Assignment 2

Quinn Perfetto, 104026025
60-454 Design and Analysis of Algorithms

February 17, 2017

Question 1 (a).

Algorithm 1: FlipSort(L, lower, upper)

Input: $L[lower..upper], lower \leq i \leq upper,\ L[i] \in \{0,1\}$
Output: $L[lower..upper]$ sorted in ascending order

begin
  if *upper - lower > 1* **then**
    FlipSort(L, lower, $\lfloor\frac{lower+upper}{2}\rfloor$);
    FlipSort(L, $\lfloor\frac{lower+upper}{2}\rfloor + 1$, upper);
    return *Merge*($L\left[lower..\lfloor\frac{lower+upper}{2}\rfloor\right],\ L\left[\lfloor\frac{lower+upper}{2}\rfloor + 1..upper\right]$)
  end
end

Algorithm 2: Merge(A, B)

Input: Two sorted lists $A[1..n]$ and $B[1..m]$ over $\{0,1\}$
Output: A sorted list $C[1..m+n]$ containing all elements of A and B
Let: $<>$ be the list concatenation operator

begin
  $index_A$:= SequentialSearch(A, 1);
  $index_B$:= SequentialSearch(B, 1);
  if $index_A == 0$ **then**
    return $A <> B$
  end
  if $index_B == 0$ **then**
    return $B <> A$
  end
  return
  $A[1..index_A - 1] <> flip(A[index_A..n] <> B[1..index_B - 1]) <> B[index_B..m]$
end

Lemma 1.1. Algorithm Merge correctly produces a sorted list

Note that SequentialSearch(L, x) refers to the algorithm defined in Chapter 0 Page 10 of the CourseWare, and returns the index of the first occurrence of x in L if it exists, and 0 otherwise. For the sake of brevity, we take $L[1..0]$ as [] (the empty list).

Case 1: $index_A == 0$
$index_A == 0 \Rightarrow A$ contains no instance of 1, i.e. $1 \leq i \leq n,\ A[i] == 0$. Since $0 \leq 0 \leq 1$ and B is assumed to be a sorted list over $\{0, 1\}$, by the transitivity of $\leq$ $A <> B$ must also be sorted. Thus the algorithm works correctly.

Case 2: $index_B == 0$
This case is similar to the above case, I thus omit the detail.

Case 3: $index_A \geq 1 \wedge index_B \geq 1$
Without loss of generality, let $A = 0^x 1^{n-x}$ and $B = 0^y 1^{m-y}$ such that $x, y \geq 0,\ x \leq n,\ y \leq m$. Since $index_A$ refers to the first occurrence of 1 in A, $A[1..index_A - 1] = 0^x$ and $A[index_A..n] = 1^{n-x}$. By a similar argument for $index_B$, $B[1..index_B - 1] = 0^y$ and $B[index_B..m] = 1^{m-y}$. Thus,

$$
\begin{aligned}
&A[1..index_A - 1] <> flip(A[index_A..n] <> B[1..index_B - 1]) <> B[index_B..m] \\
&= 0^x <> flip(1^{n-x}, 0^y) <> 1^{m-y} \\
&= 0^x <> (0^y <> 1^{n-x}) <> 1^{m-y} \\
&= 0^x 0^y 1^{n-x} 1^{m-y}
\end{aligned}
$$

Which is a sorted list of length $x + y + n - x + m - y = n + m$. Therefore the algorithm works correctly.

Lemma 1.2. Algorithm FlipSort correctly produces a sorted list.

Induction on the size of the input n.
(Induction Basis) If $n = 1$ FlipSort performs no operations and returns a single element list which is vacuously sorted.
(Induction Hypothesis) Assume that FlipSort correctly sorts all lists of size $n \leq k,\ n > 1$.
(Induction Step) Let $L'[lower..upper]$ be a list of length $k + 1$, i.e. $upper - lower = k + 1$.
The first recursive call produces a list of length,

$$
\begin{aligned}
&\lfloor \frac{lower + upper}{2} \rfloor - lower \\
&\leq \frac{lower + upper}{2} - lower \\
&= \frac{upper - lower}{2} \\
&< upper - lower && (n > 1) \\
&= k + 1
\end{aligned}
$$

Thus by the Inductive Assumption the first recursive call produces a correctly sorted list $L'[lower..\lfloor\frac{lower+upper}{2}\rfloor]$ (I).
Additionally the second recursive call produces a list of length,

$$\begin{aligned}
&upper - \lfloor\frac{lower+upper}{2}\rfloor + 1 \\
&\leq upper - \frac{lower+upper}{2} + 1 \\
&= \frac{upper-lower}{2} + 1 \\
&= \frac{k+1}{2} + 1 \\
&< k+1 && (k+1 > 2)
\end{aligned}$$

Thus by the Inductive Assumption the second recursive call produces a correctly sorted list $L'[\lfloor\frac{lower+upper}{2}\rfloor..upper]$ (II).

Finally by Lemma 1.1, (I) and (II) we know that the Merge Algorithm correctly merges the resulting lists into a sorted list $L'[lower..upper]$.
Therefore Algorithm FlipSort works correctly.

Lemma 1.3. Algorithm Merge requires at most $2n + 2m$ operations

As proved in the CourseWare SequentialSearch search performs at most n comparisons for $index_A$ and at most m comparisons for $index_B$. Further, since $Flip$ requires $O(j-i)$ time and $j - i \leq n + m$ we have at most $(n+m) + (n+m) = 2n + 2m$ operations.

Lemma 1.4. Algorithm FlipSort is $\theta(nlgn)$

Let $T(n)$ be the time required to sort a list of n elements with FlipSort.

$$T(n) = \begin{cases} T(\lfloor\frac{n}{2}\rfloor) + T(\lceil\frac{n}{2}\rceil) + 2n & n > 1 \\ 0 & otherwise \end{cases}$$

Let $T_{\lfloor\rfloor}(n) = 2T(\lfloor\frac{n}{2}\rfloor) + 2n$ and $T_{\lceil\rceil}(n) = 2T(\lceil\frac{n}{2}\rceil) + 2n$.
Using the general formula for solving recurrences, we have
$f(n) = 2n = \theta(n) = \theta(n^{log_2 2}) = \theta(n^{log_b a} lg^0 n)$

Therefore $T_{\lfloor\rfloor}(n) = T_{\lceil\rceil}(n) = \theta(nlgn)$.
Then $T_{\lfloor\rfloor}(n) \leq T(n) \leq T_{\lceil\rceil}(n) \Rightarrow T(n) = \theta(nlgn)$.

Therefore Algorithm FlipSort correctly sorts the input and runs in $\theta(nlgn)$ time. ■

Question 1 (b).

Algorithm 3: InsertFlipSort

Input: An array of elements $A[1..n]$ drawn from a totally ordered set
Output: $A[1..n]$ sorted in ascending order

```
begin
    for i := 2 to n do
        j := i - 1;
        while A[j] ≻ A[j + 1] ∧ j > 0 do
            Flip(A, j, j + 1);
            j := j - 1;
        end
    end
end
```

Lemma 1.4. Algorithm InsertFlipSort correctly produces a sorted list

We shall prove by induction that just before the kth iteration of the outer most for loop, $A[1..k]$ is sorted.
(Induction Basis) For $k = 1$ we have $A[1..k] = A[1..1]$ which is a vacuously sorted single element list.
(Induction Hypothesis) We assume just before the $k-1th$ iteration that $A[1..k-1]$ is sorted. We note that $i = k$.
(Induction Step) In order to prove that this invariant holds after the $k-1th$ iteration, we will apply induction on the number of iterations of the inner while loop to show that just before the mth iteration of the loop, $A[k-m+1..k]$ is sorted.

(Induction Basis') When $m = 1$, $A[k-m+1..k] = A[k..k]$ which is a vacuously sorted single element list.
(Induction Hypothesis') Suppose that just before the $m-1th$ iteration $A[k-(m-1)+1..k] = A[k-m+2..k]$ is sorted. Note that $j = k-(m-1) = k-m+1$.
(Induction Step') Following the $m-1th$ iteration we have,

$$\begin{aligned}&A[j] \succ A[j+1] \land j > 0\\ &\Rightarrow A[k-m+1] \succ A[k-m+2] \land k-m+1 > 0\end{aligned}$$

Calling Flip(A, k - m +1, k - m +2) effectively swaps the two positions as they are adjacent in the array. Following the call to Flip we have,

$$\begin{aligned}&A[k-m+1] \precsim A[k-m+2] \land A[k-m+2..k] \text{ is sorted}\\ &\Rightarrow A[k-m+1..k] \text{ is sorted}\end{aligned}$$

Therefore just before the start of the mth iteration $A[k-m+1..k]$ is sorted, i.e. the proposed invariant holds for all iterations of the loop (I).

(Induction Step) Given (I), and substituting j for $k-m$, upon terminaiton of the inner while loop we have,

$$A[1..k-1] \text{ is sorted} \wedge A[j+1..k] \text{ is sorted} \wedge (j = 0 \vee A[j] \lesssim A[j+1]) \qquad (j = k-m)$$

Case 1: j = 0
This means that the inner while loop iterated $k-1$ times, following the $k-1th$ iteration we have,

$$A[j+1..k] \text{ is sorted} \wedge j = 0 \Rightarrow A[1..k] \text{ is sorted}$$

Thus the invariant holds.

Case 2: $A[j] \lesssim A[j+1]$
We thus have,

$$\begin{aligned} &A[j] \lesssim A[j+1] \wedge A[j+1..k] \text{ is sorted} \\ &\Rightarrow A[j..k] \text{ is sorted} \\ &A[1..k-1] \text{ is sorted} \wedge A[j..k] \text{ is sorted} \Rightarrow A[1..k] \text{ is sorted} \end{aligned}$$

Thus the invariant holds.
Therefore Following the nth iteration we have $A[1..n]$ is sorted.
Hence, the Algorithm works correctly.

Lemma 1.5. FlipSort runs in O($n^2 lgn$) time

Key Operations: Comparison of integers, calls to Flip
Given that the inner while loop will perform at most $i-1$ comparisons, and make at most $i-1$ calls to Flip, each of which run in O$(j+1-j)$ = O(1) time, we have,

$$\begin{aligned} &\sum_{i=2}^{n} 2(i-1) \\ &= 2\sum_{i=2}^{n} i - 1 \\ &= 2(\sum_{i=2}^{n} i - \sum_{i=2}^{n} 1) \\ &= 2(\frac{n(n+1)}{2} + 1 - (n-1)) \\ &= 2(\frac{n^2+n}{2} + 1 - n + 1) \\ &= n^2 - n + 4 \\ &= O(n^2) \\ &= O(n^2 lgn) \end{aligned}$$

Algorithm 4: MaximumMinimumDistance

Input: A set of points P in the Euclidean plane
Output: $\{W, \overline{W}\}$ such that $dist\{W, \overline{W}\}$ is maximized over all partitions of P
Let: a weighted undirected edge E be defined by the 2-tuple $(\{u, v\}, w)$ where u and v are the endpoints of E and w is the weight

PairwiseDistanceGraph := $\{(\{u, v\},\ d(u, v)) \mid u,\ v \in P\}$;
MST := MinimumSpanningTree(PairwiseDistanceGraph);
HeaviestEdge := max(MST by w);
W, $\overline{W}$:= PairwiseDistanceGraph - {HeaviestEdge};

Question 3.

Lemma 3.1. Algorithm MaximumMinimumDistance correctly produces $\{W, \overline{W}\}$

We shall first prove that $\{W, \overline{W}\}$ is a disjoint parition of P.

Proof. Note that $\forall p \in P$, p is a vertex of PairwiseDistanceGraph. Since MinimumSpanningTree is assumed to work correctly, it follows that $\forall p \in P$, p is a vertex of MST (I). Since every edge in a tree is a cut edge, MST - {HeaviestEdge} must produce a disconnected graph containing two disjoint components, i.e. W and $\overline{W}$. By (I) we thus have $W \cap \overline{W} = \emptyset$ and $W \cup \overline{W} = P$. Thus $\{W, \overline{W}\}$ is a disjoint partition of P. □

It remains to show that $dist(W, \overline{W})$ is maximum over all other paritions of P.

Proof. Suppose there exists a disjoint partition $\{X, \overline{X}\}$ of P such that,

$$dist(X, \overline{X}) > dist(W, \overline{W}),\ \{X, \overline{X}\} \neq \{W, \overline{W}\}$$

Since $\{X, \overline{X}\} \neq \{W, \overline{W}\}$ there must exist some pair of vertices $\{u, v\}$ such that $\{u, v\}$ are in the same component of $\{W, \overline{W}\}$ and different components of $\{X, \overline{X}\}$. Let H_e be the weight of the heaviest edge that was removed from MST. This implies that $dist(W, \overline{W}) = H_e$ and $d(u, v) \leq H_e$. Since u and v lie in different components of $\{X, \overline{X}\}$, then $dist(X, \overline{X}) \leq d(u, v)$. Hence we have,

$$\begin{aligned}
& dist(W, \overline{W}) = H_e \wedge d(u, v) \leq H_e \\
& \Rightarrow d(u, v) \leq dist(W, \overline{W}) && (I) \\
& dist(X, \overline{X}) \leq d(u, v) && (II) \\
& \Rightarrow dist(X, \overline{X}) \leq dist(W, \overline{W}) && \text{(I, II, Transitivity)}
\end{aligned}$$

Which is a contradiction!

Therefore $dist(W, \overline{W})$ must be a maximum over all other paritions of P. □

Therefore the Algorithm MaximumMinimumDistance correctly produces the disjoint parition $\{W, \overline{W}\}$ of P such that,

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

Lemma 3.2. Algorithm MaximumMinimumDistance runs in O(n^2) time where $n = |P|$

Key Operations: Computing the Euclidean distance between points, comparison of integers

There are $\binom{|P|}{2}$ distinct pairs of points within P. Thus to compute the PairwiseDistanceGraph, $\binom{|P|}{2}$ Euclidean distances must be calculated.

The minmum spanning tree of the PairwiseDistanceGraph will have at most $\binom{|P|}{2}$ edges, therefore determining the heaviest edge will require at most $\binom{|P|}{2}$ - 1 comparisons.

Together we have,

$$\begin{aligned}
&\binom{|P|}{2} + \binom{|P|}{2} - 1 \\
&= 2\binom{|P|}{2} - 1 \\
&= |P|(|P| - 1) - 1 \\
&= |P|^2 - |P| - 1 \\
&= O(|P|^2)
\end{aligned}$$

Therefore Algorithm MaximumMinimumDistance runs in O(n^2) time where $n = |P|$.