

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

```
struct TreeNode
{
        int element;
        int count;
        struct TreeNode *left;
        struct TreeNode *right;
}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, and 4}.

Time Complexity: If a binary search tree is used then worst time complexity will be $O(n^2)$. If a balanced-binary-search tree is used then $O(nlogn)$.

Space Complexity: $O(n)$.

Problem-44 Is there any other of achieving $O(nlogn)$ complexity for the Problem-42?

Solution: Sort the input array and scan the sorted array to find the majority element.

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

Problem-45 Can we improve the complexity for the Problem-42?

Solution: If an element occurs more than $n/2$ times in A then it must be the median of A. But, the reverse is not true, so once the median is found, we must check to see how many times it occurs in A. We can use linear selection which takes $O(n)$ time (for algorithm refer *Selection Algorithms* chapter).

```
int CheckMajority(int A[], in n)
{
        1) Use linear selection to find the median m of A.
        2) Do one more pass through A and count the number of occurrences of m.
            a. If m occurs more than n/2 times then return true;
            b. Otherwise return false.
}
```

Problem-46 Is there any other way of solving the Problem-42?

Solution: Since only one element is repeating, we can use simple scan of the input array by keeping track of count for the elements. If the count is 0 then we can assume that the element is coming first time otherwise that the resultant element.

```
int MajorityNum(int[] A, int n)
{
    int majNum, count;
    element = -1; count = 0;
    for(int i = 0; i < n; i++)
    {
            // If the counter is 0 then set the current candidate to majority num and
            // we set the counter to 1.
            if(count == 0)
            {
                    element = A[i];
```

```
                count = 1;
            }
            else if(element == A[i])
            {
                // Increment counter If the counter is not 0 and
                // element is same as current candidate.
                count++;
            }
            else
            {
                // Decrement counter If the counter is not 0 and
                // element is different from current candidate.
                count--;
            }
    }
    return element;
}
```

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

Problem-47 Given an array of $2n$ elements of which n elements are same and the remaining n elements are all different. Find the majority element.

Solution: The repeated elements will occupy half the array. No matter what arrangement it is, only one of the below will be true,

- All duplicate elements will be at a relative distance of 2 from each other. Ex: n, 1, n, 100, n, 54, n ...
- At least two duplicate elements will be next to each other
  Ex: $n, n, 1, 100, n, 54, n, \ldots.$
  $n, 1, n, n, n, 54, 100 \ldots$
  $1, 100, 54, n, n, n, n \ldots.$

So, in worst case, we need will two passes over the array,

First Pass: compare $A[i]$ and $A[i+1]$
Second Pass: compare $A[i]$ and $A[i+2]$

Something will match and that's your element.

This will cost $O(n)$ in time and $O(1)$ in space.

Problem-48 Given an array with $2n + 1$ integer elements, n elements appear twice in arbitrary places in the array and a single integer appears only once somewhere inside. Find the lonely integer with $O(n)$ operations and $O(1)$ extra memory.

Solution: Since except one element all other elements are repeated. We know that $A\ XOR\ A\ = 0$. Based on this if we XOR all the input elements then we get the remaining element.

```
int solution(int* A)
{
        int i, res;
        for (i = res = 0; i < 2n+1; i++)
                res = res ^ A[i];
         return res;
}
```

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

Problem-49 Throwing eggs from an n-story building.

Suppose that we have an n story building and a set of eggs. Also assume that an egg breaks if it is thrown off floor F or higher, and will not break otherwise. Devise a strategy to determine the floor F, while breaking $O(logn)$ eggs.

Solution: Refer *Divide and Conquer* chapter.

Problem-50 Local minimum of an array.

Given an array A of n distinct integers, design an $O(logn)$ algorithm to find a *local minimum*: an index i such that $A[i-1] < A[i] < A[i+1]$.

Solution: Check the middle value $A[n/2]$, and two neighbors $A[n/2 - 1]$ and $A[n/2 + 1]$. If $A[n/2]$ is local minimum, stop; otherwise search in half with smaller neighbor.

Problem-51 Give an $n \times n$ array of elements such that each row is in ascending order and each column is in ascending order, devise an $O(n)$ algorithm to determine if a given element x in the array. You may assume all elements in the $n \times n$ array are distinct.

Solution: Let us assume that the given matrix is $A[n][n]$. Start with the last row, first column [or first row - last column]. If the element we are searching for is greater than the element at $A[1][n]$, then the column 1 can be eliminated. If the search element is less than the element at $A[1][n]$, then the last row can be completely eliminated. Now, once the first column or the last row is eliminated, now, start over the process again with left-bottom end of the remaining array. In this algorithm, there would be maximum n elements that the search element would be compared with.

Time Complexity: $O(n)$. This is because we will traverse at most 2 n points.

Space Complexity: $O(1)$.

Problem-52 Given an $n \times n$ array a of n^2 numbers, Give an $O(n)$ algorithm to find a pair of indices i and j such that $A[i][j] < A[i+1][j], A[i][j] < A[i][j+1], A[i][j] < A[i-1][j]$, and $A[i][j] < A[i][j-1]$.

Solution: This problem is same as Problem-51.

Problem-53 Given $n \times n$ matrix, and in each row all 1's are followed 0's. Find row with maximum number of 0's.

Solution: Start with first row, last column. If the element is 0 then move to the previous column in the same row and at the same time increase the counter to indicate the maximum number of 0's. If the element is 1 then move to the next row in the same column. Repeat this process until we reach last row, first column.

Time Complexity: $O(2n) \approx O(n)$ (very much similar to Problem-51).

Problem-54 Given an input array of size unknown with all numbers in the beginning and special symbols in the end. Find the index in the array from where special symbols start.

Solution: Refer *Divide and Conquer* chapter.

Problem-55 Finding the Missing Number

We 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. Given an algorithm to find the missing integer.

Example:

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

O/P 5

Solution: Use sum formula

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

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-56 In Problem-55, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Can we solve this problem?

Solution:

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

```
int FindMissingNumber(int A[], int n)
{
        int i, X, Y;
        for (i = 0; i < 9; i ++)
                X ^= A[i];
        for (i = 1; i <= 10; i ++)
                Y ^= i;
        //In fact, one variable is enough.
        return X ^ Y;
}
```

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-57 Find the Number Occurring Odd Number of Times

Given an array of positive integers, all numbers occurs 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

Solution: Do a bitwise XOR of all the elements. Finally we get the number which has odd occurrences. This is because of the fact that, $A\ XOR\ A = 0$.

Time Complexity: $O(n)$.

Problem-58 Find the two repeating elements in a given array

Given an array with $n + 2$ elements, all elements of the array are in range 1 to n and also all elements occur only once except two numbers which occur twice. Find those two repeating numbers.

Example: $6, 2, 6, 5, 2, 3, 1$ and $n = 5$

The above input has $n + 2 = 7$ elements with all elements occurring once except 2 and 6 which occur twice. So the output should be $6\ 2$.

Solution: One simple way to scan the complete array for each element of the input elements. That means use two loops. In the outer loop, select elements one by one and count the number of occurrences of the selected element in the inner loop.

```
void PrintRepeatedElements(int A[], int n)
{
        int i, j;
        for(i = 0; i < n; i++)
                for(j = i+1; j <n; j++)
                        if(A[i] == A[j])
                                printf("%d", A[i]);
}
```

Time Complexity: $O(n^2)$.
Space Complexity: $O(1)$.

Problem-59 For the Problem-58, can we improve the time complexity?

Solution: Sort the array using any comparison sorting algorithm and see if there are any elements which contiguous with same value.

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

Problem-60 For the Problem-58, can we improve the time complexity?

Solution: Use Count Array. This solution is like using a hash table. But for simplicity we can use array for storing the counts. 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 we see an element whose count is already set, print it as duplicate.

```
void PrintRepeatedElements(int A[], int n)
{
        int *count = (int *)calloc(sizeof(int), (n - 2));
        for(int i = 0; i < size; i++)
        {
                if(count[A[i]] == 1)
                        printf("%d", A[i]);
                else
                        count[A[i]]++;
        }
}
```

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-61 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there any other way of solving the problem?

Solution: Using XOR Operation. 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 k^{th} bit of $X\ XOR\ Y$ is 1, we can XOR all the elements in the array and all integers from 1 to n, whose k^{th} bits are 1. The result will be one of X and Y.

```
void PrintRepeatedElements (int A[], int size)
{
        int XOR = A[0];
        int right_most_set_bit_no;
        int n = size - 2;
        int X= 0, Y = 0;

        /* Compute XOR of all elements in A[]*/
        for(int i = 0; i < n; i++)
                XOR ^= A[i];

        /* Compute XOR of all elements {1, 2 ..n} */
        for(i = 1; i <= n; i++)
                XOR ^= i;

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

        /* Now divide elements in two sets by comparing rightmost set */
        for(i = 0; i < n; i++)
        {
                if(A[i] & right_most_set_bit_no)
                        X = X^ A[i]; /*XOR of first set in A[] */
                else
                        Y = Y ^ A[i]; /*XOR of second set inA[] */
        }
        for(i = 1; i <= n; i++)
        {
                if(i & right_most_set_bit_no)
                        X = X ^ i; /*XOR of first set in A[] and {1, 2, ...n }*/
                else
                        Y = Y ^ i; /*XOR of second set in A[] and {1, 2, ...n } */
        }

        printf("%d and %d",X, Y);
}
```

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

Problem-62 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there yet other way of solving the problem?

Solution: We can solve this by creating two simple mathematical equations. Let us assume that two numbers which we are going to find are X and Y. We know the sum of n numbers is $n(n+1)/2$ and product is $n!$. Make two equations using these sum and product formulae, and get values of two unknowns using the two equations.

Let summation of all numbers in array be S and product be P and the numbers which are being repeated are X and Y.

$X + Y = S - n(n+1)/2$
$XY = P/n!$

Using above two equations, we can find out X and Y.
There can be addition and multiplication overflow problem with this approach.

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

Problem-63 Similar to Problem-58. Let us assume that the numbers are in the range 1 to n. Also, $n-2$ elements are repeating thrice and remaining two elements are repeating twice. Find the element which is repeating twice.

Solution: If we xor all the elements in the array and all integers from 1 to n, then the all the elements which are trice will become zero This is because, since the element is repeating trice and XOR with another time from range makes that element appearing four times. As a result, output of a $XOR\ a\ XOR\ a\ XOR\ a = 0$. Same is case with all elements which repeated thrice.

With the same logic, for the element which repeated twice, if we XOR the input elements and also the range, then the total number of appearances for that element is 3. As a result, output of $a\ XOR\ a\ XOR\ a = a$. Finally, we get the element which repeated twice.

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

Problem-64 **Separate 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, 90, 8, 9, 45, 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.

Solution: The problem is very similar to *Separate 0's and 1's (*Problem-65*)* in an array, and both of these problems are variation of famous *Dutch national flag problem*.

Algorithm: Logic is little similar to Quick sort.

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

Implementation:

```
void DutchNationalFlag(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right)
        {
                /* Increment left index while we see 0 at left */
                while(A[left]%2 == 0 && left < right)
                        left++;

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

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

Time Complexity: $O(n)$.

Problem-65 Other way of asking Problem-64 but with little difference.

Separate 0's and 1's in an array

We are given an array of 0's and 1's in random order. Separate 0's on left side and 1's 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]

Solution: Counting 0's or 1's

1. Count the number of $0's$. Let count be C.
2. Once we have count, we can put C 0's at the beginning and $1's$ at the remaining $n - C$ positions in array.

Time Complexity: $O(n)$. This solution scans the array two times.

Problem-66 Can we solve the Problem-65 in once scan?

Solution: Yes. Use two indexes to traverse:

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

Do following while $left < right$:

1) Keep incrementing index left while there are 0s at it
2) Keep decrementing index right while there are 1s at it
3) If left < right then exchange $A[left]$ and $A[right]$

```
/*Function to put all 0s on left and all 1s on right*/
void Separate0and1(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
      while(left < right)
      {
            /* Increment left index while we see 0 at left */
            while(A[left] == 0 && left < right)
                  left++;

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

            /* If left is smaller than right then there is a 1 at left
            and a 0 at right.  Swap A[left] and A[right]*/
            if(left < right)
            {
```

```
                    A[left] = 0;
                    A[right] = 1;
                    left++;
                    right–;
                }
            }
}
```

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

Problem-67 Maximum difference between two elements

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

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 (Difference between 7 and 9)

Solution: Refer *Divide and Conquer* chapter.

Problem-68 Given an array of 101 elements. Out of them 25 elements are repeated twice, 12 elements are repeated 4 times and one element is repeated 3 times. Find the element which repeated 3 times in $O(1)$.

Solution: Before solving this problem let us consider the following XOR operation property.

$$a\ XOR\ a = 0$$

That means, if we apply the XOR on same elements then the result is 0. Let us apply this logic for this problem.

Algorithm:

- XOR all the elements of the given array and assume the result is A.
- After this operation, 2 occurrences of number which appeared 3 times becomes 0 and one occurrence will remain.
- The 12 elements which are appearing 4 times become 0.
- The 25 elements which are appearing 2 times become 0.

So just $XOR'ing$ all the elements give the result.

Time Complexity: $O(n)$, because we are doing only once scan.
Space Complexity: $O(1)$.