1. Divide-and-conquer, binary search, sorted array A, # of element is N.
Basic idea :

Find middle element of array A with index $\frac{\lfloor N \rfloor}{2}$

3 case :

case 1 : A $\frac{\lfloor N \rfloor}{2}$ = $\frac{\lfloor N \rfloor}{2}$, find. Return 1. (found)

case 2 : A $\frac{\lfloor N \rfloor}{2}$ < $\frac{\lfloor N \rfloor}{2}$, recursive on right half array exclusing A $\frac{\lfloor N \rfloor}{2}$

case 3 : A $\frac{\lfloor N \rfloor}{2}$ > $\frac{\lfloor N \rfloor}{2}$, recursive on left half array exclusing A $\frac{\lfloor N \rfloor}{2}$

base case :

N = 0. Return –1 (No found)

Pseudo algorithm :

```
find ( A, head, tail, size ) {
        if ( size = 0 )  return –1;
        int  k = ⌊N⌋/2
        If ( A[k] = k )   return K;
        else if ( A[k] < k )
                find ( A, k+1, tail, size-k );
        else
                find ( A, head, k-1, size-k);
}
```

running time binary search. O (log n)

10. a. Input set S, n real numbers. x is given.
Set S is unsorted.
Basic idea :

Let's be an array of size n. Perform merger sort on S. Had S sorted. This Take O (n log n) time. Now we get sorted set S. Let a = S[i]. Let b = S[j], i = 0, J = n-1 initially and based on the value of (a+b), we do following 3 operation :

1) case 1. if a+b = x return (i,j).
2) case 2. if a+b > x then we decrease j by 1, i.e. j--;
3) case 3. if a+b < x then we increase i by 1, i.e. i++;
4) base case : if I = j, return No_Found

repeat step 2 and 3 until program return (eight 1 or 3 be executed). This find algorithm run O(n) time, so the total run time is O(n log n).

Pseudo Algorithm :

Mergesort(S); (you can find it at any algorithm book)

```
find(S,x,n){
        i = 0, j = n-1;
        while ( i != j ){
                a = S[i];
                b = S[j];
                if ( a+b = x )   return (i,j);
                else if ( a+b > x )  j--;
                else  i++;
        }
        return No_Found;
}
```

b. Since S is given in a sorted order, the "find" algorithm in part (a) run in time O(n). refer to the above algorithm !

Problem # 3

Let $a = \log^3\log(n)$

We know that an element in position "i" has an actual position of

$i - a <= P[i] <= i + a$

Hence if we sort a "2a" length then element "i" will find its correct location.

Algorithm:

1. Divide the array into "n / a" divisions each of length "a"
2. Starting from one side sort the first "2a elements"
3. This results in putting the first "a" elements in their correct positions
4. Now from "a+1st " element repeat this process. Incrementing by "a" on each step
5. The final result is a sorted array

Complexity:

Use merge sort to sort the blocks of "2a elements" : *O(a log(a))*
Do this "n / a " times:

=> complexity: O(n log(a))

Problem # 4:

There are two steps for doing this.

1. Sort the input array
2. For all i sum up the element i with n+1-i element. (first with last, second with second last and so on).

Complexity:

Sorting: O(n log(n))
Sum in a loop: O (n)

=> complexity is O(n log(n))

Verification by contradiction:

Assume that we do not sum up the smallest element with the largest element then this would mean that an element greater than the smallest element will be paired up with the last element. This will give a sum which will be greater than (or equal to) the sum of the smallest and largest element because the new element that we chose was bigger than (or equal to) the smallest element. To generalize this result, assume each instance of the problem as in input to our algorithm. Hence at each instance step 2 will give the best solution.

Problem # 5

Do a depth first search on the tree with the following rules:

1. Keep a count of the number of nodes visited.
2. Do not traverse a node with value less than x (hence if root is smaller than "x" then it goes without saying that the k^{th} largest element will be smaller too!)
3. Stop traversing when the count of nodes visited becomes "k"
4. If you are able to traverse "k" elements (which are all larger than "x" as given by rule 2) then the k^{th} largest element is greater than "x" (or equal to "x" if the last element you found was equal to "x")
5. If you had to stop traversing before the node traversal count reached "k" this would mean that k^{th} largest element is smaller than "x", as you found only "a" elements larger than "x" where " $a < k$ "

Complexity: Step 3 enforces that time complexity is O (k).