Assignment 3

5-1

Assume that A and B are the two databases. A(i), B(i) are i^{th} smallest value in A and B.
Let k = $\lceil n/2 \rceil$, then A(k) is the median value of A and B(k) is the value median of B.
Suppose A(k) < B(k), then B(k) is greater than the first k elements of A. So B(k) is at least $2k^{th}$ small value. Since $2\text{k} \geq \text{n}$, the elements in B greater than B(k) can not be the median value of A and B. Thus, We can delete these $\lfloor n/2 \rfloor$ values in B and get a new database B'. Similarly, the first $\lfloor n/2 \rfloor$ elements in A are smaller than A(k) and can not be the median value of A and B. Also, we can delete these $\lfloor n/2 \rfloor$ values in A get a new database A'. Since the amount of values deleted in A and B are equal, the median value of A and B is the same as the median value of A' and B'.
Repeat above steps to A' and B' until there is only one value in each database. The smaller one in these databases is the n^{th} smallest value in A and B.

Algorithm:

```
Median(A, B, n) {
    If (n = 1) return min(a, b)    // A has only one value a and B has only one value b.
    Let k = $\lceil n/2 \rceil$
    If (A(k) < B(k)){
        Let A' = {A($\lfloor n/2 \rfloor + 1$), A($\lfloor n/2 \rfloor + 2$), …, A(n)}, B' = {B(1), B(2), …, B($\lfloor n/2 \rfloor$)}
        return Median(A', B', $\lfloor n/2 \rfloor$)
    } else {
        Let A' = {A(1), A(2), …, A($\lfloor n/2 \rfloor$)}, B' = {B($\lfloor n/2 \rfloor + 1$), B($\lfloor n/2 \rfloor + 2$), …, B(n)}
        return Median(A', B', $\lfloor n/2 \rfloor$)
    }
}
```

Let Q(n) be the number of queries asked by the algorithm to obtain the answer. Thus, Q(n) = Q($\lceil n/2 \rceil$) + 2. So Q(n) = $2\lceil \log n \rceil$.

5-2

The algorithm is very similar to that of counting inversions. The only change is that here we separate the counting of significant inversions from the merge-sort process.
Algorithm:

```
Let A = ($a_1, a_2, \dots, a_n$).
CountInv(A) {
    If (n = 1) return 0;
    Let L = {A[1], A[2], … , A[n/2]}, R = {A[n/2+1], A[n/2+2], … , A[n]}
    Let B, x = CountInv(L) // B is the sorted L
    Let C, y = CountInv(R) // C is the sorted R
    Let i = 1, j = 1, z = 0
    While (i <= length(B) and j <= length(C)) {
        If (B[i] > 2*C[j]) z += length(B)-i+1; j += 1;
        else i += 1;
    }
    // Let D[1...n] be an array of length n, and every entry is initialized with 0;
    For(k = 1 to n) {
        if(B[i] < C[j]) D[k] = B[i]; i += 1;
        else D[k] = C[j]; j += 1;
    }
    return D, (x + y + z);
}
```
Hence, the recurrence relation is T(n) = 2T(n/2) + O(n). So the time complexity is O(nlog n).

5-3

Divide the cards into two parts of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Assume that on each of the smaller lists, we have a function F such that when run on either of these two smaller lists, F returns a pair (m, c) such that account number m occurs c times in the list and c is greater than half the list, and if no such m exists, then F returns (−1, 0).
Suppose F returns (m1, c1) from the first list and (m2, c2) from the second list. If m1 = m2, then we return (m1, c1 + c2). Otherwise, we search list 2 for m1 updating c1 as we go and likewise search list 1 for m2 updating c2 as we go. If c1 > c2 we return (m1, c1), if c2 > c1, we return (m2, c2), and if c1 = c2, we return (−1, 0). If we have returned something with positive c, then a set of more than n/2 cards is equivalent to each other, and otherwise there is no such set.
The two paragraphs above describe a recursive procedure for computing F. Following the first paragraph, F needs base cases for an empty list (return (−1, 0)), a list containing a single item a (return (a, 1)), and a list containing two items, a, b (return (a, 2) if a = b, and (−1, 0) otherwise). If the list is larger than 2, then divide it into two pieces and run F on each of the smaller lists. After F returns from each of these runs, as described in the second paragraph combine them and return the appropriate pair.
The operations in the second paragraph require $\lfloor n/2 \rfloor + \lceil n/2 \rceil$ = n iterations. The cost of running F is therefore given by the recurrence T(0) = 0, T(1) = 0, T(2) = 2, T(n) = 2T(n/2) + n for n > 2.
Similar to merge sort, this has solution in Θ(n log n).

5-4

Define one vector to be A = $(q_1, q_2, \dots, q_n)$ and the other vector to be B = $\left(-\frac{1}{(n-1)^2}, -\frac{1}{(n-2)^2}, \dots, -\frac{1}{2^2}, -\frac{1}{1^2}, 0, \frac{1}{1^2}, \frac{1}{2^2}, \dots, \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2}\right)$.

Calculate the convolution of these two vectors. The convolution will contain all the items in $\sum_{i<j} \frac{q_i}{(j-i)^2} - \sum_{i>j} \frac{q_i}{(j-i)^2}$. If we multiply it by Cq_j, we can get F_j. Because the convolution can be computed with a FFT in $O(n\log n)$ time, the algorithm is $O(n\log n)$.

5-5

Label the lines in order of increasing slope so we get a list S = $\{L_1, L_2, \dots, L_n\}$.
```
Merge(A, B) {
  Let M sets of lines, p = size(A), q = size(B)
```

```
    $A_1$ and $B_q$ must be visible, we add them to M
    Calculate the intersected points (where $A_k$ meets $A_{k+1}$or $B_k$ meets $B_{k+1}$) in A and B then we get
a set of points $\{a_1, a_2, \dots, a_{p-1}, b_1, b_2, \dots, b_{q-1}\}$. Then we sort them in order of increasing x-
coordinates and get $\{c_1, c_2, \dots, c_{p+q-2}\}$.    // the sorting time is O(n)
    Compare these two sets, the first i points and last q-j points are the same. The $i, j$ satisfy that any
line in $\{a_1, a_2, \dots, a_{i-1}, a_i, b_j, b_{j+1}, \dots, b_q\}$ is still visible
    M = $\{A_1, A_2, \dots, A_{i-1}, A_i, B_j, B_{j+1}, \dots, B_q\}$
    Return M
}
Visible(S) {
    If S is empty Return null
    If S contains one or two lines Return S
    Let A, B sets of lines
    k = $\lceil n/2 \rceil$
    L = $\{L_1, L_2, \dots, L_k\}$
    R = $\{L_{k+1}, L_{k+2}, \dots, L_n\}$
    A = Visible (L)
    B = Visible (R)
    Return Merge (A, B)
}
```
Because the divide process is O(logn) and each sort time is O(n), the algorithm is $O(\text{nlogn})$

5-6

Assume the root node of T is r, its label is x_r

```
curMin = $x_r$ // curMin is the current minimum number
FindMin(r, curMin) {
      Let node $r_1$ be the left node of r and $r_2$ be the right node of r
      If either $r_1$ or $r_2$ is null
            return curMin
      If ($x_{r_1} > curMin$ && $x_{r_2} > curMin$) return curMin
      Elseif ($x_{r_1} < x_{r_2}$)
            If ($x_{r_1} < curMin$) curMin = $x_{r_1}$
            return FindMin($r_1$, curMin)
      Else
            If ($x_{r_2} < curMin$) curMin = $x_{r_2}$
            return FindMin($r_2$, curMin)
}
```
This Algorithm can help us find a local minimum of T.

5-7

Consider a sub graph of the G, suppose it is an rectangular grid graph, and v is the smallest grid of the grids on the border of this rectangle. Suppose v' is the grid adjacent to v and inside the rectangle, if v' is smaller than v, then we can conclude that there exist a local minimum inside the triangle. This is because if we can always a "decreasing path" starting from v', ending at local minimum. This decreasing path will not go across the border of the rectangle, since v' is smaller than all the grids on the border. So, base on this conclusion, we can divide and conquer the problem.

Assume that the outer of the G has a very large label. At the first step we probe all the grids of the column at the very middle of G. The column divides the graph G into two n*(n/2) rectangular sub graphs. Suppose the smallest grid of the column is v_1, then we probe the two neighbors of v_1 which are not in the same column of v_1. If the two grids are all larger than v_1, then v_1 is a local minimum. Otherwise, at least one of the two neigbhbors is smaller than v_1, so we can determine which rectangular subgraph definitely contains a local minimum, ccording the conclusion we just worked out. And at the second step, we probe all the grids of the row at the very middle of the rectangular sub graph, this row divides the sub graph into two (n/2)*(n/2) square sub graphs. Suppose the smallest grid of the row is v_2, if $v_2>v_1$, then the decreasing path starting at v_1 will not go across the row, so we choose the square subgraph which contains v_1. (At this situation v_1 will not locate at the common corner of the two squares, since v_1's neighbor is smaller than it and v_2 is larger than it.) If $v_2<v_1$, then we probe the two neighbor's of v_2 which are not at the same row of v_2, and use the same way as we due with v_1, to select one square sub graph, or assert v_2 itself is a local minimum. (At this situation v_2 will not locate at the common corner of the two squares, since v_1 is the smallest grid of the column and $v_2<v_1$.) Now we have shrunk the graph in to a (n/2)*(n/2) subgraph, with 3n/2+4 probes at most. We can apply the same process to the sub graph and finally we will get a local minimum. So the total number of probes will be T(n)=T(n/2)+3n/2+4=3n+4log(n)=O(n).