CS60-454
Design and Analysis of Algorithms
Winter 2017

Assignment 2

Due Date: March 2 (before lecture)

1. In Insertion sort, we use the *compare-and-swap* operation to do sorting. In this question, we shall use a more general operation *flip* to do sorting.

   Let $L : a_1, a_2, \ldots, a_n$ be a list of elements (drawn from a totally ordered set). The flip operation $\texttt{flip}(L, i, j)$ converts the list $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{j-1}, a_j, a_{j+1}, \ldots, a_n$

   $$\text{to } a_1, a_2, \ldots, a_{i-1}, a_j, a_{j-1}, \ldots, a_{i+1}, a_i, a_{j+1}, \ldots, a_n$$

   i.e. $\texttt{filp}(L, i, j)$ reverses the order of elements in the sublist $a_i, a_{i+1}, \ldots, a_{j-1}, a_j$.

   Assuming $\texttt{flip}(L, i, j)$ takes $O(j - i)$ time.

   (a) Given a list of elements $a_1, a_2, \ldots, a_n$ such that $a_i \in \{0, 1\}, 1 \leq i \leq n$. Present an algorithm that sorts the list in $O(n \lg n)$ time. You are allowed to use only the `flip` operation to rearrange the elements.

   (b) Redo Part (a) assuming that $a_i, 1 \leq i \leq n$, are drawn from a totally ordered set by presenting an $O(n^2 \lg n)$ time algorithm.

   Solution:

   (a) **Key idea:**

   Use a divide-and-conquer strategy similar to merge sort: Split the list into two equal halves and then recursively sort each half. Then find the index i of the first occurrence of 1 in the left-half and the index j of the last occurrence of 0 in the right-half. Apply `Flip` to the sublist $L[i..j]$.

   Algorithm $\texttt{Sort-with-Flip01}(L, \mathit{lower}, \mathit{upper})$
   Input: $L[\mathit{lower}..\mathit{upper}]$;
   Output: $L[\mathit{lower}..\mathit{upper}]$ sorted in ascending order;
   begin
   if $(\mathit{lower} < \mathit{upper})$ **then**
   $\texttt{Sort-with-Flip01}(L, \mathit{lower}, \lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor)$;
   $\texttt{Sort-with-Flip01}(L, \lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor + 1, \mathit{upper})$;

   /* Scan $L[\mathit{lower}..\lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor]$ to look for the first occurrence of 1 */
   $i := \mathit{lower}$;
   while $(i \leq \lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor \wedge L[i] = 0)$ **do** $i := i + 1$;

   /* Scan $L[\lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor + 1, \mathit{upper}]$ to look for the last occurrence of 0 */
   $j := \lfloor \frac{\mathit{lower}+\mathit{upper}}{2} \rfloor$;

while $(j < upper \wedge L[j+1] = 0)$ **do** $j := j+1$;
if $(i \leq \lfloor \frac{lower+upper}{2} \rfloor \wedge j \geq \lfloor \frac{lower+upper}{2} \rfloor + 1)$ **then** $\texttt{flip}(L, i, j)$;
end.

Theorem 1: Algorithm *`Sort-with-Flip01`* *correctly sorts the input list L into ascending order.*

Proof: (By induction on n)

Base case: When $n = 1$, L consists of one element which is a sorted list. Since $n = (lower - upper + 1) \Rightarrow 1 = (lower - upper + 1) \Rightarrow lower = upper$, the algorithm correctly returns L without doing anything to it.

Induction hypothesis: Suppose **Algorithm** `Sort-with-Flip01` correctly sorts all input lists of length $< n$.

Induction step: Consider an input list L of length n.

Let $p = \lfloor \frac{lower+upper}{2} \rfloor$.

By the induction hypothesis, the left sublist $L[lower..p]$ and the right sublist $L[p+1..upper]$ are sorted in ascending order.

On exiting the first **while** loop, either $i > \lfloor \frac{lower+upper}{2} \rfloor$ or $L[i] = 1$.

(i) In the former case, $L[i] = 0, lower \leq i \leq \lfloor \frac{lower+upper}{2} \rfloor$, which implies that the left sublist $L[lower..p]$ consists of a sequence of 0's.

(ii) In the latter case, $L[i] = 1$ and $L[i-1] = 0$ implies that $L[i]$ is the first occurrence of 1 in the left sublist which implies that $L[lower..(i-1)]$ (empty, if $i = lower$) consists of a sequence of 0's while $L[i..p]$ consists of a sequence of 1's.

On exiting the second **while** loop, either $j = upper$ or $L[j+1] = 1$.

(iii) In the former case, $L[i] = 0, \lfloor \frac{lower+upper}{2} \rfloor + 1 \leq i \leq upper$, which implies that the right sublist $L[(p+1)..upper]$ consists of a sequence of 0's.

(iv) In the latter case, if $j = \lfloor \frac{lower+upper}{2} \rfloor$, then the right sublist $L[(p+1)..upper]$ consists of a sequence of 1's.
Otherwise, $L[j] = 0$ which implies that $L[j]$ is the last occurrence of 0 in the right sublist. Hence, $L[(p+1)..j]$ consists of a sequence of 0's while $L[(j+1)..upper]$ consists of a sequence of 1's.

If $i > \lfloor \frac{lower+upper}{2} \rfloor$, then by Case ($i$), the left sublist $L[lower..p]$ consists of a sequence of 0's. Since the right sublist $L[p+1..upper]$ consists of a (possibly empty) sequence of 0's following by a (possibly empty) sequence of 1's (Cases (iii) or (iv)), the combined sublists is the list L in ascending order.

Moreover, $i > \lfloor \frac{lower+upper}{2} \rfloor$ implies that the last **if** statement if not executed, the algorithm thus returns L in ascending order.

If $j < \lfloor \frac{lower+upper}{2} \rfloor + 1$, then $j = \lfloor \frac{lower+upper}{2} \rfloor$. By Case ($iv$), the right sublist $L[(p+1)..upper]$ consists of a sequence of 1's. Since the left sublist $L[lower..p]$ consists of a (possibly empty) sequence of 0's following by a (possibly empty) sequence of 1's (Cases (i) or (ii)), the combined sublists is the list L in ascending order.

Moreover, $j < \lfloor \frac{lower+upper}{2} \rfloor + 1$ implies that the last **if** statement if not executed, the algorithm thus returns L in ascending order.

In the remaining cases, $i \leq \lfloor \frac{lower+upper}{2} \rfloor$ and $j \geq \lfloor \frac{lower+upper}{2} \rfloor + 1$. The **if** statement is execute as a result.

By Case (ii), $i \leq \lfloor \frac{lower+upper}{2} \rfloor$ implies that $L[i]$ is the first occurrence of 1 in the left sublist.

By Case (iv), $j \geq \lfloor \frac{lower+upper}{2} \rfloor + 1$ implies that $L[j]$ is the last occurrence of 0 in the right sublist.

Therefore, the sublist $L[lower..(i-1)]$ consists of a sequence of 0, the sublist $L[i..j]$ consists of a sequence of 1's following by a sequence of 0's, and the sublist $L[(j+1)..upper]$ consists of a sequence of 1's.

Since $\texttt{flip}(L, i, j)$ converts $L[i..j]$ into a sequence consisting of a sequence of 0's following by a sequence of 1's, after executing $\texttt{flip}(L, i, j)$, the resulting list L consists of a sequence of 0's following by a sequence of 1's which is in ascending order.

[e.g. For L : 000$\underline{111100000}$11, $\texttt{flip}(L, 4, 12)$ gives rise to 000$\underline{000001111}$11.] □

Theorem 2: ***Algorithm*** `Sort-with-Flip01` *takes* $O(n \lg n)$ *time to sort the input list* L *into ascending order.*

When $n \leq 1$, the outermost **if** statement is not executed. Therefore, $T(1) = O(1)$.

For $n \geq 2$, searching for the first occurrence of 1 in the left sublist takes $O(p)$ time (e.g. there is no 1 in the sublist) and searching for the last occurrence of 0 in the right sublist takes $O(n-p)$ time; (e.g. the sublist consists of 0's); the `flip` operation takes $O(n)$ time in the worst case (e.g. the left sublist consists of 1's and the right sublist consists of 0's). The total time spent on the body of the outermost **if** statement, excluding the recursive calls, is thus

$$O(p) + O(n-p) + O(n) = O(n)$$

We thus have the following recurrence for time complexity:

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n) & \text{if } \ n > 1; \\ O(1) & \text{if } \ n \leq 1. \end{cases}$$

The recurrence is similar to that of `Mergesort` and hence can be solved similar to `Mergesort` (see Chapter 2, p.31). We thus have $T(n) = O(n \lg n)$. ■

(b) **Key idea:**

Choose $a = L[1]$ as the pivot.

Scan L and label each element less than or equal to a with 0; each element greater than a with 1. Let $\ell[i]$ be the label of $L[i], 1 \leq i \leq n$.

Create $\mathcal{L}[1..n]$ such that $\mathcal{L}[i] = (\ell[i], L[i]), 1 \leq i \leq n$.

Sort $\mathcal{L}$ into ascending order of $\ell[i], 1 \leq i \leq n$, using **Algorithm** `Sort-with-Flip01` in Part (a).

[**Remark:** We can regard each $(\ell[i], L[i])$ as a binary number $\ell[i]$ with a value $L[i]$ attached to it. Therefore, whenever we move $\ell[i]$, the attached value $L[i]$ is moved along with it. In actual implementation, $\mathcal{L}$ can be represented by a list of records $(\ell[i], L[i]), 1 \leq i \leq n$, with $\ell[i]$ as the key for sorting, or as two arrays $\ell[i], 1 \leq i \leq n$, and $L[i], 1 \leq i \leq n$. In the latter case, whenever $\ell[i]$ is moved, $L[i]$ is also moved to the corresponding position in L.]

Scan $\mathcal{L}$ to look for the largest index m with $\ell[m] = 0$.
Then $L[i] \leq a, 1 \leq i \leq m$, and $L[i] > a, m < i \leq n$.
Apply $\texttt{Flip}(L, 1, m)$.
Then, recursively sort the two sublists $L[1..m-1]$ and $L[m+1..n]$.

A pseudo-code of the algorithm is as follows.

Algorithm $\texttt{Sort-with-Flip}(L, \mathit{lower}, \mathit{upper})$
Input: $L[\mathit{lower}..\mathit{upper}]$;
Output: $L[\mathit{lower}..\mathit{upper}]$ sorted in ascending order;
begin
if $(\mathit{lower} < \mathit{upper})$ **then**
    $\texttt{Split}(L, \mathit{lower}, \mathit{upper}, \mathit{splitpoint})$;
    $\texttt{Sort-with-Flip}(L, \mathit{lower}, \mathit{splitpoint} - 1)$;
    $\texttt{Sort-with-Flip}(L, \mathit{splitpoint} + 1, \mathit{upper})$;
end.

Procedure $\texttt{Split}(L, \mathit{lower}, \mathit{upper}, \mathit{splitpoint})$
Input: $L[\mathit{lower}..\mathit{upper}]$;
Output: $\mathit{splitpoint}$ such that $\begin{cases} L[i] \leq L[\mathit{lower}], & \mathit{lower} \leq i < \mathit{splitpoint}; \\ L[i] > L[\mathit{lower}], & \mathit{splitpoint} < i \leq \mathit{upper}. \end{cases}$
begin
    $a := L[\mathit{lower}]$;
    for $i := \mathit{lower}$ **step** 1 **to** upper **do**
        if $(L[i] \leq a)$ **then** $\ell[i] := 0$ **else** $\ell[i] := 1$;

    for $i := \mathit{lower}$ **step** 1 **to** upper **do** $\mathcal{L}[i] := (\ell[i], L[i])$;
    $\texttt{Sort-with-Flip01}(\mathcal{L}, \mathit{lower}, \mathit{upper})$;

    /* Scan $\mathcal{L}[\mathit{lower}..\mathit{upper}]$ to look for the first occurrence of 1 */
    $\mathit{splitpoint} := \mathit{lower}$;
    while $((\mathit{splitpoint} < \mathit{upper}) \wedge \ell[\mathit{splitpoint} + 1] = 0)$ **do** $\mathit{splitpoint} := \mathit{splitpoint} + 1$;
    $\texttt{Flip}(L, \mathit{lower}, \mathit{splitpoint})$;
end;

Lemma 1: ***Procedure*** $\texttt{Split}$ *partitions* $L[\mathit{lower}..\mathit{upper}]$ *into* $L[\mathit{splitpoint}]$ *and two sublists* $L[\mathit{lower}..(\mathit{splitpoint}-1)]$ *and* $L[(\mathit{splitpoint}+1)..\mathit{upper}]$ *such that* $L[\mathit{splitpoint}] = a$, $L[i] \leq a < L[j]$, *where* $\mathit{lower} \leq i < \mathit{splitpoint}$ *and* $\mathit{splitpoint} < j \leq \mathit{upper}$.
Proof:
On exiting the first **for** loop, we have $\ell[i] = 0 \Leftrightarrow L[i] \leq a$ and $\ell[i] = 1 \Leftrightarrow L[i] > a$. $\cdots$ (I)

When control returns from `Sort-with-Flip01`$(\mathcal{L}, lower, upper)$, by Theorem 1 of Part (a), ℓ is sorted into ascending order (i.e. it consists of a sequence of 0's following by a sequence of 1's)

$$\Rightarrow \exists m, lower \leq m \leq upper, \text{ such that } \begin{cases} \ell[i] = 0, lower \leq i \leq m \\ \ell[i] = 1, m < i \leq upper \end{cases}$$

$$\Rightarrow \exists m, lower \leq m \leq upper \text{ such that } \begin{cases} L[i] \leq a, lower \leq i \leq m, \\ L[i] > a, m < i \leq upper, \end{cases} \qquad \text{(by (I))}$$

$$\Rightarrow \exists m, lower \leq m \leq upper \text{ such that } L[i] \leq a < L[j],$$

$$\text{where } lower \leq i \leq m < j \leq upper. \cdots \text{(II)}$$

By applying a simple induction on *splitpoint*, it is easily verified that (I let you fill up the detail) at the begining of the ith iteration of the **while** loop, $L[lower..splitpoint]$ contains a sequence of 0's.

Therefore, on exiting the **while** loop, $L[lower..splitpoint]$ contains a sequence of 0's and $L[splitpoint + 1] = 1$ or $splitpoint = upper$. Hence, $splitpoint = m$.

Since $a = L[lower]$, $\ell[lower] = 0$. As the `Flip` operation only flips a subsequence beginning with a 1, $a = L[lower]$ remains unchanged throughout the execution of **Algorithm** `Sort-with-Flip01`.

Therefore, after `Flip`$(L, lower, splitpoint)$ is executed, $L[splitpoint] = a$

Hence, by letting $m = splitpoint$ in (II), the lemma follows. □

Theorem 2: Algorithm `Sort-with-Flip` correctly sorts the input list L into ascending order.

Proof: (By induction on input length n)

Induction basis: When $n \leq 1$, L is already sorted. The algorithm thus correctly returns L without doing anything to it.

Induction hypothesis: Suppose **Algorithm** `Sort-with-Flip` correctly sorts all input lists of length $< n (n \geq 2)$.

Induction step: Consider the input list $L[1..n]$.

By Lemma 1, **Procedure** `Split` partitions $L[1..n]$ into $L[splitpoint]$ and two sublists $L[1..(splitpoint - 1)]$ and $L[(splitpoint + 1)..n]$ such that $L[splitpoint] = a$ and $L[i] \leq a < L[j]$, where $1 \leq i < splitpoint$ and $splitpoint < j \leq n$.

Algorithm `Sort-with-Flip` is then called recursively to sort $L[1..(splitpoint - 1)]$ and $L[(splitpoint + 1)..n]$.

Since $splitpoint \leq n \Rightarrow splitpoint - 1 < n \Rightarrow L[1..(splitpoint - 1)]$ is of length $< n$, and

$$1 \leq splitpoint \Rightarrow n - splitpoint < n \Rightarrow L[1(splitpoint + 1)..n] \text{ is of length } < n,$$

by the induction hypothesis, **Algorithm** `Sort-with-Flip` correctly sorts $L[1..(splitpoint - 1)]$ and $L[(splitpoint + 1)..n]$ into ascending order.

It then follows that $\forall i, j, 1 \leq i < j \leq n, L[i] \leq L[j]$ (see the last part of correctness proof of `Quicksort` on the course webpage for detail).

Hence, $L[1..n]$ is in ascending order. □

Theorem 3: Algorithm `Sort-with-Flip` takes $O(n^2 \lg n)$ time to sort input list of length n.

Proof: As with the `Quicksort` algorithm presented in the lecture notes, the worst case occurs when $L[lower..(spltpoint-1)]$ is an empty list for *every* recursive call.

The algorithm will make a total of $n-1$ recursive calls.

For the ith recursive call, the length of the list is $n-i+1$. Since the time complexity of **Procedure** `Split` is dominated by the call to `Sort-with-Flip01`, by Theorem 2 of Part (a), **Procedure** `Split` takes $O((n-i+1)\lg(n-i+1)) = O(n\lg n)$ time.

It follows that each of the $n-1$ recursive calls takes $O(n\lg n)$ time.

Algorithm `Sort-with-Flip` thus takes a total of $\sum_{i=1}^{n-1} O(n\lg n) = (n-1)O(n\lg n) = O(n^2\lg n)$ time. ■

2. In analyzing the time complexity of **Algorithm** Select, we obtain the recurrence $T(n) = T(\lceil\frac{1}{5}\rceil n) + T(\frac{7}{10}n+3) + (6\lceil\frac{n}{5}\rceil + (n-1))$ from which we deduce $T(n) = O(n)$.

   Suppose that we have an algorithm for finding the kth smallest element whose time complexity is $T(n) = T(c_1 n) + T(c_2 n) + n$, where $0 < c_1, c_2 < 1$ and $c_1 + c_2 < 1$. Determine $T(n)$.

   Solution: Since when $c_1 = \frac{1}{5}$ and $c_2 = \frac{7}{10}$, the resulting recurrence is similar to that of **Algorithm** Select, and $c_1 + c_2 = \frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, we thus guess that $T(n) = O(n)$.

   We shall verify our solution by induction on n.

   Suppose $T(k) \le ck, \forall k < n.\ \cdots$ (I)

   Since $0 < c_1, c_2 < 1 \Rightarrow \frac{1}{c_1}, \frac{1}{c_2} > 1 \Rightarrow \frac{2}{c_1}, \frac{2}{c_2} > 2$.

   For $n \ge n_0 = \max\{\frac{2}{c_1}, \frac{2}{c_2}\} > 2, \cdots$ (II)

   $$c_1 + c_2 < 1 \text{ and } c_1, c_2 > 0$$
   $$\Rightarrow c_1 < 1 \text{ and } c_2 < 1$$
   $$\Rightarrow c_1 n < n \text{ and } c_2 n < n$$
   $$\Rightarrow T(c_1 n) \le c(c_1 n) \text{ and } T(c_2 n) \le c(c_2 n) \quad \text{(by (I))}$$

   Therefore,
   $$\begin{aligned} T(n) &= T(c_1 n) + T(c_2 n) + n \\ &\le c(c_1 n) + c(c_2 n) + n \\ &= (cc_1 + cc_2 + 1)n \end{aligned}$$

   Since
   $$\begin{aligned} (cc_1 + cc_2 + 1)n \le cn &\Leftrightarrow (cc_1 + cc_2 + 1) \le c \\ &\Leftrightarrow 1 \le c - (cc_1 + cc_2) \\ &\Leftrightarrow 1 \le c(1 - (c_1 + c_2)) \\ &\Leftrightarrow c \ge \frac{1}{1-(c_1+c_2)} \quad (\because 1 - (c_1 + c_2) > 0), \end{aligned}$$

   we thus have: $T(n) \le cn, \forall n \ge n_0$, for any $c \ge \frac{1}{1-(c_1+c_2)}$.

   Base cases: Let $c' = \max\{\frac{T(i)}{i} \mid 1 \le i < n_0\}$. we have $T(i) \le c'i, 1 \le i < n_0$.

   Hence, $T(n) \le \tilde{c}n, \forall n \ge 1$, where $\tilde{c} = \max\{c, c'\}$

   $$\Rightarrow \exists c > 0, n_0 \in N, T(n) \le cn, \forall n \ge n_0$$
   $$\Rightarrow T(n) = O(n). \quad ■$$

3. Let P be a set of points in the Euclidean plan. Let $\{S, \overline{S}\}$ be a partition of P (i.e. $S \subseteq P$ and $\overline{S} = P - S$). The *distance between* S *and* $\overline{S}$, denoted by $dist(S, \overline{S})$, is the shortest (Euclidean) distance between any pair of points p, q, where $p \in S$ and $q \in \overline{S}$ (i.e. $dist(S, \overline{S}) = \min\{d(p,q) | p \in S, q \in \overline{S}\}$, where $d(p,q)$ is the Euclidean distance between p and q). (Note: $d(p,q)$ is the length of the straight line segment connecting p and q).

   Present an algorithm that, given input P, determines a partition $\{W, \overline{W}\}$ of P such that $dist(W, \overline{W}) = \max\{dist(S, \overline{S}) \mid \{S, \overline{S}\}$ is a partition of $P\}$ by reduction to the minimum spanning tree problem. Each point p in P is represented by an order pair (x_p, y_q), where x_p and y_p are the x and y coordinates of p, respectively. Your reduction algorithm must run in $O(n^2)$ time, where $n = |P|$.

Solution:

Key idea: Let G be the complete weighted graph with vertex set P and edge weight being the Euclidean distance between the two end-vertices. Then the desired $dist(W, \overline{W})$ equals to the *maximum edge weight* of a minimum spanning tree of G.

Hence, the reduction algorithm-pair $(\mathcal{A}_\pi, A_S)$ is such that:

Algorithm $\mathcal{A}_\pi$ takes the point set P as input and produces the complete weighted graph G as output;

Algorithm A_S takes a minimum spanning tree T of G as input and returns a partition $\{W, \overline{W}\}$ of P such that $dist(W, \overline{W}) = \max\{dist(S, \overline{S}) \mid \{S, \overline{S}\}$ is a partition of $P\}$. $\square$

Let $G = (P, E, w)$ be the *complete* weighted graph such that P is the vertex set, E is the edge set, and $\forall\{p,q\} \in E, w(p,q) = d(p,q)$, where $w(p,q)$ is the weight of the edge $\{p,q\}$.

Lemma 1: *Let $T = (P, E_T, w)$ be a minimum spanning tree of G. Then $\exists\{S, \overline{S}\}$ such that $\max\{w(x,y) \mid \{x,y\} \in E_T\} = dist(S, \overline{S})$.*

Proof:

Let $\{s,t\} \in E_T$ such that $w(s,t) = \max\{w(x,y) \mid \{x,y\} \in E_T\}$. $\cdots$ (I)

Since T is a tree, T is circuit-free (Definition of tree)

$\Rightarrow \{s,t\}$ does not lie on a cycle ($\because$ a cycle is a circuit)

$\Rightarrow \{s,t\}$ is a bridge. (60-231 courseware, Theorem 11.1.1)

$\Rightarrow T - \{s,t\}$ is a disconnected graph consisting of two connected components $T_1 = (S_1, E_{T_1})$ and $T_2 = (S_2, E_{T_2})$ such that $s \in S_1 \wedge t \in S_2$.

(60-231 courseware, Section 11.2, e.g. 2)

Since S_1 and S_2 form a partition of P, $S_2 = \overline{S}_1$. (60-231 courseware, Lemma 10.2.5)

Let $p \in S_1$ and $q \in \overline{S}_1$ such that $d(p,q) = dist(S_1, \overline{S}_1)$. $\cdots$ (II)

If $\{p,q\} = \{s,t\}$, then $d(p,q) = d(s,t) \Rightarrow dist(S_1, \overline{S}_1) = d(s,t)$ (by (II))

$\Rightarrow dist(S_1, \overline{S}_1) = w(s,t)$ (Definition of w)

$\Rightarrow \max\{w(x,y) \mid \{x,y\} \in E_T\} = dist(S, \overline{S})$. (by (I))

If $\{p,q\} \neq \{s,t\}$, consider the graph $T \cup \{p,q\} \setminus \{s,t\}$ (the graph resulting from T after edge $\{p,q\}$ is added to T and edge $\{s,t\}$ is removed from T).

Since $s, p \in S_1$ and T_1 is connected, there is an $p - s$ path in T_1 and hence in T.

Likewise, $t, q \in S_2$ and T_2 is connected imply that there is an $t - q$ path in T_2 and hence in T.

It follows that the two paths and the two edges $\{p, q\}$ and $\{s, t\}$ form a cycle in $T \cup \{p, q\}$.

Since edge $\{s, t\}$ lies on the cycle, it is not a bridge (60-231 courseware, Theorem 11.1.1).

Therefore, removing $\{s, t\}$ does not result in a disconnected graph, i.e. $T \cup \{p, q\} \setminus \{s, t\}$ is connected.

Moreover, as T is a tree, $|E_T| = |P| - 1$ (60-231 courseware, Theorem 12.1.2(e)). $\cdots$(III)

Let E' be the edge set of $T \cup \{p, q\} \setminus \{s, t\}$.

Then $E' = E_T \cup \{\{p, q\}\} - \{\{s, t\}\} \Rightarrow |E'| = |E_T| + 1 - 1 = |E_T| = |P| - 1.$ (by (III))

Therefore, $T \cup \{p, q\} \setminus \{s, t\}$ is connected and $|E'| = |P| - 1$, where P is its vertex set

$\Rightarrow T \cup \{p, q\} \setminus \{s, t\}$ is a tree (60-231 courseware, Theorem 12.1.2(e)) and hence a spanning tree of G.

Since T is a minimum spanning tree,

$$
\begin{aligned}
& \sum_{e \in E'} w(e) \geq \sum_{e \in E_T} w(e) \\
\Rightarrow & \sum_{e \in E_T \cup \{\{p,q\}\} - \{\{s,t\}\}} w(e) \geq \sum_{e \in E_T} w(e) && \text{(Definition of } E'\text{)} \\
\Rightarrow & \sum_{e \in E_T} w(e) + w(p, q) - w(s, t) \geq \sum_{e \in E_T} w(e) \\
\Rightarrow & \sum_{e \in E_T} w(e) + d(p, q) - w(s, t) \geq \sum_{e \in E_T} w(e) && \text{(Definition of } w\text{)} \\
\Rightarrow & \; d(p, q) - w(s, t) \geq 0 \\
\Rightarrow & \; dist(S_1, \overline{S_1}) - w(s, t)) \geq 0 && \text{(by (II))} \\
\Rightarrow & \; w(s, t) \leq dist(S_1, \overline{S_1}) \cdots \text{ (IV)}
\end{aligned}
$$

On the other hand, $d(s, t) \in \{d(x, y) \mid x \in S_1 \wedge y \in \overline{S_1}\}$

$$
\begin{aligned}
&\Rightarrow \min\{d(x, y) \mid x \in S \wedge y \in \overline{S}\} \leq d(s, t) \quad \text{(Definition of min)} \\
&\Rightarrow dist(S_1, \overline{S_1}) \leq d(s, t) \quad \text{(Definition of } dist(S_1, \overline{S_1})\text{)} \\
&\Rightarrow dist(S_1, \overline{S_1}) \leq w(s, t) \quad (w(s, t) = d(s, t)) \cdots \text{ (V)} \\
&\Rightarrow w(s, t) \leq dist(S_1, \overline{S_1}) \wedge dist(S_1, \overline{S_1}) \leq w(s, t) \quad \text{((IV),(V), I6)} \\
&\Rightarrow w(s, t) = dist(S_1, \overline{S_1}) \quad (\leq \text{ is antisymmetric}) \\
&\Rightarrow \max\{w(x, y) \mid \{x, y\} \in E_T\} = dist(S_1, \overline{S_1}) \quad \text{(by (I))} \qquad \square
\end{aligned}
$$

Lemma 2: *Let $T = (P, E_T, w)$ be a minimum spanning tree of G. Then $\exists \{s, t\} \in E_T$ such that*

$$\max\{dist(S, \overline{S}) | \{S, \overline{S}\} \text{ is a partition of } P\} \leq w(s, t).$$

Proof: Let $dist(W, \overline{W}) = \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\}$. $\cdots$ (I)

Let $p \in W$ and $q \in \overline{W}$ such that $d(p, q) = dist(W, \overline{W})$.

Since T is a spanning tree of G

$\Rightarrow T$ is connected (Definition of tree)

$\Rightarrow$ there exists a path Q connecting p and q in T (Definition of connected graph)

$\Rightarrow \exists\{s, t\}$ on path Q, hence in T, such that $s \in W$ and $t \in \overline{W}$. ($\because p \in W$ and $q \in \overline{W}$)

Then $d(s, t) \in \{d(p', q') \mid p' \in W \wedge q' \in \overline{W}\}$ ($\because s \in W$ and $t \in \overline{W}$)

$\Rightarrow \min\{d(p', q') \mid p' \in W \wedge q' \in \overline{W}\} \leq d(s, t)$ (Definition of min)

$\Rightarrow dist(W, \overline{W}) \leq d(s, t)$ (Definition of $dist(W, \overline{W})$)

$\Rightarrow dist(W, \overline{W}) \leq w(s, t)$ (Definition of w)

$\Rightarrow \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\} \leq w(s, t)$. (by (I)) $\square$

Theorem 3: *Let $T = (P, E_T, w)$ be a minimum spanning tree of G and $\{s, t\} \in E_T$ such that $w(s, t) = \max\{w(x, y) \mid \{x, y\} \in E_T\}$. Let $\{W, \overline{W}\}$ be vertex sets of the connected components of $T - \{s, t\}$. Then*

$$dist(W, \overline{W}) = \max\{dist(S, \overline{S}) | \{S, \overline{S}\} \textit{ is a partition of } P\}.$$

Proof:

In the proof of Lemma 1, we proved that $\max\{w(x, y) \mid \{x, y\} \in E_T\} = dist(W, \overline{W}) \cdots$ (I)

Then $dist(W, \overline{W}) \leq \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\}$ (Definition of max) $\cdots$ (II)

$\Rightarrow \max\{w(x, y) \mid \{x, y\} \in E_T\} \leq \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\} \cdots$ (A)

((I), (II), $\leq$ is transitive)

By Lemma 2, $\exists\{s, t\} \in E_T$ such that

$$\max\{dist(S, \overline{S}) | \{S, \overline{S}\} \text{ is a partition of } P\} \leq w(s, t). \cdots \text{(III)}$$

Then $w(s, t) \leq \max\{w(x, y) \mid \{x, y\} \in E_T\}$ (Definition of max) $\cdots$ (IV)

$\Rightarrow \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\} \leq \max\{w(x, y) \mid \{x, y\} \in E_T\} \cdots$ (B)

((III), (IV), $\leq$ is transitive)

Hence, $\max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\} = \max\{w(x, y) \mid \{x, y\} \in E_T\}$.

((A),(B), $\leq$ is antisymmetric)

$\Rightarrow \max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\} = dist(W, \overline{W})$. (by (I)) $\square$

Theorem 3 shows that the problem of determining $\max\{dist(S, \overline{S}) | \{S, \overline{S}\}$ is a partition of $P\}$ can be reduced to that of determining the largest edge weight of a minimum spanning tree.

We thus have the following reduction algorithms $(\mathcal{A}_\pi, \mathcal{A}_\mathcal{S})$.

Algorithm $\mathcal{A}_\pi$

Input: A set of points $P = \{p_i \mid p_i = (x_i, y_i), 1 \leq i \leq n\}$ on the plan;

Output: The edge set of a complete weighted graph $G = (P, E, w)$ such that:

$$w(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, 1 \le i < j \le n.$$

begin

for $i := 1$ **step** 1 **to** $n - 1$ **do**

  for $j := i + 1$ **step** 1 **to** n **do**

    output$(\{p_i, p_j\}, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2})$; // an edge with weight

end.

Algorithm $\mathcal{A}_{\mathcal{S}}$

Input: The edge set E_T of a minimum spanning tree T of G.

Output: A partition $(W, \overline{W})$ of P such that:

$$dist(W, \overline{W}) = \max\{dist(S, \overline{S}) \mid (S, \overline{S}) \text{ is a partition of } P\}$$

begin

/* Create an adjacency-lists structure for the spanning tree $T - \{s, t\}$, where $\{s, t\} \in E_T$ such that $w(s, t) = \max\{w(x, y) \mid \{x, y\} \in E_T\}$ */

1. **for** $i := 1$ **step** 1 **to** n **do** $A[p_i] := null$; // create an array of linked-list pointers

/* Read in the edges in E_T */

2. $max.edge := null$; $max.wt := 0$; // to record the edge in E_T with largest edge weight

3. **while** (input file is non-empty) **do**

  read$(\{x, y\}, w(x, y))$;

  if $(w(x, y) > max.wt)$ **then** // found new $max.edge$

    $max.wt := w(x, y)$; // update $max.wt$

    if $(max.edge \neq null)$ **then** // insert current $max.edge$ into the adjacency lists

      $\{x', y'\} := max.edge$;

      Insert y' into $A[x']$ // the adjacency list of x';

      Insert x' into $A[y']$ // the adjacency list of y';

    $max.edge := \{x, y\}$; // update $max.edge$

  else // insert edge $\{x, y\}$ into the adjacency lists

    Insert y into $A[x]$ // the adjacency list of x;

    Insert x into $A[y]$ // the adjacency list of y;

4. /* Determine $(W, \overline{W})$ such that $dist(W, \overline{W}) = \max\{dist(S, \overline{S}) \mid (S, \overline{S})$ is a partition of $P\}$

  $\{s, t\} := max.edge$;

  Traverse $T - \{s, t\}$ starting from vertex s to determine W;

  Traverse $T - \{s, t\}$ starting from vertex t to determine $\overline{W}$;

end.

Theorem 4: Algorithm $\mathcal{A}_\pi$ correctly creates the complete weighted graph $G = (P, E, w)$.

Proof: This is trivial and I let you fill in the detail. $\square$

Lemma 5: Algorithm $\mathcal{A}_\pi$ takes $O(n^2)$ time, where $n = |P|$.

Proof: The body of the inner for loop takes $O(1)$ time.

The inner **for** loop takes $\sum_{i<j\leq n} O(1)$ time, where $1 \leq i < n$.

The outer **for** loop thus takes $\sum_{1\leq i<n} \sum_{i<j\leq n} O(1)$

$$= \sum_{1\leq i<j\leq n} O(1)$$

$$= \frac{n(n-1)}{2} O(1)$$

$$= O(\frac{n(n-1)}{2})$$

$$= O(n^2) \text{ time.} \quad \square$$

Lemma 6: *In the course of executing **Algorithm** $\mathcal{A}_\mathcal{S}$, at the end of the kth iteration of the **while** loop, $A[1..n]$ is an adjacency-lists structure of the graph induced by the vertex set P and the k edges read in thus far excluding one that has the largest weight among them which is $max.edge$ and whose weight is $max.edge$.*

Proof: (By induction on k)

You should be able to complete a simple proof of such nature by now. So, I let you fill in the detail. $\square$

Theorem 7: Algorithm $\mathcal{A}_\mathcal{S}$ correctly determines a partition $\{W, \overline{W}\}$ of P such that

$$dist(W, \overline{W}) = \max\{dist(S, \overline{S}) | \{S, \overline{S}\} \text{ is a partition of } P\}.$$

Proof:

When execution of the **while** loop in Step 3 terminates, the edges in E_T have all been read in. By Lemma 5, $A[1..n]$ is an adjacency-lists structure of the graph induced by P and the edge set $E_T - \{max.edge\}$.

Let $max.edge = \{s, t\}$. Then $A[1..n]$ is an adjacency-lists structure of the graph $T - \{s, t\}$.

Since T is a minimum spanning tree of G and $w(s, t) = \max\{w(u, v) \mid \{u, v\} \in E_T\}$, let $\{W, \overline{W}\}$ be the vertex sets of the connected components of $T - \{s, t\}$.

By Theorem 3, $dist(W, \overline{W}) = \max\{dist(S, \overline{S}) | \{S, \overline{S}\} \text{ is a partition of } P\}$.

Without loss of generality, let $s \in W$ and $t \in \overline{W}$. Since $W \cap \overline{W} = \emptyset$, in Step 4, the first traversal of $T - \{s, t\}$ starting from s determines W while the second traversal of $T - \{s, t\}$ starting from t determines $\overline{W}$ (note: the traversal can be any tree/graph traversal techniques you learned in 60-254 (Data Structures)).

The theorem thus follows. $\square$

Lemma 8: Algorithm $\mathcal{A}_\mathcal{S}$ takes $O(n)$ time.

Proof:

Step 1 takes $O(n)$ time. Step 2 takes $O(1)$ time.

In Step 3, since it takes $O(1)$ time to insert an entry at the beginning of an adjacency list, the body of the **while** loop takes $O(1)$ time per iteration. Hence, constructing the adjacency list structure for $T - max.edge (= T - \{s, t\})$ takes $O(n)$ time as $|E_T| = n - 1$.

In Step 4, T is a tree $\Rightarrow$ the two connected components of $T - \{s, t\}$ are trees.

Since W and $\overline{W}$ are the vertex sets of the connected components, traversing the two connected components takes $O(|W|)$ and $O(|\overline{W}|)$ time, respectively (see your Data structures textbook).

As $\{W, \overline{W}\}$ is a partition of P, the total time spent on Step 4 is thus $O(|W|) + O(|\overline{W}|) = O(|P|) = O(n)$.

Hence, **Algorithm** $\mathcal{A}_{\mathcal{S}}$ takes $O(n) + O(1) + O(n) + O(n) = O(n)$ time. $\square$

Theorem 9: The reduction algorithms $(\mathcal{A}_\pi, \mathcal{A}_{\mathcal{S}})$ takes $O(n^2)$ time.

Proof: Immediate from Lemmas 5 and 8. $\blacksquare$