Question 1

a) What are the possible solutions for the problem?

1. Traverse the given list, using a queue to store the elements that are not in a given list but are in its list of numbers. Also, there is an array to store the numbers of the number in the list. Then we use the recursion to handle the list, each time we dequeue one element as index and delete it from the given list. At the same time, we check whether the array that store the numbers of the number in the list at that index become zero or not. If it becomes zero, we add it to the queue again and do the recursion until the queue is empty, then output the result.
2. Traverse the given list and find out the elements that are not in a given list but are in its list of numbers, and delete it from the given list directly, and repeat the progress until there are no elements that are not in a given list but are in its list of numbers.

b) How do you solve this problem?

Input: using two array to store the data, one for the given list, another for the numbers of each number in the given array. For example: list {2, 1, 2, 2, 4}, we have counting array {1, 3, 0, 1, 0}, each number represent the times that the numbers exist in the given list, since the given list has 1 of 1, 3 of 2, 0 of 3, 1 of 4, 0 of 5.

Delete the elements: we first check out the elements that are not in a given list but are in its list of numbers, and store it into a queue. The element can be used as index, we delete the elements in that index in the given list and update the counting array at the same time. If some element become zeros, we push it to the queue and repeat the progress until the queue is empty and return the final result.

c) Why is your solution better than others?

If we use the solution 2, as I given above, the time complexity can be very big since it traverses the list every time it deletes the elements, and may cause index error. My solution is efficient since it uses a queue to store the data and it has a clear stop criterion that it stops when the queue is empty.

given list $A = \{2, 1, 2, 2, 4\}$

$$A\_num\_count = \left\{\begin{array}{ccccc} 1, & 3, & 0, & 1, & 0 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 1 & 2 & 3 & 4 & 5 \end{array}\right\}$$

$\downarrow \quad \downarrow$

queue $= \{3, 5\}$

$\Downarrow$ delete

$A = \{2, 1, 0, 2, 0\}$

3→2 1→0

$A\_num\_count = \{1, 2, 0, 0, 0\}$

$\downarrow$

queue $= \{4\}$

$\Downarrow$ delete

$A = \{2, 1, 0, 0, 0\}$

$\updownarrow \ \updownarrow \ \updownarrow$

$A\_num\_count = \{1, 2, 0, 0, 0\}$ $\Longrightarrow$ Finish.

result = 3.

Question 2

a) What are the possible solutions for the problem?
   1. Using recursion to handle the complexity calculation, passing parameters for the next recursion, when meeting L / 2 numbers of “E” it exits the cycle and return the result.
   2. We have a complexity list first; it is used to store the complexity of each meet’s “F”. For example {1, 0, 1}, it has meets 3 “F”, so it records 3 elements represent each F’s complexity. Using a stack to store the position of each “F” index, when meeting an “E”, we pop an element from the stack as index and update our data set with the current index. For example, the complexity list above {1, 0, 1}, our stack is {0, 1, 2}. We meet our first “E”, so we pop an element from the stack, that is “2”, so we first find out the maximum number in the complexity list after the index 2 to the end, and add it to the complexity list[index], and fill all elements after the index 2 to the end with 0. This means that we have settled the current complexity of “E”. If we meet a new “F”, we push it to the stack. Repeating the progress until the stack is empty, we can get the result.

b) How do you solve this problem?
Input: The inputs are nested in a function that checks the complexity, and they are all be seemed as strings.

Check complexity: according to the question, there are only 1 or 0 complexity for each “F”. When x, y are both numbers or both n, the complexity is 0. we simply check out whether x, y are the same for the situation x and y are both n. When x, y are both numbers, we can check x[0] y[0] is a digit or not using the *isdigit()* function. And we combined the two conditions above we can derive an F’s complexity.

Updates the complexity: as we talk above, we use a stack to store the position of each “F” index, when meeting an “E”, we pop an element from the stack as index and update our data set with the current index. We will eventually get a list like {total complexity, 0, 0, 0, …,0}. The first element is the total complexity of the input.

c) Why is your solution better than others?
My solution can correctly handle the complexity nesting and computation very well, while the solution 1 I mention above cannot handle complexity nesting since it only uses recursion and pass parameters for the next recursion, the complexity is cumulative at the same level. My solution using the stack can update the complexity correctly since it records every F’s index, when meeting a “E” it can settle complexity in a timely manner and won’t cumulate the complexity at the same level.

complexity_list:	Stack (F_index)		Operations
1	0	Push 0	F i 1 n
1 1	0 1	Push 1	F x 1 n
1 $\bigcirc\!\!\!\!1$ | ...... max=0 ↳ 1+0	0	Pop 1	E
1 1 1	0 2	Push 2	F y 1 n
1 $\bigcirc\!\!\!\!1$ | 0 max=0 ↓ 1+0	0	Pop. 2	E
2 | 0 0 max=1 ↓ 1+1	empty	Pop 0	E

result = complexity_lish[0] = 2.

⇓

$O(n^2)$