Assigment3

name: Cai Hesen StuID:A0313271R

Q1

solution:

1.1 The algorithm is divided into two parts:
Include all the cards in the cnadidate list.
While the number of cards in the candidate list is greater than 1:Create a new empty candidate list.Group the cards in the current candidate list into pairs (if the number is odd, keep the last card unpaired).For each pair of cards, use a matching device to check if they are the same.case1:If they are the same, keep one of them in the new candidate list.
case2:If they are different, discard the pair.
If there is an unpaired card, add it to the new candidate list.Update the candidate list to the new candidate list.When the algorithm ends,there are two probable cases.case1:there is one candidiate left(we need to go to the next part)
case2:there is no one left.The algorithm ends and return no subset

In the worst case, each round approximately halves the number of candidates, resulting in $O(\log n)$ rounds.Each round involves at most $O(n)$ matching queries. Therefore, the total number of queries in this phase is $O(n \log n)$.

And we need to verify the candidate,just count how many times the candidate match with the left card.That costs $O(n)$ matching queries.
The total time is $O(n \log n)$ + $O(n)$ = $O(n \log n)$.
1.2 proof of the algorithm's correctness

Initialization:At the beginning,if the correct candidate exists,the candidate is included.
Maintenance
Suppose at the beginning og the loop,the correct candidate **R**is in the list,and group the cards into pairs:
for any pair,there are three cases.case1:the pair has two **R**s,so the **R** will be selected into the new list.case2:the pair has one **R** and another card. So both of cards are discarded,but the **R** is still the major,the situation doesn't change in the next loop.case3:the pair has no **R**,which doesn't have an impart on the **R**'s status.
so after the loop,the **R** is still in the list,and the correctness is ensured.
Termination:

if the Candidate list contains only one card:
According to the loop invariant, if the correct candidate m exists, then it must be that card. At this point, the algorithm proceeds to the verification phase to further confirm whether it meets the required condition

if the Candidate list is empty:
This indicates that no card was retained during the pairwise comparisons. This scenario typically corresponds to the absence of any candidate meeting the required condition

Q2:
solution:

2.1 Algorithm Steps

1. the Recursive Division part
   If $left \geq right$, return 0 and thay is the base case: single element or empty array has no inversions.Compute $mid = \lfloor (left + right)/2 \rfloor$.Recursively count inversions in the left half: $count_{left} = \text{CountInversions}(A, left, mid)$.Recursively count inversions in the right half: $count_{right} = \text{CountInversions}(A, mid + 1, right)$.
2. the Merging and Counting Cross-Inversions part
   Use a merge procedure to count the number of inversions where one element is in the left half and the other in the right half.During merging, if $A[i] > A[j]$, then all elements from $A[i]$ to $A[mid]$ form an inversion with $A[j]$, contributing $mid - i + 1$ to the inversion count.Store the merged result in a temporary array and copy it back to A.

$$count = count_{left} + count_{right} + count_{merge}$$

2.2 Correctness Proof
Base Case: When the array has only one element, there are no inversions, and the function correctly returns 0.

Inductive Step:
The recursive calls correctly count inversions within each half.The merge step correctly counts cross-inversions by leveraging the sorted order.Since we check when $A[i] > A[j]$ and count all remaining elements in the left half, all and only the necessary inversions are counted.Thus, by induction, the algorithm correctly counts all inversions in the array.

2.3 Time Complexity Analysis
The recurrence relation follows the structure of Merge Sort:

$$T(n) = 2T(n/2) + O(n)$$

Using the Master Theorem, we find that:

$$T(n) = O(n \log n)$$

Q3
solution:

We can identify two key invariants—one for the outer loop and one for the inner loop—that together prove the correctness of the sorting process.

3.1 Invariant 1: Outer Loop Invariant

At the start of the i-th iteration of the outer loop (where $i = 1, 2, \ldots, n-1$), the last $i-1$ elements of the array are in their final, sorted positions.After each full pass through the inner loop, the largest unsorted element "bubbles up" to its correct position at the end of the array. Thus, by the time the i-th pass begins, the last $i-1$ positions have been fixed and will not be modified further.

3.2 Invariant 2: Inner Loop Invariant

At the start of the j-th iteration of the inner loop (within a particular outer loop iteration), the elements processed so far (from the beginning of the unsorted portion) are in the correct order relative to each other, and the current unsorted segment is being rearranged so that the largest element moves towards its final position.During each iteration of the inner loop, adjacent elements are compared and swapped if they are in the wrong order. This ensures that by the end of the inner loop, the largest element among the unsorted portion has moved to the end of that segment, preparing it to be fixed in place by the outer loop.

3.3 Proof of Invariants

Initialization:
Outer Loop:
Before any iterations, no element is guaranteed to be in its final position. However, after the first complete pass, the largest element will have bubbled to the last position, so the invariant holds with the last 0 elements (trivially) already correct, and then the process fixes one element.

Inner Loop:
At the start of the inner loop, the unsorted segment is unprocessed. The first element is trivially in order with respect to itself.

Maintenance:
Inner Loop:

Suppose that at the beginning of an iteration, the invariant holds. As the inner loop compares adjacent elements and swaps them if necessary, it gradually moves the largest element of the unsorted portion toward the end. Thus, the invariant is maintained at the start of each subsequent inner loop iteration.

Outer Loop:
Assume that after $i - 1$ iterations, the last $i - 1$ elements are sorted. During the i-th iteration, the inner loop guarantees that the largest element from the unsorted part moves to position $n - (i - 1)$. This action fixes one more element in its final position, thereby preserving the outer loop invariant for the next iteration.

Termination
When the outer loop has completed $n - 1$ iterations, every element except possibly the first one has been placed in its final position. Since the array consists of n elements and the unsorted portion has shrunk to just one element, the entire array is sorted.