Benjamin Tate
04/09/2017
CS 325 – Section 400
Homework #1

1) $8n^2 < 64n\lg n$
   True when **n > 2^(1/8)**
2)
   a. f(n) = O(g(n)), because $\lim_{n\to\infty}(n^{0.25})/(n^{0.5}) = 0$.
   b. f(n) = Ω(g(n)), because $\lim_{n\to\infty}(n)/(\log^2 n) = \infty$.
   c. f(n) = Θ(g(n)), because $\lim_{n\to\infty}(\log n)/(\ln n) = 1/\ln(2)$.
   d. f(n) = Θ(g(n)), because the limit as n approaches ∞ is 5,000,000.
   e. f(n) = O(g(n)), because the limit as n approaches ∞ is 0.
   f. f(n) = O(g(n)), because the limit as n approaches ∞ is 0.
   g. f(n) = Θ(g(n)), because the limit as n approaches ∞ is ½.
   h. f(n) = Ω(g(n)), because the limit as n approaches ∞ is ∞.
   i. f(n) = O(g(n)), because the limit as n approaches ∞ is 0.
   j. f(n) = O(g(n)), because the limit as n approaches ∞ is 0.
3)
   a. An efficient algorithm for finding the min and max of a list would be to compare the numbers in the list by pairs. Each pair would first be compared to each other, and then the smaller would be compared to the min and the larger to the max.
   Psuedocode:
```
Min = max = L[0]
for (itr = 1; itr < length, itr +=2):
   if list[itr] < list[itr+1]:
      if list[itr] < min, then min = list[itr]
      if list[itr+1]  > max, then max = list[itr+1]
   else:
      if list[itr+1] < min, then min = list[itr+1]
      if list[itr] > max, then max = list[itr]
```
   b. This algorithm performs at most 1.5n comparisons, because it makes 3 comparisons per pair of numbers (the first being list[itr] vs. list[itr+1], and the second and third being the comparisons of those to min and max).
   c. Initialization: min = max = 9
   Loop 1: 3 < 5 → 3 < min(9), and 5 < max(9) → min = 3, max = 9
   Loop 2: 10 > 1 → 1 < min(3), and 10 > max(9) → min = 1, max = 10
   Loop 3: 7 < 12 → 7 > min(1), and 12 > max(10) → min = 1, max = 12
   The total number of executions in this case is 9, which is less than 1.5*7.
4)
   a. **False:** For example, let $f_1(n) = x$, $f_2(n) = x^2$, and $g(n) = x^3$. In this case, $\lim_{n\to\infty}f_1(n)/g(n)$ and $\lim_{n\to\infty}f_2(n)/g(n)$ both equal 0, meaning that $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$, but $\lim_{n\to\infty}f_1(n)/f_2(n) = 0$, which means that $f_1(n) = O(f_2(n))$, not $\Theta(f_2(n))$.
   b. **True:**
   There exists constants c_1 and N_1, such that $0<f_1(n)\leq c_1g_1(n)$ for all $n\geq N_1$.
   There exists constants c_2 and N_2, such that $0<f_2(n)\leq c_2g_2(n)$ for all $n\geq N_2$.
   Let $G(n) = \max\{g_1(n), g_2(n)\}$, and let c = 2.
   $f_1(n)\leq G(n)$ for $n\geq N_1+N2$, and $f_2(n)\leq G(n)$ for $n\geq N_1+N_2$, so
   $0<f_1(n)+f_2(n)\leq G(n)+G(n)$, so
   $0<f_1(n)+f_2(n)\leq 2G(n)$, which means $0<f_1(n)+f_2(n)\leq cG(n)$.
   Therefore, $f_1(n)+f_2(n) = O(g(n)) = O(\max\{g_1(n), g_2(n)\})$.

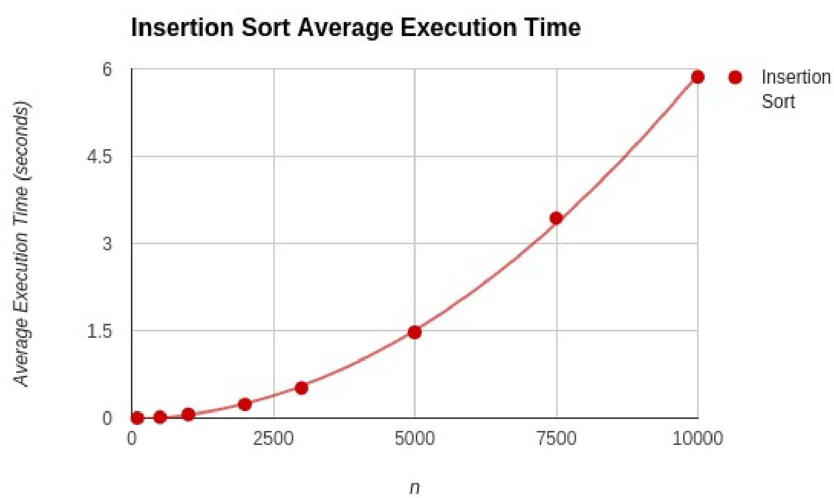
5)

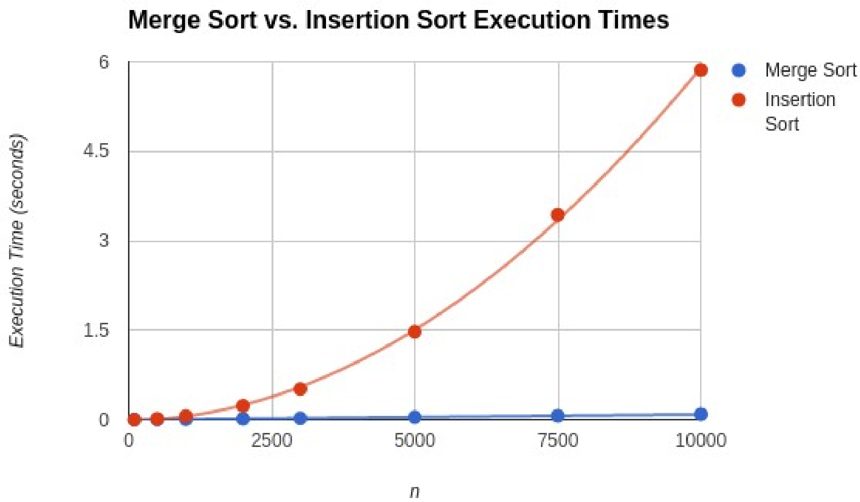
a. *See attached file, 'hw1p5.py' for source code.*

b. *Five repetitions were run for each value of n.

Average Execution Times* (sec)		
n	Merge Sort	Insertion Sort
100	0.0004762649536	0.0006517887115
500	0.003063964844	0.01423921585
1000	0.006812667847	0.06034593582
2000	0.01465492249	0.2324795723
3000	0.02322435379	0.5142115593
5000	0.04127230644	1.473435259
7500	0.06482887268	3.435245657
10000	0.09091959	5.861822796

c.





The individual plots are useful for visualizing the rates at which each algorithm's execution time increases with growing values of n, particularly the fact that merge sort increases fairly linearly, but overall, the combined plot is far more useful for looking at just how much faster than insertion sort merge sort becomes at higher values of n.

d. Merge sort displays a strong linear pattern, while insertion sort displays a quadratic growth pattern.
$T_{merge} = 9.1127924501471 \cdot 10^{-6}x - 0.0024911646568351$
$T_{insertion} = 5.741472705 \cdot 10^{-8}x^2 + 1.786022393 \cdot 10^{-5}x - 1.90567523 \cdot 10^{-2}$

e. My data matches up very well with the theoretical runtimes of O(n) for merge sort and $O(n^2)$ for insertion sort.