Ryan Vollmer - CS 325 - HW 1

1)
8n^2 = 64nlgn
8n = 64lgn # / n
n = 8lgn # / 8
n = 1.1 and 43.6

therefore with rounding,
2 < n < 43
in which 8n^2 is faster than 64nlgn

2)
a) big-o - n^(1/4) grows at a slower rate than n^(1/2)
b) omega - n grows at a faster rate than log(n)
c) theta - both have log growth rate
d) theta - both are n^2 growth rate
e) big-o - log(n) grows at a slower rate than sqrt(n),
therefore, n*log(n) will also grow at a slower rate than n*sqrt(n)
f) theta - both grow at exponential rate of n
g) big-o - 2^n grows at a slower rate than 2^(n+1)
h) big-o - 2^n grows at a slower rate than 2^(2^n)
i) big-o - 2^n grows at a slower rate than n!
j) big-o - log(n) grows at a slower rate than sqrt(n)

3)
a)
```
// Breaks all values into n/2 min and max
// then gets min of all mins,
// then gets max of all maxs,
// returns min and max
MIN_MAX(A)
  min = A[A.length-1], max = A[A.length-1]
  mins = [], maxs = []
  // deal with odd input length (1)
  if A.length is odd
    len = A.length - 1
  else
    len = A.length
  // sort into mins and maxs (n/2)
  for i=0 to len step 2
    if A[i] > A[i+1]
      add A[i+1] to mins
      add A[i] to maxs
    else
      add A[i] to mins
      add A[i+1] to maxs
  // get min from mins
  for i=0 to mins.length (n/2)
```

```
    if mins[i] < min
      min = mins[i]
  // get max from maxs (n/2)
  for i=0 to max.length
    if maxs[i] > max
      max = maxs[i]
  return (min,max)
```

b)

sort mins and max = n/2 comparisons
get min from mins = n/2 comparisons
get max from maxs = n/2 comparisons
total = 3(n/2) comparisons = 1.5n comparisons

c)

```
L=[9,3,5,10,1,7,12] #n=7
# 0 compares, 0 total
min=12, max=12
mins=[], maxs=[]
# 1 compare, 1 total
len=6
# first loop -> 3 compares, 4 total
i=0, mins=[3], maxs=[9]
i=2, mins=[3,5], maxs=[9,10]
i=4, mins=[3,5,1] maxs=[9,10,7]
# second loop -> 3 compares, 7 total
i=0, min=3
i=1, min=3
i=2, min=1
# third loop -> 3 compares, 10 total
i=0, min=12
i=1, min=12
i=2, min=12
# 0 compares, total 10 -> total(10) / n(7) = 1.42n
return min,max #1,12
```

4)

a)

To prove: if f1(n) = O(g(n)) and f2(n) = O(g(n)) then f1(n) = theta(f2(n))

1. By definition, f1(n) = O(g(n)) implies there exists some positive constants c1 and n0 such that 0 <= f1(n) <= c1*g(n) for all n >= n0.

2. Similarly, f2(n) = O(g(n)) implies there is some positive constants c2 and n1 such that 0 <= f2(n) <= c2*g(n) for all n >= n1.

3. By definition f1(n) = theta(f2(n)) implies there exists some constants c3,c4, and n2 such that 0 <= c3*f2(g(n)) <= f1(g(n)) <= c4*f2(g(n)) for all n >= n2.

By counter example, let f1(n) = n, f2(n) = n^2, and g(n) = n^2.
f1(n) =
O(n^2) holds true,
f2(n) = O(n^2) holds true,
f1(n) = theta(n^2) does not hold true therefore the implication is false.

b)
if f1(n) = O(g1(n)) and f2(n) = O(g2(n)) then f1(n) + f2(n) = O(max{g1(n), g2(n)})

1. By definition, f1(n) = O(g1(n)) implies there exists some positive constants c1 and n0 such that 0 <= f1(n) <= c1*g1(n) for all n >= n0.

2. Similarly, f2(n) = O(g2(n)) implies there is some positive constants c2 and n1 such that 0 <= f2(n) <= c2*g2(n) for all n >= n1.

In the case g1(n) >= g2(n), then:
c2*g2(n) <= c1*g1(n) for all n >= n0 and all n >= n1.
This implies g2(n) = O(g1(n)), and f1(n) + f2(n) = O(g1(n)) holds true.

In the case g2(n) >= g1(n), then:
c1*g1(n) <= c2*g2(n) for all n >= n0 and all n >= n1.
This implies g1(n) = O(g2(n)), and f1(n) + f2(n) = O(g2(n)) holds true.

Since both cases of f1(n) + f2(n) = O(max{g1(n), g2(n)}) hold true, the hypothesis is correct.

5)
a) see q5.py (mergeSort,insertionSort)
b) see q5.py (main,test)
c) see q5.xlsx (Average Case). I think the combined graph better represents the data on a grander scale/if comparing which algorithm is more efficient.
d) nlgn best represents the runtime of mergeSort
n^2 best represents the runtime of insertionSort
see q5.xlsx for equations of trend.
e) they ran fairly accurately, with R^2 being 0.99 in both cases. The graph trend lines look fairly close to what they should be.

extra) see q5.xlxs (worse case/best case).
Worst case:
insertionSort = n^2
mergeSort = nlgn
I think the comparison graph best displays the data, as it shows the large difference in growth
Best Case:
insertionSort = n
mergeSort = nlgn
I think the comparison graph best displays the data, as it shows the large difference in growth