1. Solution:

Data:

Stack: the stack.
Top: point to the top box of the stack.
Max: representation of top of the stack.
Elem: element that we want to insert on top of the stack.

Pseudocode for push/insert:

```
push(Stack,Top,Max,Elem)
if Top=Max
    print overflow and return
set Top=Top+1
set Stack[top]=Elem
return
```

Pseudocode of pop/remove :

```
pop(Stack,Top,Elem)
if top==0
    print underflow and return
set Elem=Stack[top]
set Top=Top-1
return
```

Pseudocode of Findmin:

```
if top(Stack)<top(Auxiliary Stack)
    {
    push in Auxiliary Stack
    }
if popElement==top(Auxiliary Stack)
    {
    pop
    }
```

Here, there are two stacks named stack and auxiliary stack. The auxiliary stack is a temporary stack for finding the minimum element in the stack. If the element at the top of the stack is less than the top of the auxiliary stack then element would be pushed in the auxiliary stack and if the pop element is equal to the top of the auxiliary stack then we pop out the element.

Time Complexity Analysis:

For push/insert operation: O(1) (As insertion 'push' in a stack takes constant time)
For pop/remove operation: O(1) (As deletion 'pop' in a stack takes constant time)
For 'FindMin' operation: O(1) (As we have used an auxiliary stack which has its top as the minimum element)
So the time complexity of this stack is constant.

2. **Solution:**

We can start by sorting the list, which takes O(n log n) time. Then move all the way through the list, keeping track of which candidate has the most votes so far, and how many votes they have received. If the number of votes for the current candidate exceeds the previous maximum, make it the new maximum. The second (counting) stage takes O(n) time, so the whole process takes O(n log n) time.

3. **Solution:**

It is given that a node contains an element, auxiliary and a pointer to its parent. We need to find out a path from node x to node y in O(k) where k is the length of the path. This can be easily done if we traverse from the child node to the parent as every child node will have a single parent and will lead us to the targetted parent in O(k).

Algorithm(pseudocode) :

```
Begin:
    b= child_node
    a = parent_node
    PathFromatob = empty list
    temp = b
    push temp in PathFromatob
    while temp is not equal to a:
        temp = temp's parent node
        push temp in PathFromatob
    end while
    return PathFromatob
End
```

Since we don't need to use the auxiliary, no changes are needed to be made to the tree. The run time of the given algorithm is of the order of the length of the path from the child node to the parent node, so the time complexity is O(k).

4. **Solution:**

Python3 code:

```
def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda x: x[0])
        merged = []
        for interval in intervals:
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
                merged[-1][1] = max(merged[-1][1], interval[1])
```

```
        return merged
```

The time complexity of sorting is O(n log n), where n is the size of the array. Once the array of intervals is sorted, merging takes linear time. So the time complexity of the method is O(n log n).

5. **Solution:**

Algorithm: Binary Search Approach:

Data:

S: The first array
T: The second array
n: The size of the first array
n: The size of the second array
k: The order of the required element.

Result: Returns the kth smallest element in the union of S and T

```
bestl ← 0;
L←max (0, k - n);
R←n;
While L ≤ R do
    mid ← (L+R)/2;
    i←mid;
    j←k-i;
    if j = 0 OR A[i – 1] > B[ j – 1] then
        R←mid - 1;
        bestl←mid;
    else
        L←mid + 1;
    end
end
i←bestl;
j←k- i;
if i ≠0 then
    return min (B[j], A[i – 1] );
else
    returnB[j – 1];
end
```

We use this algorithm to find the smallest *i*, such that the biggest value in range [0..i] from S is bigger than the biggest value in range [0..k-i] from T. As we can see, we're taking i elements from S and i-k elements from T. Therefore, we take total k elements.

For each step we compare the two values we got. If we don't choose to take any elements from T, or the value of S[i-1] is larger than T[j-1] , then we know we reached a valid value of *i* and store it. Also we should also try to find a smaller value of *i*. Therefore, we should make the searching range equal to the left one.

On the other hand, if we should take some elements from S, then the kth smallest value is either T[j] or S[i-1]. The reason is that we know that S[i-1] is bigger than all elements in the range T[0, j-1].So the answer is minimum (S[i-1] , T[j]).

The time complexity of the binary search approach is O(log n), where n is the size of the array.

6.**Solution:**
Let A[1 ...n] denote the given array and denote the order statistic by k. The black-box subroutine on A returns the (n/2) element. If k = n/2 then we get what we want. Else, we scan through A and divide into two groups A1, A2 those elements less than A[n/2] and those greater than A[n/2], respectively. If k < n/2, we find the order statistic for the kth element in A1. If k > n/2, we find the order statistic for the (n/2−k)th element in A2.
Algorithm:

```
SIMPLESELECT(A, k)
    BLACK-BOX(A)
    IF k = n/2 return A[n/2]
    DIVIDE(A) /* returns A1, A2 */
    IF k < n/2
        SIMPLESELECT (A1, k)
    ELSE
        SIMPLESELECT (A2, n/2 - k)
END SIMPLESELECT
```

The time complexity of computing the median using the black-box subroutine is O(n), and the time complexity of dividing the array is O(n). Let T(n) be the time complexity of computing the k-th order statistic using the algorithm described above. Then $T(n) \leq cn + T(n/2) = c(n + n/2 + n/4 + n/8 + ... + T(1)) \leq 2cn = O(n)$