Practice Set 1: Solutions

Problem 1:

The pseudocode for insertion sort was given in class:

for $i = 2$ to n
  for $j = i$ down to 2
    If $A[j] < A[j-1]$
      Swap $A[j]$ and $A[j-1]$
    else break

Number of swaps:

In the best-case scenario, the array is already sorted. Note that there are 0 swaps performed in this case. In the worst-case scenario, the array is sorted in reverse order. The first iteration of the above loop performs one swap. The second iteration performs two swaps, etc. The last iteration performs $n-1$ swaps. Therefore the total number of swaps in the worst-case is

$$1 + 2 + \ldots + (n-1) = \frac{n^2}{2} - \frac{n}{2}$$

Number of comparisons:

The best-case scenario is when the array is already sorted. At each iteration of the outer for-loop, the inner for-loop only performs *one* comparison, and then breaks. There are $n-1$ executions of the inner for-loop, and so there are a total of $n-1$ comparisons. The worst-case scenario was examined in class, and we showed that the worst-case number of comparisons is $\frac{n^2}{2} - \frac{n}{2}$

Problem 2:

The outer for loop of insertion sort carries out $n-1$ iterations. If the array is already sorted, the inner for loop will perform a single comparison, and then break. Therefore each iteration of the inner for loop runs in constant time, c. Any initial operations performed by the function also run in constant time, d. Therefore the total best-case runtime is $T(n) = c(n-1)+d$. We can show that this is $O(n)$ because $T(n) = cn - c + d \leq (c+d)n$.

In summary, the best-case runtime of Insertion sort is $O(n)$ and the worst-case runtime of insertion sort (shown in class) is $O(n^2)$. It turns out that on *average* the runtime of insertion sort is nevertheless $O(n^2)$.

Problem 3:

The reverse procedure will swap elements at opposite ends of the array. The for loop proceeds through half the array making swaps. Note that the index of the lower median is $\lfloor (j+i)/2 \rfloor$. Therefore the swapping can terminate at index $\lfloor (j+i)/2 \rfloor$.

Reverse(A, i, j)
  ind = j
  for $k = i$ to $\lfloor (j+i)/2 \rfloor$
    Swap $A[k]$ and $A[$ ind $]$
    ind = ind - 1

If array A contains n elements, the algorithm carries out a for loop of exactly $\lfloor n/2 \rfloor$ iterations, regardless of the input ordering. Therefore the best and worst-case runtimes are the same. Each iteration of the for loop runs in constant time. Therefore the runtime of algorithm is $T(n) = a\frac{n}{2} + b$ for some constant a. This can be shown to be $O(n)$ as follows:

$$T(n) = a \cdot \frac{n}{2} + b \leq (a+b)n$$

for all $n \geq 1$. By definition of big-oh, the reverse algorithm runs in time $O(n)$.

Problem 4

Selection sort works by continuously selecting the minimum from the unsorted section of the array, and then placing it in the sorted section of the array, using a swap. Suppose the input to the algorithm is array A indexed from 1 to n. The pseudo-code is given below:

```
for i= 1 to n-1
    min_index = i
    for j = i+1 to n
       if A[j] < A[min_index]
          min_index = j
    Swap A[min_index] and A[i]
```

Each iteration of the the inner for loop runs in constant time, say d. There are $n-1$ iterations of the inner for-loop when $i=1$, accounting for a total time of $d(n-1)$. There are $n-2$ iterations of the inner for-loop when $i=2$, accounting for a total time of $d(n-2)$. The pattern continues and the sum of all the operations carried out by the inner for loop is exactly:

$$d(n-1)+d(n-2)+\ldots+d(1)=d\left(\frac{n(n-1)}{2}\right)$$

There are also exactly $n-1$ swaps that are performed, and each swap takes constant time, say k, accounting for $k(n-1)$ time. The total runtime is therefore $d\left(\frac{n(n-1)}{2}\right)+k(n-1)$, which has the form $T(n)=an^2+bn+c$. We can show this is $O(n^2)$:

$$T(n)=an^2+bn+c\leq an^2+bn^2+cn^2=(a+b+c)n^2$$

Note that in the above analysis, the number of steps performed is always the same, *regardless* of the input. Therefore the best-case and worst-case runtimes are identical: they are both $O(n^2)$. This differs from Insertion Sort, which has a best-case runtime of $O(n)$. Therefore if there is a high chance that your input is sorted, Insertion sort should be preferred.

Problem 5:

The algorithm below is similar to selection sort. It works by finding the index of the minimum element in the array indexed between i and j It then **reverses** the subarray so that the minimum is in the correct sorted position. The procedure is the repeated on the remaining elements. This process continues until the entire array is sorted. The initial call is to RSort$(A,1,n)$:

RSort(A,i,j)
  for $a=i$ to $j-1$
    min-index $=a$
    for $b=a+1$ to j
      if $A[b]<A$[min-index]
      min-index $=b$
    Reverse(A, a, min-index)

Problem 6

Assume that the input to Bubble sort is an array A indexed from 1 to n. The algorithm below passes through the elements of the array and swaps any two adjacent elements that are not in relative order. The process continues until a full pass can be made without any swaps.

Bubble-Sort$(A[1,\ldots,n])$

```
made-swap = true
while(made-swap)
     made-swap = false
     for i = 1 to n-1
         if A[i+1] < A[i]
             Swap A[i] and A[i+1]
             made-swap = true
```

Worst-case: If the array is sorted in **reverse** order, then after each pass of Bubblesort the maximum element of the unsorted section is moved into the last position. For example, starting with $5,4,3,2,1$ after one pass of bubble-sort the array becomes: $4,3,2,1,5$, and after the second pass the array becomes $3,2,1,4,5$. Each pass places exactly **one** element in its final sorted position, and therefore n passes are required to finish sorting the array. Each pass performs exactly $n-1$ comparisons in the for-loop, and therefore the worst-case number of comparisons is $n(n-1)$. The number of swaps in the worst case is identical: The number of swaps made during the first pass is $n-1$, during the second is $n-2$, etc. Therefore the total number of swaps is $n(n-1)/2$ in the worst case.

Since the number of swaps and comparisons in the worst-case is of the form $T(n)=an^2+bn+c$, the run-time is **quadratic** and is therefore $O(n^2)$.

Best-case: Bubblesort is effective if the input array is **already sorted**. It is able to detect this right away, and return the sorted array after one pass. This is because on a sorted array, $1,2,3,4,5$, Bubblesort performs no swaps on the first pass, and therefore will not repeat a second pass. Therefore the best-case number of swaps is 0 and the best-case number of comparisons is $n-1$. The runtime is therefore $T(n)=an+b$, which we have shown is $O(n)$.

Problem 7:

As noted in the previous question, after the first pass of bubble sort, the maximum element is placed in the last position. Therefore the next iteration only needs to verify swaps up to the second last element. The update is shown below:

```
for k = n-1 down to 1
     made-swap = false
     for i = 1 to k
         if A[i+1] < A[i]
               Swap A[i] and A[i+1]
               made-swap = true
     if made-swap = false
         break
```

The inner for loop is executed exactly k times, for $k = n-1$ down to 1. Each iteration of the inner for loop runs in constant time. As shown in the previous problem, the outer for loop is executed the maximal number of times when the input array is sorted in reverse order. Therefore, the total runtime in the worst case is of the form $c(1+2+3+\ldots+(n-1)) = c(n(n-1)/2) = c(\frac{n^2}{2} - \frac{n}{2}) \leq cn^2$ and so is $O(n^2)$.

Problem 8:

- The result of each pass of SelectSort1 on A is shown below:

  pass 1: 5, 4, 3, 2, 1, 6, 7

  pass 2: 3, 2, 1, 4, 5, 6, 7

  pass 3: 1, 2, 3, 4, 5, 6, 7

- SelectSort1 is not a correct algorithm, because it does not always result in a sorted array. For example, on input $A = 6, 5, 4, 3, 2, 1$, the result of the algorithm is $A = 2, 1, 4, 3, 6, 5$. SelectSort2 is a correct algorithm (for arrays of length $n \geq 3$) because the second while loop is a copy of the bubble-sort algorithm, which we know correctly sorts an array

- Given an array of length n that is sorted in reverse order, we consider two cases. If n is odd, then SwapSort2 produces a correctly sorted array after the first while loop. However, if n is even, the first while loop terminates with an unsorted array. Therefore the worst-case is when n is even. There are $n-2$ comparisons made on each iteration of the first while loop, and the while loop terminates after $n/2$ passes. Therefore the total number of comparisons in the first while loop is $(n-2)n/2$. Next the algorithm proceeds to the second while loop. Consider as an example the input $A = 8, 7, 6, 5, 4, 3, 2, 1$, which will be $2, 1, 4, 3, 6, 5, 8, 7$ after the first while loop. The second while loop performs $n-1$ comparisons at each pass. The array is sorted after one single pass, and a second pass is performed in order to terminate. A total of $2(n-1)$ comparisons are made in the second while loop.

  Therefore the worst-case number of comparisons for an input array sorted in reverse order is is $n(n-2)/2 + 2(n-1)$

Problem 9:

- TRUE. Since the algorithm runs in $O(n^2)$, then $T(n) \leq cn^2$, and so $T(n) \leq cn^3$. Therefore the algorithm is $O(n^3)$.
- FALSE. This statement is not always true. For example, if $T(n) = n^2$, then $T(n)$ is $O(n^2)$ but NOT $O(n)$.
- FALSE. This statement is not always true. For example, if $T(n) = n$, then $T(n)$ is $O(n^2)$ but NOT $\Theta(n^2)$.
- TRUE. Since the algorithm runs in time $\Omega(n^2)$, then $T(n) \geq cn^2 \geq cn$. Therefore the algorithm is also $\Omega(n)$.
- FALSE. This statement is not always true. For example, if $T(n) = n^2$, then $T(n)$ is $\Omega(n^2)$ but is NOT $\Omega(n^3)$.

Problem 10:
Since $f(n) = n^2 + \log n + n \leq 3n^2 \leq 3n^3$, then $f(n)$ is $O(n^2)$ and $O(n^3)$ but not $O(n)$.
Since $f(n) \geq n^2 \geq n \geq \log n$, then $f(n)$ is $\Omega(n^2)$ and $\Omega(n)$ and $\Omega(\log n)$.
Since $f(n)$ is $O(n^2)$ and $\Omega(n^2)$, then $f(n)$ is $\Theta(n^2)$. It is NOT $\Theta(n)$.

Problem 11:

- $f(n) = \log(n^2) + \log^2(n) + \sqrt{n}$
  $= 2\log n + (\log n)^2 + n^{.5}$ → Show $f(n)$ is $\Theta(n^{0.5})$

  $O(n^{1/2})$: Goal: Show $f(n) \leq c \cdot n^{0.5}$

  $2\log n + (\log n)^2 + n^{.5} \leq 2 \cdot n^{.5} + (n^{1/4})^2 + n^{.5} = 4n^{.5}$ $\therefore f \leq 4n^{.5}$ $\forall n \geq K$ $\therefore f(n) \leq 3n^{2.5}$

  $\log n \leq n$ $\forall n \geq 16$ $\log n \leq n^{1/4}$ $\forall n \geq K$

  $\Omega(n^{0.5})$: Goal: Show $f(n) \geq c \cdot n^{0.5}$

  $2\log n + (\log n)^2 + n^{.5} \geq n^{.5}$ $\therefore f(n) \geq n^{.5}$ $\forall n \geq 1$

- $f(n) = n^2(\log n) + n(\log n)^2$ is $\Theta(n^2 \cdot \log n)$

$O(n^2 \cdot \log n)$: Goal: $f(n) \le c \cdot n^2 \log n$

$f(n) = n^2(\log n) + n(\log n)(\log n) \le n^2 \cdot \log n + n(n)\log n = 2n^2 \cdot \log n$ $\therefore f$ is $O(n^2 \log n)$

($\log n \le n \ \forall n \ge 1$)

$\Omega(n^2 \log n)$ Goal: Show $f(n) \ge c \cdot n^2 \cdot \log n$

$n^2(\log n) + n(\log n)^2 \ge n^2 \cdot \log n$ $\therefore f$ is $\Omega(n^2 \cdot \log n)$

- $f(n) = n^3 + n^2 \cdot \log(n)$ is $\Theta(n^3)$

$O(n^3)$: Goal: $f(n) \le c \cdot n^3$

$n^3 + n^2 \cdot \log n \le n^3 + n^2 \cdot n = 2n^3$ $\therefore f$ is $O(n^3)$

($\log n \le n \ \forall n \ge 1$)

$\Omega(n^3)$ Goal: $f(n) \ge c \cdot n^3$

$n^3 + n^2 \cdot \log n \ge n^3$ $\therefore f$ is $\Omega(n^3)$

- $f(n) = \sum_{K=1}^{n} (2K+1) = \sum_{K=1}^{n} 2K + \sum_{K=1}^{n} 1 = \frac{2n(n+1)}{2} + n = 2n^2 + 2n + n = 2n^2 + 3n$ is $\Theta(n^2)$

$O(n^2)$: Goal: $f(n) \le c \cdot n^2$

$f = 2n^2 + 3n \le 2n^2 + 3n^2 = 5n^2$ $\therefore f$ is $O(n^2)$

$\Omega(n^2)$: Goal: show $f(n) \ge c \cdot n^2$

$f = 2n^2 + 3n \ge 2n^2$ $\therefore f$ is $\Omega(n^2)$

Problem 12:

- $f(n) = n \cdot \log_2 n + n \log_3 n$ Show $f(n)$ is $\Theta(n \log_2 n)$ (ANY CONSTANT BASE!)

$O(n \log n)$: Goal: Show $f(n) \le c \cdot n \log_3 n$

$n \cdot \log_2 n + n \cdot \log_3 n \le n \log_2 n + n \log_2 n = 2n \log_2 n$ $\therefore f$ is $O(n \cdot \log n)$

($\log_3 n \le \log_2 n$)

$\Omega(n \log n)$ Goal: Show $f(n) \ge c \cdot n \cdot \log_2 n$

$n \log_2 n + n \log_2 n \ge n \log_2 n$ $\therefore f$ is $\Omega(n \cdot \log n)$

- $f(n) = (2^n + n \cdot 2^n)(n^2 + 3^n) = 2^n \cdot n^2 + 2^n \cdot n^3 + 6^n + 6^n \cdot n$ $f(n)$ is $\Theta(6^n \cdot n)$

$O(6^n \cdot n)$ Goal: Show $f(n) \le c \cdot 6^n \cdot n$

$f(n) = 2^n \cdot n^2 + 2^n \cdot n^3 + 6^n + 6^n \cdot n$

$= n \cdot (2^n \cdot n) + n \cdot (2^n \cdot n^2) + 6^n + 6^n \cdot n \le 6^n \cdot n + 6^n \cdot n + 6^n \cdot n + 6^n \cdot n$

($2^n \le 6^n \ \forall n \ge 1$; $2^n \le 6^n \ \forall n \ge 1$)

$= 4 \cdot (6^n \cdot n)$

$\therefore f(n)$ is $O(6^n \cdot n)$

$\Omega(6^n)$ Goal: Show $f(n) \ge c \cdot 6^n \cdot n$

$f(n) = 2^n \cdot n^2 + 2^n \cdot n^3 + 6^n + 6^n \cdot n \ge 6^n \cdot n$ $\therefore f(n)$ is $\Omega(6^n \cdot n)$

- $f(n) = \log(n^{.2}) + \log(n^2) = 0.2 \log n + 2 \log n = 2.2 \log n$ is $\Theta(\log n)$

$f(n) = \underbrace{2.2}_{c} \log(n)$ is $O(\log n)$ and $\Omega(\log n)$

- $f(n) = n^{0.2} + \underbrace{\log(n^8)}_{c} = n^{0.2} + 8 \cdot \log n$ is $\Theta(n^{0.2})$

$O(n^{0.2})$ Goal: show $f(n) \leq c \cdot n^{0.2}$

$f(n) = n^{0.2} + \underbrace{8 \cdot \log n}_{\leq n^{0.2} \text{ for } n \geq K} \leq n^{0.2} + 8 \cdot (n^{0.2}) = 9 \cdot n^{0.2}$ $\therefore f$ is $O(n^{0.2})$

$\Omega(n^{0.2})$ Goal: show $f(n) \geq c \cdot n^{0.2}$

$f(n) = n^{0.2} + 8 \cdot \log n \geq n^{0.2}$ $\therefore f$ is $\Omega(n^{0.2})$

Problem 13:

- $f(n) = n^2 + n$

$\Omega(n^2)$ Goal: $f(n) \geq cn^2$

$f(n) = n^2 + n$

$\geq n^2$

$\therefore f$ is $\Omega(n^2)$ } Tighter lower bound.

$\Omega(n)$ Goal: $f(n) \geq c \cdot n$

$f(n) = n^2 + n$

$\geq n$

$\therefore f$ is $\Omega(n^2)$

- $f(n) = n^2 - 3n$

$O(n^3)$:

Goal: $f(n) \leq c \cdot n^3$

$f(n) = n^2 - 3n$

$\leq n^2 \leq n^3$

$\therefore f$ is $O(n^3)$

$O(2^n)$:

Goal: $f(n) \leq c \cdot 2^n$

$f(n) = n^2 - 3n$

$\leq n^2$

$\leq 2^n \quad \forall n \geq 4$

$\therefore f$ is $O(2^n)$

$O(n^2)$

Goal: $f(n) \leq c \cdot n^2$

$f(n) = n^2 - 3n$

$\leq n^2$

$\therefore f$ is $O(n^2)$

TIGHTEST UPPER BOUND.

Problem 14:

From smallest to biggest (asymptotically):

$(\log n^3)$	$\Theta(\log n)$
$\frac{n^2+1}{n+6}$ and $\sqrt{n^2 + \log n}$	$\Theta(n)$
$n(\log n)^2$	$\Theta(n(\log n)^2)$
$n^2(\log n)$	$\Theta(n^2 \log n)$
$(2^n \cdot n^2)$	$\Theta(2^n \cdot n^2)$
$(n \cdot 3^n)$	$\Theta(n \cdot 3^n)$
$(4^n + n)$	$\Theta(4^n)$
$6n!$	$\Theta(n!)$
n^n	$\Theta(n^n)$