CSE3358 Problem Set 1
Solution

Please review the Homework Honor Policy on the course webpage
http://engr.smu.edu/~saad/courses/cse3358/homeworkpolicy.html

Problem 1: Bubble sort
We have studied Insertion sort in class and argued that it works correctly.

```
INSERTION-SORT($A$)
1 for $j \leftarrow 2$ to $length[A]$
2     do $key \leftarrow A[j]$
3         $i \leftarrow j - 1$
4         while $i > 0$ and $A[i] > key$
5             do $A[i+1] \leftarrow A[i]$
6                 $i \leftarrow i - 1$
7         $A[i+1] \leftarrow key$
```

At the beginning of each iteration of the "outer" **for** loop, which is indexed by j, the subarray consisting of elements $A[1..j-1]$ constitute the currently sorted elements, and the elements $A[j+1..n]$, where $n = length[A]$, correspond to the elements that are still to be considered. In fact, elements $A[1..j-1]$ are the elements originally in positions 1 through $j-1$, but now in sorted order. We can state this property of $A[1..j-1]$ formally as a **loop invariant**:

LOOP INVARIANT: *At the start of each iteration of the* **for** *loop of lines 1-7, the subarray* $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$ *but in sorted order.*

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

- **Initialization**: It is true prior to the first iteration of the loop
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1..j-1] = A[1..2-1] = A[1..1] = A[1]$, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the outer **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until the proper position for $A[j]$ is found (lines 3-6), at which point the value of $A[j]$ is inserted (line 7). A more formal treatment of this property would require us to state and show a loop invariant for the inner **while** loop (but we will not do it at this point).

Termination: Finally, we examine what happens when the loop terminates. For insertion sort, the outer **for** loop ends when j exceeds n, i.e. when $j = n + 1$. Substituting $n + 1$ for j in the wording of the loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. But the subarray $A[1..n]$ is the entire array! Hence, the entire array is sorted, which means that the algorithm is correct.

Consider the following algorithm known as Bubble sort. It works by repeatedly swapping adjacent elements that are out of order:

```
BUBBLE-SORT($A$)
1 for $i \leftarrow 1$ to $length[A]$
2     do for $j \leftarrow length[A]$ downto $i + 1$
3            do if $A[j] < A[j-1]$
4                  then exchange $A[j] \leftrightarrow A[j-1]$
```

(a) (5 points) Explain in english, very clearly, how the algorithm BUBBLE-SORT works (for this you need to understand what the code is doing and construct from it a process in your mind, one thing you could do to help is try it on some small examples, say 6 numbers, and see what it is doing).

ANSWER: BUBBLE-SORT works by first scanning the array A from $A[n]$ all the way down to $A[1]$ and exchanging any adjacent elements that are out of order. This first scan guarantees that the smallest element in $A[1..n]$ (i.e. the smallest element of A) will end up in $A[1]$. BUBBLE-SORT then scans the array A for a second time from $A[n]$ all the way down to $A[2]$ and exchanges again any adjacent elements that are out of order. This second scan guarantees that the smallest element in $A[2..n]$ (i.e. the next smallest element of A) will end up in $A[2]$. BUBBLE-SORT continues to perform scans in this way (up tp n scans), and in every scan it guarantees that the element with rank i ends up in $A[i]$. After the last scan, the array A is sorted.

(b) (5 points) Let A' denote the output of BUBBLE-SORT(A). To prove that BUBBLE-SORT is correct, we need to prove that **(1)** it terminates and that **(2)** $A'[1] \leq A'[2] \leq ... \leq A'[n]$, where $n = length[A]$. What else must we proved to show that BUBBLE-SORT actually sorts?

ANSWER: We need to show also that A' is some permutation of A. Otherwise, there is no guarantee that the elements of A' are those of A. For instance, here's an algorithm that takes an array A as input, terminates, and produces an array A' such that $A'[1] \leq A'[2] \leq ... \leq A'[n]$, but does not necessarily sort A.

```
FUNNY-SORT($A$)
1 for $i \leftarrow 1$ to $length[A]$
2     do $A[i] \leftarrow i$
```

(c) (10 points) Prove that the following loop invariant holds for the **for** loop in lines 2-4.

LOOP INVARIANT: *At the start of each iteration of the* **for** *loop of lines 2-4, the smallest element in the subarray* $A[i..n]$ *is in the subarray* $A[i..j]$.

Your proof should use the structure of the loop invariant proof presented above.

ANSWER:

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = length[A] = n$. The subarray $A[i..j] = A[i..n]$, and therefore, the claim is trivially true. In other words, the smallest element in the subarray $A[i..n]$ is in the subarray $A[i..n]$.

Maintenanace: We need to show that each iteration maintains the loop invariant. Assume the loop invariant holds prior to iteration j, i.e. the smallest element in $A[i..n]$ is in $A[i..j]$. We need to show that the invariant still holds prior to iteration $j-1$ (j goes downward). If the smallest element of $A[1..n]$ was in $A[i..j-1]$ prior to iteration j, we are done, since the smallest element never moves to the right. Therefore, prior to iteration $j-1$, the smallest element of $A[i..n]$ is in $A[i..j-1]$. If the smallest element of $A[1..n]$ was not in $A[1..j-1]$ prior to iteration j, this means that is was in $A[j]$ prior to iteration j, because it has to be in $A[i..j]$ prior to iteration j. In that case, during iteration j, $A[j]$, being the smallest element, will definitely be exchanged with $A[j-1]$, and from here on, that smallest element can never move to the right. Therefore, prior to iteration $j-1$, the smallest element in $A[i..n]$ is in $A[i..j-1]$.

Termination: Finally we examine what happens when the loop terminates. The loop ends when j exceeds $i+1$ from below, i.e. when $j = i$. Substituting i for j in the wording of the loop invariant, we have that the smallest element of $A[i..n]$ is in $A[i]$.

(d) (10 points) Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove $A'[1] \leq A'[2] \leq ... \leq A'[n]$. Your proof should use the structure of the loop invariant presented above.

ANSWER: Using the termination condition above for the inner loop (indexed by j) we can conceptually re-write the BUBBLE-SORT algorithm as follows:

```
BUBBLE-SORT($A$)
1      for $i \leftarrow 1$ to $length[A]$
2-4        do place smallest element of $A[i..length[A]]$ in $A[i]$
```

We can consider the following loop invariant for the main **for** loop.

LOOP INVARIANT: *At the start of every iteration of the main* **for** *loop, the subarray* $A[1..i-1]$ *contains the smallest* $(i-1)$ *elements of* A *in sorted order.*

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $i = 1$. The subarray $A[1..0]$ is empty, and herefore, it contains the smallest $(i-1) = 0$ elements of A in sorted order. The invariant is trivially true.

Maintenance: We need to show that each iteration maintains the loop invariant. Assume the loop invariant holds prior to iteration i, i.e. the subarray $A[1..i-1]$ contains the smallest $(i-1)$ elements of A in sorted order. We need to show that the invariant still holds prior to iteration $i+1$. Let's see what happens in iteration i. In iteration i, we place the smallest element of $A[i..n]$ in $A[i]$ without touching $A[1..i-1]$. Therefore, the smallest i elements of A will be in the subarray $A[1..i]$ in sorted order prior to iteration $i+1$.

Termination: Finally we examine what happens when the loop terminates. The loop ends when i exceeds $length[A] = n$, i.e. when $i = n+1$. Substituting $n+1$ for i in the wording of the loop invariant, we have that the subarray $A[i..n]$ contains the smallest n elements of A in sorted order. But this is the whole array, which gives the desired result.

(e) (5 points) What is the worst-case running time of BUBBLE-SORT? Use Θ notation as we did in class. How does it compare to the running time of insertion sort?

ANSWER:

```
BUBBLE-SORT($A$)
1 for $i \leftarrow 1$ to $length[A]$
2     do for $j \leftarrow length[A]$ downto $i+1$                $\Theta(n-i)$
3             do if $A[j] < A[j-1]$                             $\Theta(1)$
4                  then exchange $A[j] \leftrightarrow A[j-1]$     $\Theta(1)$
```

As we can see from the above, the running time $T(n) = \sum_{i=1}^{n} \Theta(n-i)[\Theta(1)+\Theta(1)] = \sum_{i=1}^{n} \Theta(n-i)\Theta(2) = \sum_{i=1}^{n} \Theta(n-i)\Theta(1) = \sum_{i=1}^{n} \Theta(n-i) = \Theta(\sum_{i=1}^{n}(n-i)) = \Theta((n-1)n/2) = \Theta(n^2)$. So this is comparable to Insertion sort in terms of asymptotic efficiency.

(f) (5 points) Which one you think is faster in practice, INSERTION-SORT or BUBBLE-SORT, and why?

ANSWER: Regardless of the input array A, Bubble sort scans $A[1..n]$, then $A[2..n]$, then $A[3..n]$, ..., $A[n..n]$. Therefore, the amount of work done by BUBBLE-SORT is $\Theta(n + (n-1) + (n-2) + ... + 1)$ which is equal to $\Theta(1 + 2 + ... + (n-1) + n)$. This is the worst-case scenario of Insertion sort where in every iteration the maximum amount of shifting is done. Therefore, practically, Insertion sort performs better than Bubble sort.

(g) (10 points) Implement both in any programming language of your choice. Try them on randomly generated inputs of size 10, 20, 50, 100, 500, 1000. Report their real running times as measured by the system clock. Does the numbers agree with your intuition on part (f)?