Name: _____

Entry number: _____

There are 5 questions for a total of 75 points.

---

1. You are given n points $X \subseteq \mathbb{R}^d$ (i.e., n points in d-dimensional Euclidean space) and a positive integer k. You are asked to design an algorithm to find a subset $C \subseteq X$ of size k such that the following quantity is minimized:

$$\max_{x \in X} \{\min_{c \in C} \{||x - c||\}\}$$

(*That is, a subset C of X containing k points such that the maximum distance of a point in X to the nearest point in the set C is minimized.*)

Consider the following greedy algorithm for this problem.

`GreedySelect`(X, k)
- Let c_1 be an arbitrary point in X
- $C \leftarrow \{c_1\}$
- for $i = 2$ to k
  - Pick a point $c_i \in X$ that is farthest from points in C.
    (*That is,* $c_i = \arg\max_{x \in X} \{\min_{c \in C} \{||x - c||\}\}$)
  - $C \leftarrow C \cup \{c_i\}$
- return(C)

(a) (5 points) Show that the above greedy algorithm does not always produce an optimal solution.

Solution: Consider a simple 1-Dimensional input where $n = 3$ and $k = 1$. The three points are $-1, 0, 1$. The optimal solution is $C = \{0\}$. On the other hand, since the algorithm picks the first center arbitrarily, it can pick the center 1 in which case the solution is not optimal.

(b) (10 points) Show that `GreedySelect` gives an approximation guarantee of 2.

Solution: Let C be the set of centers produced by `GreedySelect`(X, k) and let C^* denote some optimal solution. Let $y = \arg\max_{x \in X} \min_{c \in C} ||x - c||$. So, y is the point that is farthest from the set of centers C and let $R = \min_{c \in C} ||y - c||$ be this farthest distance. Consider the set $C' = C \cup \{y\}$. Note that due to the manner in which the centers in the set C are picked, each pair of points in the set C' have distance at least R. Now, at least two points in the set C' share the same nearest center in the set C^*. Let these points be p and q and the center be c^*. We have $||p - c^*|| + ||q - c^*|| \geq ||p - q|| \geq R$. So, we have $\max(||p - c^*||, ||q - c^*||) \geq R/2$ which further gives

$$\max_{x \in X} \{\min_{c \in C^*} \{||x - c||\}\} \geq \frac{1}{2} \cdot \max_{x \in X} \{\min_{c \in C} \{||x - c||\}\}.$$

2. (15 points) You are given n points on a two-dimensional plane. A point (x, y) is said to *dominate* (x', y') iff $x > x'$ and $y > y'$. Design an algorithm to output all points that are not dominated by any other point. Give pseudocode, discuss running time, and give proof of correctness.

Solution: Here is a divide and conquer algorithm for this problem. The point set is given as an array P. We will sort the array P based on the X-coordinate before giving this as an input to the recursive program below. Pre-sorting the points will take $O(n \log n)$ time.

`NotDominated(`P`)`

- $n \leftarrow |P|$
- If $(n = 1)$return$(\{P[1]\})$
- $P_L \leftarrow P[1], P[2], ..., P[\lfloor n/2 \rfloor]$
- $P_R \leftarrow P[\lfloor n/2 \rfloor + 1], P[2], ..., P[|P|]$
- $S_L \leftarrow$ `NotDominated(`P_L`)`
- $S_R \leftarrow$ `NotDominated(`P_R`)`
- Let y denote the largest Y-coordinate of points in S_R
- Let S'_L be the points in S_L that have Y-coordinate $<$ y
- return$(S_L \cup S_R - S'_L)$

Let T denote all the points in P that are not dominated.

Claim 1: If $P = \{p\}$, then $T = \{p\}$.

Proof. When P contains only one point then that point is trivially not dominated. □

Let S_L be the points in P_L that are not dominated, S_R be the points in P_R that are not dominated. Let y be the Y coordinate of the point in S_R that has the largest Y-coordinate. Let S'_L be all the points in S_L that have Y-coordinates $\leq y$.

Claim 2: $T = S_L \cup S_R - S'_L$

Proof. We will prove (i) $T \subseteq S_L \cup S_R - S'_L$ and (ii) $S_L \cup S_R - S'_L \subseteq T$. We start by giving a proof of (i). Let p be any point in T. If $p \in P_L$, then $p \in S_L$ and $p \in S_L - S'_L$. If $p \in P_R$, then $p \in S_R$. So, $p \in S_L \cup S_R - S'_L$.

Now, consider any point $p \in S_R \cup (S_L - S'_L)$. If $p \in S_R$, then $p \in T$ since p is not dominated by any point in P_L. If $p \in S_L - S'_L$, then again $p \in T$ since the Y-coordinate of p is $\geq$ the Y-coordinate of any point in P_R and hence is not dominated by any point in P_R. □

The proof of correctness of our divide and conquer algorithm follows from the above claims.

Running time: The time to sort the points in increasing order of X-coordinates is $O(n \log n)$. Let $T(n)$ denote the worst case running time of `NotDominated`. Then we have $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$ and $T(1) = O(1)$. The solution of this recurrence relation is $T(n) = O(n \log n)$. So, the total running time is $O(n \log n)$.

3. (15 points) Let S and T be sorted arrays each containing n elements. Give an algorithm to find the n^{th} smallest element out of the $2n$ elements in S and T . Give pseudocode, discuss running time, and give proof of correctness.

Solution: The following algorithm outputs the n^{th} smallest element.

`NthElement`(S, T, n)
- If $(n = 1)$return$(\min(S[1], T[1]))$
- $i \leftarrow \lfloor n/2 \rfloor$; $j \leftarrow \lceil n/2 \rceil$
- If $(S[i] = T[j])$return$(T[j])$
- If $(S[i] < T[j])$
  - $A \leftarrow S[i+1]...S[n]$; $B \leftarrow T[1]...T[j-1]$
  - return(`NthElement`$(A, B, n - \lfloor n/2 \rfloor)$)
- Else
  - $A \leftarrow S[1]...S[i-1]$; $B \leftarrow T[j+1]...T[n]$
  - return(`NthElement`$(A, B, n - \lfloor n/2 \rfloor)$)

When $n = 1$, then the smallest element in arrays S and T is just $\min(S[1], T[1])$. Let $i = \lfloor n/2 \rfloor$ and $j = \lceil n/2 \rceil$. If $S[i] = T[j]$, then $T[j]$ is the n^{th} smallest number since there are $(i + j - 1) = n - 1$ numbers that are $\leq T[j]$ when S and T are combined. If $S[i] < T[j]$, then for any $1 \leq k \leq i$ $S[k]$ cannot be the n^{th} smallest number since the number of integers that are larger than $S[k]$ is at least $n - k + n - j + 1 \geq n - i + n - j + 1 = n + 1$. Similarly, for any $j \leq k \leq n$, $T[k]$ cannot be the n^{th} smallest number. So, when $S[i] < T[j]$, then the n^{th} smallest integer in S and T combined is the same as the $(n - \lfloor n/2 \rfloor)^{th}$ smallest integer in $S[i+1]....S[n]$ and $T[1]....T[j-1]$. Also, when $S[i] > T[j]$, then the n^{th} smallest integer in S and T combined is the same as the $(n - \lfloor n/2 \rfloor)^{th}$ smallest integer in $S[1]....S[i-1]$ and $T[j+1]....T[n]$.

Running time: The recurrence relation for the running time if given by $T(n) \leq T(\lceil n/2 \rceil) + O(1)$ and $T(1) = O(1)$ which solves to $T(n) = O(\log n)$.

4. (20 points) This is problem no. 5 from Chapter 5 in the book. You are given n non-vertical lines in the plane, labeled $L_1, ..., L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost – intuitively some portion of it can be seen if you look down from "$y = \infty$". Design an algorithm that takes n lines as input and returns all the lines that are visible. Give pseudocode, discuss running time, and give proof of correctness.

Solution: We give a divide and conquer algorithm for this problem. We divide the problem based on the slopes of the lines. That is the value a_i's. We sort the lines in increasing order of a_i. Let $L_1, ..., L_n$ be the lines in increasing order of slopes. Sorting costs $O(n \log n)$ times. As a preprocessing step, we also check if there are parallel lines. For any two parallel lines, we remove the line that has a lower Y-coordinate at $x = 0$. This preprocessing costs $O(n)$ time. We then use the following recursive algorithm to find the visible lines. The recursive algorithm, returns the visible "envelope" of all the lines. That is, it outputs a sequence of line "segments" (in increasing order of slopes) $(l_1, ..., l_k)$ that form the visible outline when viewed from $y = \infty$. It also outputs the points $(p_{1,2}, p_{2,3}, ..., p_{k-1,k})$ which are intersection points of line pairs (l_1, l_2), (l_2, l_3), ..., (l_{k-1}, l_k) respectively. See figure 1 for a visual representation of this.

`FindEnvelope(`L`)`

- If ($|L| = 1$)return($(L[1], [])$)
- $L_L \leftarrow L[1], ..., L[\lfloor |L|/2 \rfloor]$
- $L_R \leftarrow L[\lceil |L|/2 \rceil], ..., L[n]$
- $(l_L, p_L) \leftarrow$ `FindEnvelope(`L_L`)`
- $(l_R, p_R) \leftarrow$ `FindEnvelope(`L_R`)`
- return(`MergeEnvelope(`$(l_L, p_L), (l_R, p_R)$`)`)

`MergeEnvelope(`$(l_L, p_L), (l_R, p_R)$`)`

- $n \leftarrow |p_L|$; $m \leftarrow |p_R|$
- If $l_L[n+1]$ intersects with $l_R[j]$ at point s, then
  return($[l_L[1], ..., l_L[n+1], l_R[j], ..., l_R[m+1]], [p_L[1], ..., p_L[n], s, p_R[j], ..., p_R[m]]$)
- If $l_L[1]$ intersects with $l_R[j]$ at point s, then
  return($[l_L[1], l_R[j], ..., l_R[m+1]], [s, p_R[j], ..., p_R[m]]$)
- If $l_R[1]$ intersects with $l_L[i]$ at point s, then
  return($[l_L[1], ..., l_L[i], l_R[1], ..., l_R[m+1]], [p_L[1], ..., p_L[i-1], s, p_R[1], ..., p_R[m]]$)
- If $l_R[m+1]$ intersects with $l_L[i]$ at point s, then
  return($[l_L[1], ..., l_L[i], l_R[m+1]], [p_L[1], ..., p_L[i-1], s]$)
- *// Now we only deal with intersection of the envelopes defined by* $l_L[2], ..., l_L[n]$ *and* $l_R[2], ..., l_R[m]$
- $i \leftarrow 1$; $j \leftarrow 1$
- While ($p_L[i].x \leq p_R[j].x$)$i \leftarrow i + 1$
- While ($j < m$)
  - $i \leftarrow i - 1$
  - While ($p_L[i].x \leq p_R[j+1].x$)
    - If line segment $l_L[i+1]$ intersects with $l_R[j+1]$ at s, then
      return($[l_L[1], ..., l_L[i+1], l_R[j+1], ..., l_R[m+1]], [p_L[1], ..., p_L[i], s, p_R[j+1], ..., p_R[m]]$)
    - $i \leftarrow i + 1$
  - $j \leftarrow j + 1$

The proof of correctness of the recursive algorithm follows from the fact that for the base case, the algorithm returns the correct answer and the merge routine correctly returns the envelope of all the lines. Indeed, since the the slope of all lines in L_L are smaller than the slope of all lines in L_R, the envelope of L_L and L_R can only intersect in one point. This intersection of envelopes defines the envelope of all lines. The merge routine just finds this intersection point of the two envelopes and returns the merged envelope. We explicitly check intersection for segments $l_L[1], l_L[n+1], l_R[1], l_R[m+1]$. After this, we check all candidates for possible intersection of $l_R[2], ..., l_R[m]$. So, we exhaustively check all possibilities for intersection. Figure 2 and 3 shows an example of merging.

Running time: The sorting of lines and dealing with parallel lines are done as a preprocessing step and from the previous discussion, we know that the running time bound for pre-processing

is $O(n \log n)$. The recurrence relation for the running time of the recursive algorithm is given by $T(n) \leq 2 \cdot T(n/2) + O(n)$ which solves to $T(n) = O(n \log n)$. So the overall running time is $O(n \log n)$.

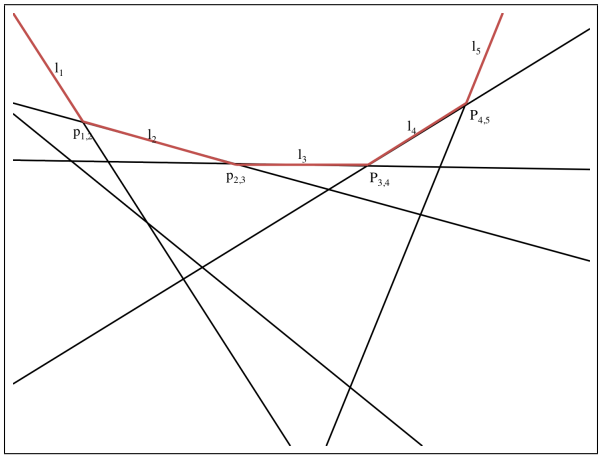

Figure 1: Example of lines and its "envelope"

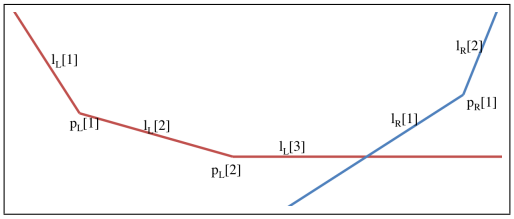

Figure 2: Envelopes of left and right

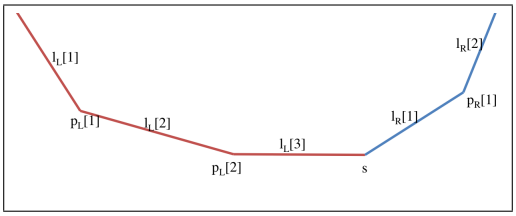

Figure 3: Merged envelope

5. (10 points) You are given an unsorted integer array A containing n distinct integers (assume n is a power of 2). You are supposed to design an algorithm that finds the minimum and the second minimum element in the array A. The running time for this problem is measured in terms of the number of pairwise comparisons that your algorithm performs in the worst case. (*Note that you can find the minimum and the second minimum using* $(2n-3)$ *comparisons by a linear scan.*) Design an algorithm that solves the problem using at most $(n + \log n - 2)$ comparisons. Give pseudocode. Discuss correctness and argue why the number of comparisons is at most $(n + \log n - 2)$.

Solution: Consider a tournament between these n given integers in the array A. In round-1, matches between $A[i]$ and $A[i+1]$ is held for all odd i. For each of these matches, the winner is the one with smaller value. The winners proceed to play in the second round. Again, there are matches in pairs of two and similar winning rule applies. This forms a binary tree. Note that the winner is at the root of the binary tree and because of our winning rule, this is the minimum element of the array. The total number of comparisons required is equal to the total number of matches played in this tournament which is equal to $n/2+n/2^2+...+1 = \frac{n}{2}\cdot(1+1/2+1/2^2+...+1/2^{\log n-1}) = (n-1)$. In order to find the second minimum, we just need to **trace** the path of the minimum element in the tournament (this is because the second minimum element can only be beaten by the minimum element) and consider the opponents of the minimum elements. Out of these opponents, we need to pick the minimum element. This will cost $\log n - 1$ comparisons since there are $\log n - 1$ opponents of the winner in the tournament. So, the total number of comparisons is $n + \log n - 2$. Here is the pseudocode the implements the above idea. We are making the assumption that n is a power of 2.

```
MinSecMin(A, n)
 - k ← log n
 - for i = 2^k to 2^{k+1} − 1
     - B[i] ← A[i − 2^k + 1]
 - for i = k − 1 to 0
     - for j = 2^i to 2^{i+1} − 1
         - B[j] ← min (A[2j], A[2j + 1])
 - min ← B[1]
 - if(min = B[2]){secMin = B[3]; j ← 2 }
 - else {secMin = B[2]; j ← 3}
 - while(j < 2^k)
     - if(min = B[2j])
         - if(secMin > B[2j + 1])secMin ← B[2j + 1]
         - j ← 2j
     -else
         - if(secMin > B[2j])secMin ← B[2j]
         - j ← 2j + 1
 - return((min, secMin))
```