Student Name: ______________________________ J#: ________________

CSC 323 Algorithm Design and Analysis, Fall 2017
Instructor: Dr. Natarajan Meghanathan
Quiz 1 Solutions (Sept. 12, 2017) Max. Points: 35 Max. Time: 20 min.

1) (15 pts) Given below is the pseudo code for a sorting algorithm called "Selection Sort" that works as follows on an array A[1...n].
(i) Identify the basic operation.
(ii) Derive the best-case and worst-case number of times the basic operation will be executed.
(iii) Determine the asymptotic time complexity of the algorithm by using the most appropriate notation.

```
for i = 1 to n do
        min_index = i
        for j = i+1 to n do
                if A[j] < A[min_index]
                        min_index = j
                end if
        end for
        Swap(A[i], A[min_index])
end for
```

Solution:
The logic is similar to the logic of finding the minimum element in an array. However, in this algorithm, in each iteration (index i ranging from 1 to n), we seek to put the ith smallest element at index i by scanning the array from index (j =) i+1 to n.
During the beginning of the ith iteration, we assume the ith smallest element is at index i (min_index) and scan through the array to the right (index j = i+1 to n) to see if there is any index j such that A[j] < A[min_index]; if such an index is come across, we set min_index to be j. After the j loop is exited during the ith iteration, min_index will have the correct index for the ith smallest element and we have to just swap the current element at index i with the element at min_index.

Consider the following array of 6 elements:

Index	**1**	**2**	**3**	**4**	**5**	**6**
Data	**4**	**5**	**2**	**1**	**7**	**8**

During the 1st iteration (i = 1), we seek to find the 1st smallest element at put it at index i. In this pursuit, we set the min_index = 1 (assuming element 4 is the temporary 1st smallest element) and compare A[min_index] with everything to the right (inside the j loop). We come across an index j = 3 for which A[j=3] < A[min_index=1] and we set min_index = 3. Upon further scanning, we come across index j = 4 for which A[j=4] < A[min_index=3] and we set min_index = 4. The value of min_index stays the same for the rest of the j loop. After exiting the j loop, we swap A[min_index = 4] with A[i=1]. So, the array looks like this at the end of the first iteration.

Index	**1**	**2**	**3**	**4**	**5**	**6**
Data	**1**	**5**	**2**	**4**	**7**	**8**

During the 2nd iteration (i=2), we work on the array at the end of the 1st iteration. We seek to find the 2nd smallest element by starting with an estimate for min_index = 2. We run the j loop from index i+1 to n, where i = 2. We come across j = 3 for which A[j=3] < A[min_index = 2]; so we set min_index = 3 and it remains the same for the rest of the j loop. After exiting the j loop, we swap A[min_index = 3] with A[i=2]. So, the array looks like this at the end of the second iteration.

Index	**1**	**2**	**3**	**4**	**5**	**6**
Data	**1**	**2**	**5**	**4**	**7**	**8**

We repeat each iteration like this and make sure that the ith smallest element is at index i at the end of the ith iteration.

(i) Comparison (A[j] with A[min_index]) is the basic operation that we do repeatedly in the algorithm.
(ii)
Best case: We have to run the j loop from index i+1 to n for each value of i. Even if the input array is sorted (for e.g., 2 4 5 7 8 9), we have to make sure that everything to the right of '2' is indeed greater than or equal to 2. Likewise, for every element, we need to check all the elements to its right.

Worst case: The worst case is same as best case. For an ith iteration, we need to run the j loop from index i+1 to n just to make sure that we pick the ith smallest element from index i to n and put such an element in the ith index at the end of the ith iteration.

Best case/Worst case # comparisons = $\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}[(n)-(i+1)+1] = \sum_{i=1}^{n}(n-i)$
= (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 + 0
= 1 + 2 + 3 + ... + (n-1)
= n(n-1)/2

(iii) Since best case = worst case, the most asymptotic notation to use would be Θ and the asymptotic time complexity is $\Theta(n^2)$.

2) (10 pts) Derive the asymptotic relationship between the two functions: $n^2\log(n)$ and $n\log(n^{100})$
Let f(n) = $n^2\log(n)$
Let g(n) = $n\log(n^{100})$ = (n)*(100*logn)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n^2\log(n)}{100n*\log(n)} = \lim_{n\to\infty}\frac{n}{100} = \infty$$

The numerator function f(n) grows much faster than the denominator function g(n). Hence, we have to use the Big-O notation and the faster growing function f(n) goes inside the O notation.
We have g(n) = O(f(n)) That is, $n\log(n^{100}) = O(n^2\log(n))$

3) (10 pts) Let f(n) = $5n^3$ + 6n + 2. Find a function g(n) such that f(n) = O(g(n)) and f(n) ≠ Θ(g(n)). Show that your choice for g(n) is correct using the Limits approach.

We need to choose a function g(n) that strictly grows much faster than f(n) and NOT at the same rate as f(n). So, if the most dominating term in the f(n) function is a n^3 term, we need to choose g(n) to be a function whose most dominating term is larger than n^3 as n->∞. We will choose g(n) = n^4 and prove that f(n) = $5n^3$ + 6n + 2 = O(n^4) using the limits approach.

$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{5n^3+6n+2}{n^4} = \lim_{n\to\infty}\left(\frac{5}{n}+\frac{6}{n^3}+\frac{2}{n^4}\right) = 0$. This implies the denominator function grows much faster than the numerator function as n->∞. Hence, f(n) = $5n^3$ + 6n + 2 = O(n^4).