COMP 3711 – Design and Analysis of Algorithms
2019 Spring Semester – Written Assignment # 1
Distributed: Feb 20 2019 – Due: March 1, 2019

Your solutions should contain (i) your name, (ii) your student ID #, and (iii) your email address
Notes:

- Please write clearly and briefly.
- Please follow the guidelines on doing your own work and avoiding plagiarism given on the class home page.
  In particular ***don't forget to acknowledge individuals who assisted you, or sources where you found solutions.*** Failure to do so will be considered plagiarism.
- Also follow the submission guidelines on the class web page. Use white, unwatermarked paper, e.g., no student society stationary.
  - If handwritten, solutions should be single sided, start a new page for every problem, be single column and leave space between consecutive lines and more space between paragraphs.
  - If typed, try to use an equation editor, e.g., latex , the equation editor in MS-WORD or whatever the equivalent is in whatever typesetting system you are using
- This assignment is due by 13:00 on March 1, 2019 in BOTH hard AND soft copy formats. A hard copy should be deposited in one of the two COMP3711 assignment collection boxes outside of room 4210. A soft copy for our records in PDF format should also be submitted via the online CASS system. See the Assignment 1 page in Canvas for information on how to submit online.
- The default base for logarithms will be 2, i.e., $\log n$ will mean $\log_2 n$. If another base is intended, it will be explicitly stated, e.g., $\log_3 n$.

Problem 1 [21 pts] For each pair of expressions (A, B) below, indicate whether A is O, Ω, or Θ of B. Note that zero, one, or more of these relations may hold for a given pair. List all correct ones. It often happens that some students will get the directions wrong, so please write out the relation in full, i.e., $A = O(B), A = \Omega(B)$, or $A = \Theta(B)$ and not just $O(B)$, $\Omega(B)$ or $\Theta(B)$, omitting the A.

(a) $A = n^4 - n^3, B = 1000n^3 + 100n^2 + 10n$;

(b) $A = \log_9 4n, B = \log_{4.7} n^3$;

(c) $A = 50n \log n \log\log n, B = 2^n$;

(d) $A = (\sqrt{2})^{\log n}, B = \sqrt{\log n}$;

(e) $A = 2^{3\log_3 n}, B = 47n^3$;

(f) $A = 2^{\log_2 n^5}, B = 6n^5$;

(g) $A = \ln\log n, B = 10^{100}$
B, which is 1 followed by 100 zeros, is called a "googol".

Solution:

(a) $A = \Omega(B)$;

(b) $A = O(B), A = \Omega(B), A = \Theta(B)$;

(c) $A = O(B)$;

(d) $A = \Omega(B)$;

(e) $A = O(B)$.
Note that $A = 2^{3\log_3 n} = (2^3)^{\log_3 n} = n^{\log_3 8} = O(47n^3)$.

(f) $A = O(B), A = \Omega(B), A = \Theta(B)$;
Note that $A = 2^{\log_2 n^5} = (n^5)^{\log_2 2} = n^5$.

(g) $A = \Omega(B)$.
This is because B is a a constant value while A is an increasing (albeit slowly) function.

Problem 2 [25 pts] Give asymptotic upper bounds for $T(n)$ satisfying the following recurrences. Make your bounds as tight as possible. A correct answer will gain full credits. It is not necessary to show your work BUT, if your answer was wrong, showing your work steps may gain you partial credits. If showing your work, you may use theorems shown in class or can prove results from scratch.
For (a), (b) you may assume that n is a power of 5/3; for (c), (d), (e) you may assume that it is a power of 3.

(a) $T(1) = 1; T(n) = T(3n/5) + 1$ for $n > 1$.

(b) $T(1) = 1; T(n) = T(3n/5) + n$ for $n > 1$.

(c) $T(1) = 1; T(n) = 9T(n/3) + n^2$ for $n > 1$.

(d) $T(1) = 1$; $T(n) = 7T(n/3) + n^2$ for $n > 1$.

(e) $T(1) = 1$; $T(n) = 5T(n/3) + n$ for $n > 1$.

Solution:

(a) $T(n) = O(\log n)$. *Master Theorem Case 2.*

(b) $T(n) = O(n)$. *Master Theorem Case 3.*

(c) $T(n) = O(n^2 \log n)$. *Master Theorem Case 2.*

(d) $T(n) = O(n^2)$ *Master Theorem Case 3.*

(e) $T(n) = O(n^{\log_3 5})$. *Master Theorem Case 1.*

A small number of students gave (almost) EXACT solutions and not asymptotic upper bounds. They had a few points deducted because they did not anwer the question asked.

Problem 3: [24 pts] Give an asymptotic upper bound for $T(n)$ satisfying the following recurrences. Make your bound as tight as possible. You must prove this bound from scratch (either using the expansion method or the tree method). You may assume that n is a power of 2.

(a)

$$T(1) = 1; \quad T(n) = 4T(n/2) + n^2 \text{ for } n > 1$$

(b)

$$T(1) = 2; \quad T(n) = 5T(n/2) + n^2 \text{ for } n > 1$$

Solution (a):

$T(n)$

$= 4T(n/2) + n^2$, *for* $n > 1$

$= 4[4T(n/2^2) + (\frac{n}{2})^2] + n^2$

$= 4^2T(n/2^2) + (\frac{4}{2^2} + 1) \cdot n^2$

$= 4^2[4T(n/2^3) + (\frac{n}{2^2})^2] + (\frac{4}{4} + 1) \cdot n^2$

$= 4^3T(n/2^3) + (\frac{4^2}{4^2} + \frac{4}{4} + 1) \cdot n^2$

...

$= 4^iT(n/2^i) + [(4/4)^{i-1} + (4/4)^{i-2} + ... + 1] \cdot n^2$

$= 4^hT(n/2^h) + h \cdot n^2$

$= 4^{\log_2 n}T(1) + \log_2 n \cdot n^2$, *by setting* $h = \log_2 n$

$* = n^2 + n^2 \log_2 n$, *given T(1) = 1*

$= O(n^2 \log_2 n)$

where (*) comes from the fact that

$$4^{\log_2 n} = 4^{\frac{\log_4 n}{\log_4 2}} = 4^{\log_4 n \cdot \log_2 4} = (4^{\log_4 n})^{\log_2 4} = n^{\log_2 4} = n^2$$

Solution (b):

$T(n)$

$= 5T(n/2) + n^2$, *for* $n > 1$

$= 5[5T(n/2^2) + (\frac{n}{2})^2] + n^2$

$= 5^2T(n/2^2) + (\frac{5}{2^2} + 1) \cdot n^2$

$= 5^2[5T(n/2^3) + (\frac{n}{2^2})^2] + (\frac{5}{4} + 1) \cdot n^2$

$= 5^3T(n/2^3) + (\frac{5^2}{4^2} + \frac{5}{4} + 1) \cdot n^2$

...

$= 5^iT(n/2^i) + [(5/4)^{i-1} + (5/4)^{i-2} + ... + 1] \cdot n^2$

$= 5^hT(n/2^h) + [\frac{(5/4)^h - 1}{5/4 - 1}] \cdot n^2$

$\leq 5^{\log_2 n}T(1) + \frac{5^{\log_2 n}}{4^{\log_2 n}} \cdot 4n^2$, *by setting* $h = \log_2 n$

$* = 2n^{log_2 5} + 4n^{log_2 5}$, *given T(1) = 2*

$= O(n^{log_2 5})$

where (*) comes from the fact that

$$5^{\log_2 n} = 5^{\frac{\log_5 n}{\log_5 2}} = 5^{\log_5 n \cdot \log_2 5} = (5^{\log_5 n})^{\log_2 5} = n^{\log_2 5}$$

$$4^{\log_2 n} = 4^{\frac{\log_4 n}{\log_4 2}} = 4^{\log_4 n \cdot \log_2 4} = (4^{\log_4 n})^{\log_2 4} = n^{\log_2 4} = n^2$$

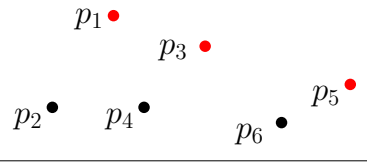
Problem 4 [30 pts]

In two dimensions, point $p = (x, y)$ is *larger* than point $p' = (x', y')$ if

$$p'.x < p.x \quad \text{and} \quad p'.y < p.y.$$

Let $S = \{p_1, \ldots, p_n\}$ be a list of n two-dimensional points $p_i = (x_i, y_i)$. Point $p_j \in S$ is a *largest* point in S if no other point in S is larger than p_j. $L(S)$ is the set of largest points in S, i.e.,

$$L(S) = \{p_j \ : \text{for all } i \neq j,\ p_i \text{ is not larger than } p_j\}\,.$$



p_1 is larger than p_2.
p_1 and p_3 are incomparable;
neither is larger than the other.
$L(S) = \{p_1, p_3, p_5\}$.

The example in the box illustrates the definitions.

In this problem you should design an $O(n \log n)$ divide-and-conquer algorithm for finding $L(S)$, given input S. Do this following the four steps on the next page. Each step should be labelled and answered separately.

1. Let $S^1 = \left\{p_1^1, p_2^1, \ldots, p_{n_1}^1\right\}$ and $S^2 = \left\{p_1^2, p_2^2, \ldots, p_{n_2}^2\right\}$ be two sets of points such that for both $i = 1, 2$, S^i has the following properties:
   (a) $L\left(S^i\right) = S^i$, i.e., every point in S^i is a largest point in S^i.
   (b) Each S^i is sorted by x coordinate, i.e., $p_1^i.x \leq p_2^i.x \leq \cdots \leq p_{n_i}^i.x$.

   Describe an $O(n_1 + n_2)$ time procedure $Merge(S^1, S^2)$ that returns $L\left(S^1 \cup S^2\right)$, the set of largest points in the union of the two sets. The returned set should be sorted by x-coordinate.
2. Explain why your procedure $Merge(S^1, S^2)$ is correct and why it runs in $O(n_1 + n_2)$ time.
3. Using $Merge(S^1, S^2)$ as a subroutine, describe a $O(n \log n)$ time Divide-and-Conquer algorithm $FINDL(S)$ that returns $L(S)$. The returned set should be sorted by x-coordinate.
4. Explain why your procedure $FINDL(S)$ is correct and why it runs in $O(nlogn)$ time.

By convention, a set of points will be stored in an array S where $S[i] = p_i$ and $size(S) = n_i$. You may assume that arrays can be created in constant time and can be passed by reference by subroutines. Your procedures should return an array containing the result points, sorted by x coordinate.

Continued on next page....

In parts 1 and 3 the description should be in clearly documented pseudocode.

Finally, for simplicity, you may assume that no two points share the same x-coordinate or the same y-coordinate.

Solution:

This is a very well known problem that goes by many names: "skylines" in databases, "maximal points" in computational geometry and "pareto optimas" in economics.

The phrase "p is larger than q" is often expressed as "p dominates q" and, notationally, as $q << p$. Note that this relation is transitive, i.e., if $q << p$ and $p << r$ then $q << r$. The points in $L(S)$ are called the "maxima" of S.

1. The algorithm below works in two phases.

   Phase 1 merges S^1 and S^2 by x-coordinate using the the exact same code as Mergesort. The resulting size $n_1 + n_2$ point list is stored in $\bar{S}$.

   Phase 2 scans $\bar{S}$ from right to left.

   At step k it checks if $S[k]$ is in $L(S^1 \cup S^2)$ and, if it is, stores it.

   It maintains value t such that, after step k, the points stored in $\bar{S}[t \ldots n_1 + n_2]$ are the maxima of $S^1 \cup S^2$ seen so far.

   Merge(S^1, S^2)

   0. Let $\bar{S}$ be an array of size $n_1 + n_2$.

   % Phase 1: Sort S^1, S^2 by increasing x-coordinate. Store in $\bar{S}$
   1. $i = 1$; $p^1_{n_1+1} = (\infty, \infty)$;
   2. $j = 1$; $p^2_{n_2+1} = (\infty, \infty)$;
   3. For $k = 1$ to $n_1 + n_2$
   4. If $p^1_i.x < p^2_j.x$ then
   5. $\bar{S}[k] = p^1_i$; $i = i + 1$
   6. Else
   7. $\bar{S}[k] = p^2_j$; $j = j + 1$

   % Phase 2: Max_y stores largest y-coordinate seen so far
   8. $t = n_1 + n_2$; $Max_y = \bar{S}[t]$
   9. For $k = n_1 + n_2 - 1$ downto 1 % Scan points from right to left
   10 If $\bar{S}[k].y > Max_y$ then % If current point maximal, save it
   11 $Max_y = \bar{S}[k].y$
   12 $t = t - 1$
   13 $\bar{S}[t] = \bar{S}[k]$

   % The line below copies the entries $\bar{S}[t], \bar{S}[t+1], \ldots, \bar{S}[n_1 + n_2]$
   % into a new array and returns this new array
   14. Return$(\bar{S}[t \ldots n_1 + n_2])$

   Marking Note: This could easily been written as a one-pass algorithm, combining the two phases into one. The split into two phases is to make it more understandable. For completeness we provide a one-pass equivalent of the next page

   Also note that the use of Max_y is actually superfluous. It is not difficult to prove that $Max_y = S[t].x$. We included Max_y to make the algorithm description clearer.

This is the one-pass method. It is not difficult to see that it is equivalent to the two pass method.

Merge2(S^1, S^2)

0. Let $\bar{S}$ be an array of size $n_1 + n_2$.
% $\bar{S}[t \ldots n_1 + n_2]$ will store current maxima found.
% $\bar{S}[t-1]$ will store new rightmost point that needs to be tested
% If it is maximal, t will be decremented.

% Step 1: Find rightmost point and make it rightmost maxima
1. $t = n_1 + n_2$; $p_0^1 = (-\infty, -\infty)$; $p_0^2 = (-\infty, -\infty)$;
2. If $p_{n_2}^2.x < p_{n_1}^1.x$ then
3. $\quad i = n_1 - 1$; $j = n_2$; $\bar{S}[t] = p_{n_1}^1$
4. Else
3. $\quad i = n_1$; $j = n_2 - 1$ $\bar{S}[t] = p_{n_2}^2$

% Step 2: Scan through points from right to left
% Storing maxima so far and testing if current point is maximal
4. For $k = 1$ to $n_1 + n_2 - 1$
5. $\quad$ If $p_j^2.x < p_i^1.x$ then
6. $\quad\quad \bar{S}[t-1] = p_i^1$; $\quad i = i - 1$
7. $\quad$ Else
8. $\quad\quad \bar{S}[t] = p_j^2$; $\quad j = j - 1$

9. $\quad$ If $\bar{S}[t-1].y > \bar{S}[t].y$ then
10. $\quad\quad t = t - 1$

11. Return$(\bar{S}[t \ldots n_1 + n_2])$

Finally, the algorithms above worked by scanning the sorted points from right to left without ever using the property that S^1 and S^2 were individually maximal sets.

It is also possible to write a merge algorithm that goes from left to right, using the property that S^1 and S^2 were individually maximal. We present it without proof of correctness

Merge3(S^1, S^2)

0. Let $\bar{S}$ be an array of size $n_1 + n_2$.
% $\bar{S}[1 \ldots t]$ will store current maxima found.
% i, j will denote the current S^1, S^2 points being scanned

% Step 1: Initialization
1. $t = 0$; $p^1_{n_1} = (\infty, -\infty)$; $p^2_{n_2} = (\infty, -\infty)$;
2. $i = 1; j = 1$

% Step 2: Scan through points from left to right, storing maxima so far
% and testing if current point is maximal and, if not, scanning past it

4. For $k = 1$ to $n_1 + n_2$ % Choose next point in x order

5. If $p^1_i.x < p^2_j.x$ then % Next point is p_i
6. If $p^1_i.y > p^2_j.y$ then
7. $t = t + 1$; $\bar{S}[t] = p_i$ % p_i is maximal
8. $i = i + 1$ % move to p_{i+1}

9. Else % Next point is p_j
10. If $p^2_j.y > p^1_i.y$ then
11. $t = t + 1$; $\bar{S}[t] = p_j$ % p_i is maximal
12. $j = j + 1$ % move to p_{j+1}

13. Return$(\bar{S}[1 \ldots t])$

2. We only prove correctness of Merge(S^1, S^2)
(and not Merge2(S^1, S^2)).

To prove correctness of Merge(S^1, S^2) we will need the following fact
(Correctness of the algorithm is proved on the next page):

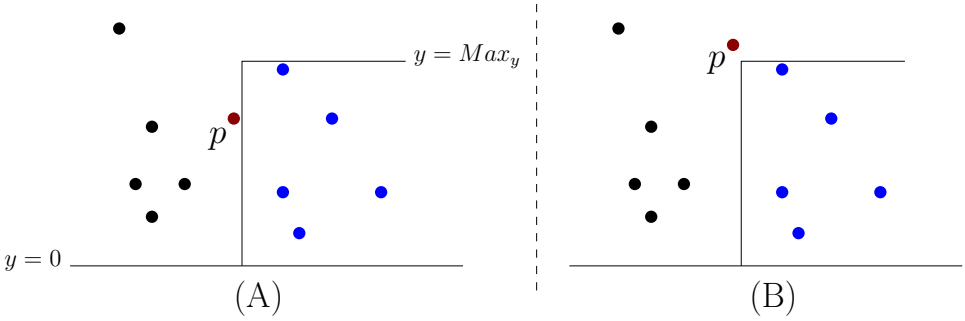
Fact 1: Let S be a point set and $p \notin S$. Then

$p \in L(\{p\} \cup S)$ if and only if $\forall q \in S$, if $p.x < q.x$ then $p.y > q.y$.

Proof of Fact 1:
Recall $\forall q \in S$ means "for all q in S.
Also, from the definition, if $p << q$ is not true, then
"if $p.x < q.x$ then $p.y > q.y$".

$p \in L(\{p\} \cup S)$ if and only if $\forall q \in S$, it is not true that $p << q$
if and only if $\forall q \in S$, if $p.x < q.x$ then $p.y > q.y$.

QED.



This diagram illustrates Fact 1.

- In both (A) and (B), S is the set of blue and black points and p is the new point to be added.
- The blue points are the ones with x-coordinate larger than $p.x$ and Max_y is the largest y-coordinate of a blue point.
- In (A), $p.y < Max_y$ so $p \notin L(\{p\} \cup S)$.
- In (B), $p.y > Max_y$ so $p \in L(\{p\} \cup S)$.

To see the correctness of the algorithm first note that lines 1-7 correctly sort the points in S by x-coordinate (this is exactly the same merge code we saw in Mergesort). After line 7, the points in $\bar{S}$ are the points in $S^1 \cup S^2$ sorted by x-coordinate.

For the purposes of analysis (this is not physically implemented by the algorithm) let S' be a copy of $\bar{S}$. S' will remain static.

Now consider lines 9-13. Note that Fact 1 implies that

$S'[k] \in L(S^1 \cup S^2)$
if and only if
$\forall q \in S' \setminus S'[k]$, if $S'[k].x < q.x$ then $S'[k].y > q.y$
if and only if
$\forall q \in S'[k+1, \ldots, n+1]$, $S'[k].y > q.y$
if and only if
$S'[k].y > \max_{j>k} S'[j].y$

Recall that, that at each loop iteration, before starting line 9, $Max_y = \max_{j>k} S'[j].y$

This means that when line 10 is checking whether $\bar{S}[k].y > Max_y$ it is correctly checking whether $\bar{S}[k] = S'[k]$ is a maximal point in $S^1 \cup S^2$. Since the algorithm stores each $\bar{S}[k]$ that passes this test, it is exactly returning the maximal points in $S^1 \cup S^2$. It is returning them in sorted order because it is returning them in the same order in which they appear in S', which is sorted order.

The fact that it runs in $O(n_1 + n_2)$ follows directly from the fact that each of the for loops (lines 3 and 9) do only $O(1)$ work for a fixed value of k.

3. The code is below

FINDL(S)
1. If $|S| = 1$ then
2. Return(S)
3. Else
   % S'_1 are S'_2 are two new arrays
4. $S'_1 = p_1, \ldots p_{\lfloor size(n)/2 \rfloor}$; $S'_2 = p_{\lceil size(n)/2 \rceil}, \ldots, n$.
5. $S_1 = FINDL(S'_1)$; $S_2 = FINDL(S'_2)$.
6. Return(Merge(S_1, S_2)).

4. For correctness we need Fact 3. We first prove the intermediate Fact 2.

   Fact 2: If $q \in S$ but $q \notin L(S)$ then $\exists p \in L(S)$ such that $q << p$.
   Proof of Fact 2: Rename q as q_0.
   If $q_0 \notin L(S)$ there exists a $q_1 \in L(S)$ such that $q_0 << q_1$.
   If $q_1 \notin L(S)$ there exists a $q_2 \in L(S)$ such that $q_1 << q_2$.
   If $q_2 \notin L(S)$ there exists a $q_3 \in L(S)$ such that $q_2 << q_3$.

   Because S is finite and a point can not be used twice (why?) this process must stop in a finite amount of time at some $q_i \in S$ yielding

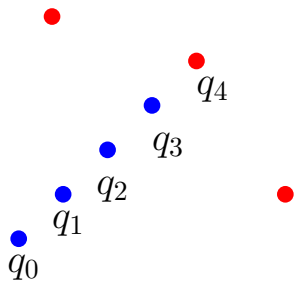
   $$q_0 << q_1 << \cdots << q_i.$$

   Setting $p = q_i$ gives $q = q_0 << q_i \in L(S)$, completing the proof. QED.

   The diagram below illustrates the proof. Red points are the maxima. q_0 is not maximal. The proof could build the chain

   $$q_0 << q_1 << q_2 << q_3 << q_4$$

   which shows that $q_0 << q_4 \in L(S)$.



Fact 3:

$$L(S_1 \cup S_2) = L\left(L(S_1) \cup L(S_2)\right).$$

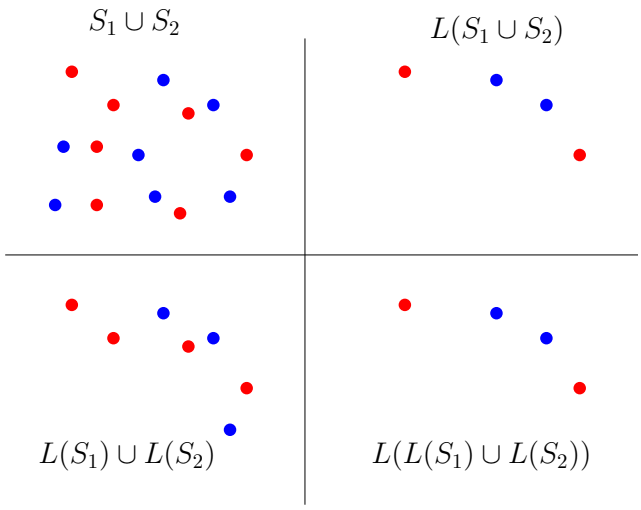
Proof of Fact 3: Let $p \in S_1$. We will show that

$$(1) \quad p \in L(S_1 \cup S_2) \quad \text{if and only if} \quad p \in L\left(L(S_1) \cup L(S_2)\right).$$

By then swapping S_1 and S_2 (1) is also correct for $p \in S_2$ and thus Fact 3 is correct.

First suppose $p \in S_1$ and $p \in L(S_1 \cup S_2)$.
By definition, there does not exist any $q \in S_1 \cup S_2$ such that $p << q$.
Since $L(S_1) \cup L(S_2) \subseteq S_1 \cup S_2$ this implies there is no $q \in L(S_1) \cup L(S_2)$ such that $p << q$.
Thus $p \in L\left(L(S_1) \cup L(S_2)\right)$.

Now suppose $p \in S_1$ but $p \notin L(S_1 \cup S_2)$.
From Fact 2 there exists $q \in S_1 \cup S_2$ with $p << q$.
There are four cases:
(i) $p \notin L(S_1)$: by definition, $p \notin L\left(L(S_1) \cup L(S_2)\right)$.
(ii) $p \in L(S_1)$, $q \in S_1$:
Not possible, since $p << q$, with $q \in S_1$ contradicts $p \in L(S_1)$.
(iii) $p \in L(S_1)$, $q \in L(S_2)$: Since $p << q$, $p \notin L\left(L(S_1) \cup L(S_2)\right)$.
(iv) $p \in L(S_1)$, $q \notin L(S_2)$, $q \in S_2$:
From Fact 2 there exists $q' \in L(S_2)$ such that $q << q'$.
But then $p << q << q'$ so $p << q'$ and again $p \notin L\left(L(S_1) \cup L(S_2)\right)$.
This completes the proof of both directions. so (1) is correct. QED.



The diagram above illustrates Fact 3 with S_1 being the red points and S_2 being the blue ones.

The correctness of the algorithm follows directly from Fact 3 and induction as follows,.

If $|S| = 1$ the algorithm is obviously correct.

If $|S| > 1$ the two recursive calls are correct by the induction hypothesis, returning $S_1 = L(S_1')$ and $S_2 = L(S_2')$, each sorted by x-coordinate. Item 2 proved that Merge(S_1, S_2) returns $L(S_1 \cup S_2)$ (sorted by increasing x-coordinate). Thus

$$\begin{array}{lrcll} (*) & L(S) & = & L(S_1' \cup S_2') & \\ & & = & L(L(S_1') \cup L(S_2')) & \text{(from Fact 3)} \\ & & = & L(S_1 \cup S_2) & \\ & & = & \text{Merge}(S_1, S_2) & \end{array}$$

so the algorithm returns the correct answer.

By observation the running time satisfies

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

(with the $O(n)$ coming from the Merge procedure). This is the merge-sort recurrence so

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

Marking Note: Any proof of correctness would be *required* to explicitly state Fact (3) and (*) or something equivalent and prove that they are correct.

Without such statements there is no way to know that the algorithm is correct since the last step of your algorithm is NOT explicitly finding the maximl points in the complete set. It is only finding the maximal points in a subset.