Question 1

Show how to sort n integers in the range 0 to $n^6 - 1$ in $O(n)$ time.

We can use radix sorting to handle this problem. The steps are showed as follow:

- We transform each of the n integer in to n radix number.
- We sorted the n integer from the lowest digit to the highest digit by counting sort.

After these two steps, we can get the ordered result. We show the algorithm in Alg 1.0.1.

Algorithm 1.0.1 n-radix sort

Input: n integer $a_1, a_2, ..., a_n$

1: Transform $a_1, a_2, ..., a_n$ into n-radix numbers
2: **for** $i = 1; i <= 6; ++i$ **do**
3: Sort $a_1, a_2, ..., a_n$ by *i-th* least digit using counting sort
4: **end for**
5: Return sorted result

We can find that, the time of each counting sort is $O(n)$, and the total time cost is $O(n)$.

Question 2

Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, ..., a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two ordering are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(nlogn)$ algorithm to count the number of significant inversions between two orderings.

It's similar as the algorithm of finding the number of inversions expect that we need two pointers in the left array, which point to the current element of the left array and the least number that greater than twice of the current element of the right array. Consider the following example show in figure 2.1. There are two arrays, they are array A and array B. Suppose Both of the two arrays have been sorted and counted about the inversion number satisfying $i < j$ and $a_i > a_j$(we call this *twice inversion*), we now need to merge A and B as well as count the twice inversion between A and B.

Figure 2.1 Two subarrays

As is showed in figure 2.2, we construct merge array which is initialized as null. And we set three pointers i, j, k. Pointer i and pointer j point the current element in array A and array B, respectively. Pointer k points to the element in A which is the least number greater the the current element in array B.

We compare each current element in A and B. If $A[i] < B[j]$, then we add $A[i]$ to the merge array, and move forward pointer i by 1 step. Besides we need to keep pointer k not behind pointer i. If $A[i] > B[j]$, we need to find the least element $A[k]$ in A, which is greater than double of $A[j]$. We set the current twice inversion number rn as $len(A) - k + 1$, and add the current twice inversion number to the result. Figure 2.3 shows one state of the middle progress. And Figure 2.4 shows the final state.

$rn = 6$

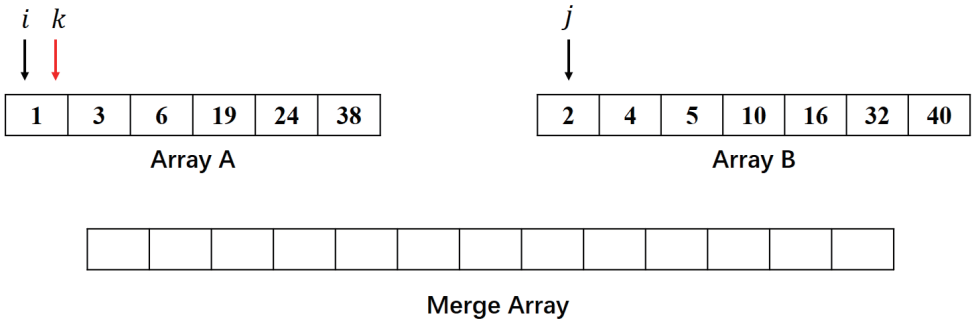

Figure 2.2 Two subarrays

$rn = 4$

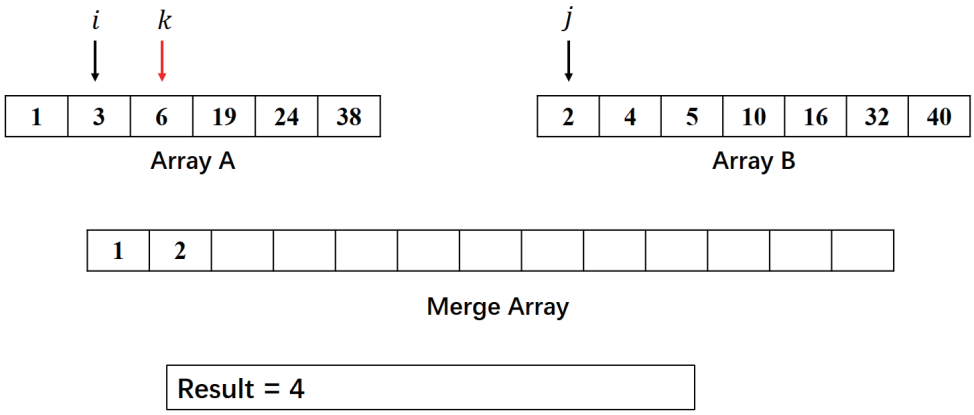

Figure 2.3 Two subarrays

$rn = 0$

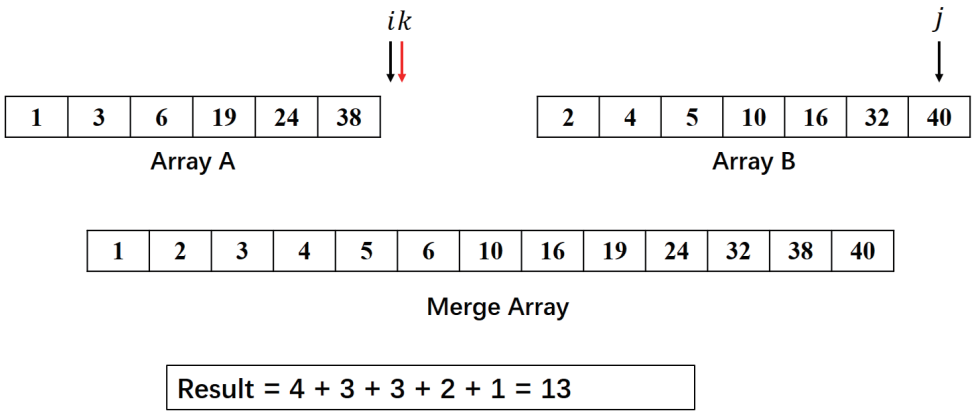

Figure 2.4 Two subarrays

Next, we give the merge part algorithm in Alg 2.0.2.

Algorithm 2.0.2 Count twice inversion number

Input: Array A and B

```
1: Result = 0
2: MergeArray = ϕ
3: while i < length(A) and j < lengthB do
4:   if A[i] < B[j] then
5:     Add A[i] to MergeArray
6:     i + +
7:     if k < i then
8:       k = i
9:       rn − −
10:    end if
11:  else
12:    while k < length(A) and A[k] <= 2B[j] do
13:      k + +
14:      rn − −
15:    end while
16:    Add B[j] to merge array
17:    Add rn to the result
18:    j++
19:  end if
20: end while
21: Add the remain elements in A or B to merge result
22: Return sorted result
```

Question 3

Suppose your're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(nlogn)$ invocations of the equivalence tester.

We find that if there is a set of more than $n/2$ of cards that are all equivalent to one another, the mid-number of the n cards must in this set. So the method is easy. We first sort these n cards and select $\lfloor n/2 \rfloor + 1$-*th* one. Then we compare this element with all the others to test whether there are $\lfloor n/2 \rfloor$ elements equal to it. If so, we say there exists this set, else there not exists this set. We know the time of sorting is $O(nlogn)$ and the test time is $O(n)$. So the total time is $O(nlogn)$.

Actually, we can decide the answer with only $O(n)$ by using *Linear time selection algorithm*. That is each time we use part of the quick sort algorithm to find the position of the current element. If the position is less than the middle position, we recursive select the left part, else, we recursive select the right part. The time cost is $O(n)$.

Question 4

Suppose now that you're given $n \times n$ grid graph G.(An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i-k|+|j-l| = 1$.)

We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G.(Note that G has n^2 nodes.)

We first give a example. Consider the grid graph show in figure 4.1.

1	7	5	8	6	1	2
3	8	5	9	6	10	7
1	2	1	7	8	6	2
3	4	7	9	7	8	7
1	10	4	10	6	4	6
2	1	2	10	5	7	10
8	9	8	9	9	9	9

Figure 4.1 A grid graph example

We can use divide-and-conquer to handle this problem. Intuitively, we can find the minimum element *tmin* in the middle column and test whether both the left and right element are greater than *tmin*. If so, we can return *tmin* directly, else we choose the one that less than *tmin* and then recursive choose the minimum element.

Considering that we should declare the specific area, we can add a board to the grid graph G, and keep the elements on the boarders are always greater than the current element. We show the steps as follows.

1. First, we add four boarders to G as is showed in figure 4.2. And fill the border with the value greater than all value in G;
2. We calculate the minimum value of elements on the borders and on the middle row and middle column. And we get the current minimum value is 3;

3. We test the untested neighbors and find the element 1 above it is less than it. So we choose this element as get into the sub-window at the up-left corner.
4. We recursively calculate the minimum value of elements on the borders and on the middle row and middle column. And we get element 2. And then we can choose the element 1 on the left on element 2 as the local minimum of G.

11	11	11	11	11	11	11	11	11
11	1	7	5	8	6	1	2	11
11	3	8	5	9	6	10	7	11
11	1	2	1	7	8	6	2	11
11	3	4	7	9	7	8	7	11
11	1	10	4	10	6	4	6	11
11	2	1	2	10	5	7	10	11
11	8	9	8	9	9	9	9	11
11	11	11	11	11	11	11	11	11

Figure 4.2 A grid graph example

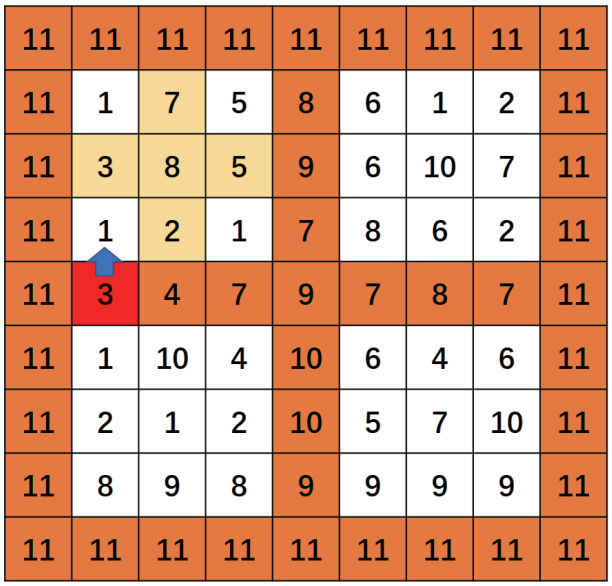

Figure 4.3 A grid graph example

We give the algorithm in Alg 4.0.3. We can find that calculating the minimum value in each recursive step costs $O(n)$. And we have $T(n) = T(\frac{n}{4}) + O(n)$. Therefore the complexity of running time is $O(n)$.

Algorithm 4.0.3 Find local minimum value

Input: Grid M

1: Add four borders to M and fill it with the value greater than the maximum value in M.
2: Set the current windows size S as Size(M)+2
3: **while** $S > 0$ **do**
4: Find the minimum element *elem* in current four borders, middle row and middle column
5: **if** *elem* is the local minimum **then**
6: Return *elem*
7: **else**
8: Choose the windows which contains the neighbor *elemNB* of *elme* satisfying $elemNB < elem$
9: Set S as $\lfloor S/2 \rfloor$
10: **end if**
11: **end while**