Assignment 2

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

March 6, 2017

Question 1 (a).

Idea: FlipSort is essentially MergeSort with a modified Merge subroutine that assumes the sorted input arrays consist solely of 1's and 0's. These sorted arrays can be merged by flipping the inner unsorted subsequence between the first occurrence of 1 in either.

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.

We shall show this by 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,

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

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).

Idea: QuickFlipSort is a variant of QuickSort which uses FlipSort to perform the partition operation.

Algorithm 3: QuickFlipSort

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

begin
  if *upper - lower > 1* **then**
    $A'[lower+1..upper] := \{\text{y} \mid x \in A[lower+1..upper], \text{y} = \text{x} \gtrsim A[lower]\}$;
    FlipSort(A', lower+1, upper);
    firstone := SequentialSearch(A', 1);
    splitpoint := $\begin{cases} upper, & \text{if firstone} == 0 \\ \text{firstone} - 1, & \text{otherwise} \end{cases}$;
    Flip(A, lower, splitpoint);
    QuickFlipSort(A, lower, splitpoint - 1);
    QuickFlipSort(A, splitpoint + 1, upper);
  end
end

Lemma 1.5. Algorithm QuickFlipSort correctly produces a sorted list.

We shall show this by induction on n, the size of the input array.
(Induction Basis) For $n = 1$ the algorithm performs no operations and thus produces a vacuously sorted single element list.
(Induction Hypothesis) Assume that QuickFlipSort correctly sorts all lists of size $n,\ \forall n \leq k,\ n > 1$.
(Induction Step) Let $L = [lower..upper]$ such that $upper - lower = k + 1$.
Since the correctness of FlipSort has been proven, A' will contain a sorted list of 0's and 1's such that all 0's correspond to values in $A[lower+1..upper]$ that are less than $A[lower]$, and all 1's correspond to values that are greater than or equal to $A[lower]$.

We shall now show that following the call to Flip,

$$A[splitpoint] \gtrsim A[i], i < splitpoint \wedge A[splitpoint] \lesssim A[i], i > splitpoint \text{ (I)}$$

i.e. $A[splitpoint]$ is in it's correctly sorted position.

Case 1: firstone == 0
This implies that $\nexists A[i] \gtrsim A[lower], lower + 1 \leq i \leq upper$, i.e. $A[lower]$ is the maximum value in $A[lower..upper]$. In this case $splitpoint = upper$ and hence the call Flip(A, lower, splitpoint = upper) places $A[lower]$ at the end of the input list. Since $A[lower]$ is the maximum value, it is trivial to see that the above claim holds.

Case 2: firstone > 0
This implies that there are $upper - firstone$ values greater than $A[lower]$, and $firstone - lower - 1$ values less than $A[lower]$. Following the call to Flip(A, lower, splitpoint = firstone - 1) we have $A[lower]$ in position $firstone - 1$. Therefore there are $upper - firstone$ values following in the list, and $firstone - lower - 1$ values preceding in the list. Thus $A[splitpoint]$ is in it's correctly sorted position.

It remains to show that the recursive calls produce correctly sorted lists.
Since $splitpoint \leq upper$,

$$\begin{aligned} &splitpoint - lower - 1 \leq upper - lower - 1 \\ &\Rightarrow splitpoint - lower - 1 \leq k \qquad \text{(upper - lower = k + 1)} \end{aligned}$$

Therefore QuickFlipSort(A, lower, splitpoint - 1) will produce a correctly sorted list $A[lower..splitpoint - 1]$ by the induction hypothesis (II).
Similarly since $splitpoint \geq lower$,

$$\begin{aligned} &upper - splitpoint \leq upper - lower \\ &\Rightarrow upper - splitpoint - 1 \leq upper - lower - 1 \\ &\Rightarrow upper - splitpoint - 1 \leq k \qquad \text{(upper - lower = k + 1)} \end{aligned}$$

Therefore QuickFlipSort(A, splitpoint + 1, upper) will produce a correctly sorted list $A[splitpoint + 1..upper]$ (III).
Given (I), (II), and (III) we have $A[lower..upper]$ is sorted.
Thus, the Algorithm correctly sorts the input.

Lemma 1.6. Algorithm QuickFlipSort performs at most O($n^2 lgn$) operations

Much like traditional QuickSort, QuickFlipSort performs the most operations when $A[lower]$ is the maximum element in the list. In this case Flip(A, lower, splitpoint) performs O(n) operations, and we reduce the size of the input by 1 for each recursive call. Given that FlipSort performs $\theta(nlgn)$ operations we have,

$$T(n) = \begin{cases} T(n-1) + n + nlgn, & \text{if } n > 1 \\ 0, & \text{otherwise} \end{cases}$$

We guess that T(n) = O($n^2 lgn$).
(Induction Hypothesis) Assume $T(k) \leq ck^2 lgk, \ \forall k < n$

(Induction Step) Since $n-1 < n$ (I),

$$\begin{aligned}
T(n) &= T(n-1) + n + nlgn \\
&\leq c(n-1)^2 lg(n-1) + n + nlgn \qquad \text{(I)} \\
&= (cn^2 - 2cn + c)lg(n-1) + n + nlgn \\
&= cn^2 lg(n-1) - 2cnlg(n-1) + clg(n-1) + n + nlgn \\
&= cn^2 lg(n-1) - 3(\frac{2}{3}cnlg(n-1)) + clg(n-1) + n + nlgn \\
&= cn^2 lg(n-1) - (\frac{2}{3}cnlg(n-1) - clg(n-1)) - (\frac{2}{3}cnlg(n-1) - n) - (\frac{2}{3}cnlg(n-1) - nlgn)
\end{aligned}$$

$$\frac{2}{3}cnlg(n-1) \geq clg(n-1) \Rightarrow n \geq 3$$

$$\frac{2}{3}cnlg(n-1) \geq n \Rightarrow n \geq 3, c \geq \frac{3}{2}$$

$$\frac{2}{3}cnlg(n-1) \geq nlgn \Rightarrow c > 3, n \geq 3$$

Therefore,

$$\begin{aligned}
&cn^2 lg(n-1) - (\frac{2}{3}cnlg(n-1) - clg(n-1)) - (\frac{2}{3}cnlg(n-1) - n) - (\frac{2}{3}cnlg(n-1) - nlgn) \\
&\leq cn^2 lg(n-1) \\
&\leq cn^2 lg(n)
\end{aligned}$$

For $n \geq 3, c > max\{\frac{3}{2}, 3\}) = 3$.
(Induction Basis) Let $c' = max\{\frac{T(2)}{4}, 3\}$.
Then $T(n) \leq c'n^2 lgn, \ \forall n \geq 2$.
Hence, $T(n) = O(n^2 lgn)$.

Question 2. $T(n) = T(c_1 n) + T(c_2 n) + n$ where $0 < c_1, c_2 < 1$ and $c_1 + c_2 < 1$

We guess that $T(n) = O(n)$.
(Induction Hypothesis) Suppose $T(n) \leq ck, \ \forall k < n$ (I).
(Induction Step) Since $0 < c_1, c_2 < 1$, we have $c_1 n < n \wedge c_2 n < n, \ \forall n > 1$. Therefore,

$$\begin{aligned}
T(n) &= T(c_1 n) + T(c_2 n) + n \\
&\leq cc_1 n + cc_2 n + n \qquad \text{(I)} \\
&= n(cc_1 + cc_2 + 1)
\end{aligned}$$

We are to deduce when $cc_1 + cc_2 + 1 \leq c$,

$$\begin{aligned} & cc_1 + cc_2 + 1 \leq c \\ \Rightarrow & 1 \leq c - cc_1 - cc_2 \\ \Rightarrow & 1 \leq c(1 - (c_1 + c_2)) \\ \Rightarrow & \frac{1}{1 - (c_1 + c_2)} \leq c \qquad\qquad (c_1 + c_2 < 1 \Rightarrow 1 - (c_1 + c_2) > 0) \end{aligned}$$

Thus $T(n) \leq cn$ when $\frac{1}{1-(c_1+c_2)} \leq c$, $\forall n > 1$.

(Induction Basis) Let $T(1) = a$, where a is a constant.
Then, $T(1) = a \leq c$ for any $c \geq a$.

Therefore $T(n) \leq cn$, $\forall n \geq 1$.
Hence, $T(n) = O(n)$.

Question 3.

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};

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 $d(u, v) \leq H_e$ (by the Cut Property of MSTs). 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 $n = |P|$.