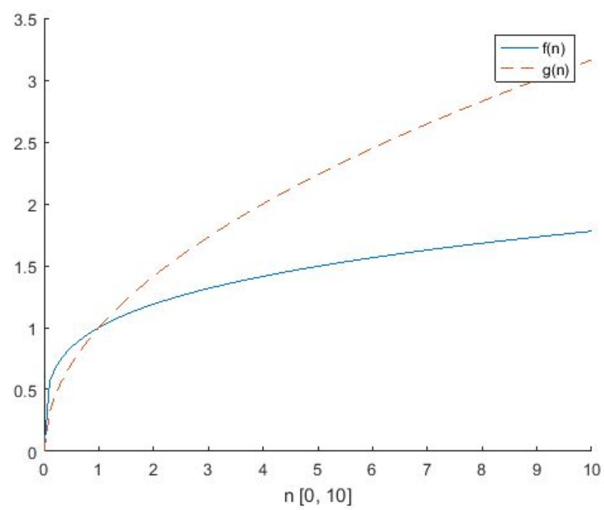
Adina Edwards
CS 325
1 Oct. 2017

HW 1

1. Describe a $\Theta(n\ logn)$ time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Explain why the running time is $\Theta(n\ logn)$.

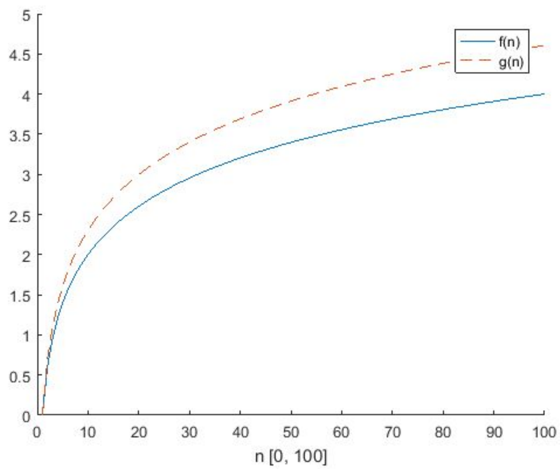
   Finding a specific value or two in a set is far easier to do once that set is sorted. Merge sort will give us a sorted set in $O(n\ logn)$ time.
   Once the list is sorted, we can start searching for two values that add up to x. We can do this using binary search, in which we can ignore any part of the sorted list that is larger than x, resulting in $O(n)$ time instead of the normal $O(logn)$ time.
   The total time for merge sort and binary search would be $O(n\ logn) + O(n)$. However, asymptotic analysis is mostly concerned with the dominating factor as this will overwhelm the less significant factors for large values of n. This results in the overall time being defined as $O(n\ logn)$.

2. For each of the following pairs of functions, either $f(n)$ is $O(g(n))$, $f(n)$ is $\Omega(g(n))$, or $f(n)$ is $\theta(g(n))$. Determine which relationship is correct and explain.
   a. $f(n) = n^{0.25}$ $\qquad g(n) = n^{0.5}$



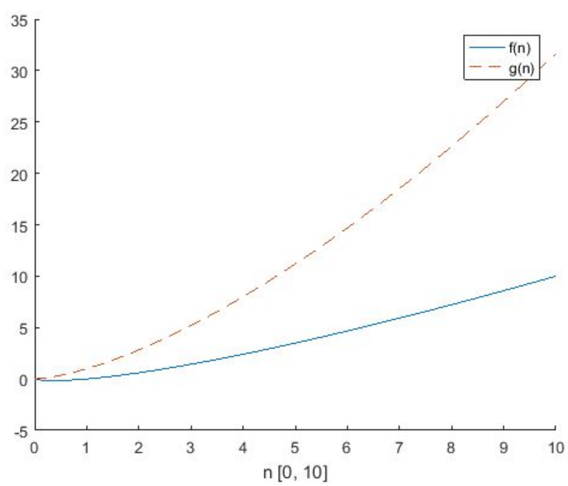
   $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^{0.25}}{n^{0.5}} = 0$ which implies that $0 \le f(n) \le g(n)$. The graph above shows $1 * f(n)$ in blue and $1 * g(n)$ in orange. There exist positive constants c_1 and n_o such that $0 \le f(n) \le c_1 g(n)$ when $n \ge n_o$, which means that $f(n) \in O(g(n))$.

b. $f(n) = log\ n^2$ $g(n) = ln\ n$



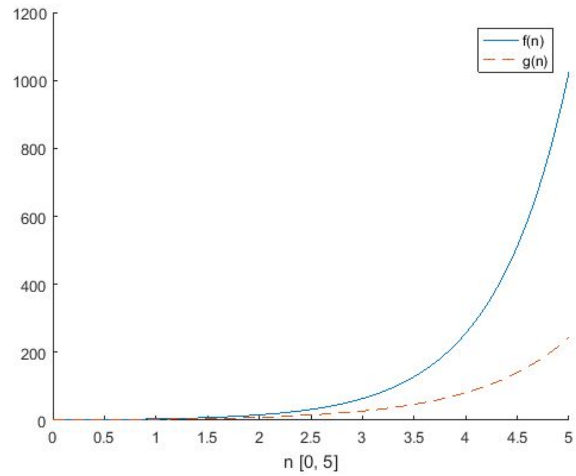
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log\ n^2}{ln\ n} = \lim_{n\to\infty} \frac{(ln\ n^2)/(ln\ 10)}{ln\ n} = \lim_{n\to\infty} \frac{2(ln\ n)}{(ln\ n)(ln\ 10)} = \frac{2}{ln\ 10}$ There exist positive constants c_1, c_2 and n_o such that $0 \le c_2 g(n) \le f(n) \le c_1 g(n)$ when $n \ge n_o$, which means that $f(n) \in \theta(g(n))$.

c. $f(n) = nlog\ n$ $g(n) = n\sqrt{n}$



$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n\ log\ n}{n\sqrt{n}} = 0$ which implies that $0 \le f(n) \le g(n)$. There exist positive constants c_1 and n_o such that $0 \le f(n) \le c_1 g(n)$ when $n \ge n_o$, which means that $f(n) \in O(g(n))$.

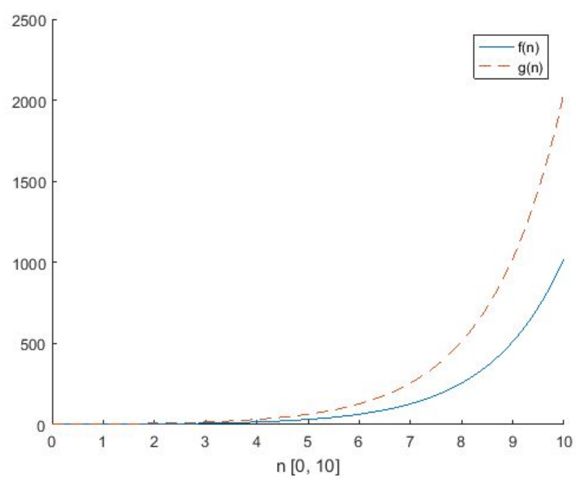
d. $f(n) = 4^n$ $g(n) = 3^n$



$f(n) \in \Omega(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4^n}{3^n} = \infty$ which implies that $0 \leq g(n) \leq f(n)$. There exist positive constants c_1 and n_o such that $0 \leq c_1 g(n) \leq f(n)$ for all $n \geq n_o$. By definition, $f(n) \in \Omega(g(n))$.

e. $f(n) = 2^n$ $g(n) = 2^{n+1}$



$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{2^{n+1}} = 0$ which implies that $0 \leq f(n) \leq g(n)$. There exist positive constants c_1 and n_o such that $0 \leq f(n) \leq c_1 g(n)$ when $n \geq n_o$, which means that $f(n) \in O(g(n))$.

f. $f(n) = 2^n$ $g(n) = n!$

$f(n) \in O(g(n))$

This can be demonstrated by showing that there exist positive constants c_1 and n_o such that $0 \leq f(n) \leq c_1 g(n)$ for all $n \geq n_o$. In this case, n_0 is between 3 and 4; for all $n \geq 4$, $f(n) \leq c_1 g(n)$. By definition, $f(n) \in O(g(n))$.

3. Let f_1 and f_2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counterexample.

   a. If $f_1(n) \in O(g(n))$ and $f_2(n) \in O(g(n))$ then $f_1(n) + f_2(n) \in O(g(n))$.
      According to the definition of O, there exist positive constants c_1 *and* n_o such that $0 \leq f_1(n) \leq c_1 g(n)$ for all $n \geq n_o$ and there exist positive constants c_2 *and* n_1 such that $0 \leq f_2(n) \leq c_2 g(n)$ for all $n \geq n_1$.

      Adding the two functions together gives us:

      $$0 + 0 \leq f_1(n) + f_2(n) \leq c_1 g(n) + c_2 g(n) \text{ for all } n \geq max(n_o, n_1)$$

      $$0 \leq f_1(n) + f_2(n) \leq (c_1 + c_2) g(n) \text{ for all } n \geq max(n_o, n_1)$$

      In O notation, we can ignore the constants as we are concerned with the asymptotic bounds of the function as n gets significantly large. Since our statement is already in the form of the definition of O, we can say $f_1(n) + f_2(n) \in O(g(n))$ when $f_1(n) \in O(g(n))$ and $f_2(n) \in O(g(n))$.

   b. If $f(n) \in O(g_1(n))$ and $f(n) \in O(g_2(n))$ then $g_1(n) \in \Theta(g_2(n))$.
      According to the definition of Θ, there exist positive constants c_1, c_2 and n_0 such that $c_1 g_2(n) \leq g_1(n) \leq c_2 g_2(n)$ for all $n \geq n_o$.

      However, it could be the case that $f(n) \in O(n)$ and $f(n) \in O(n^2)$. If this were true, then $c_1 n^2 \leq n \leq c_2 n^2$ for all $n \geq n_o$ would have to be true. $c_1 n^2 \leq n$ is false when $n > 1$.

      $4(3^2) \leq 3$ when $n = 3$, but 36 *is not* ≤ 3

      $0.5(3^2) \leq 3$ when $n = 3$, but 4.5 *is not* ≤ 3

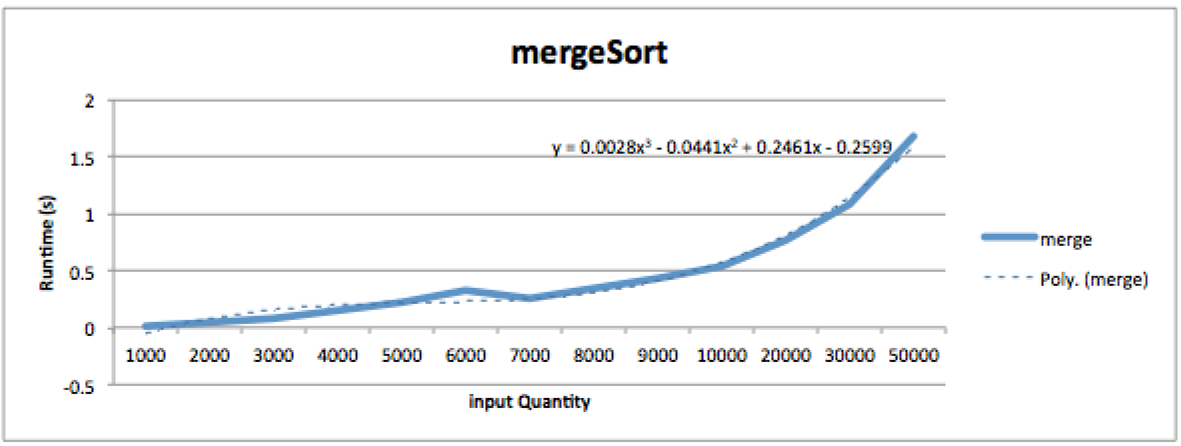
      The statement is false.

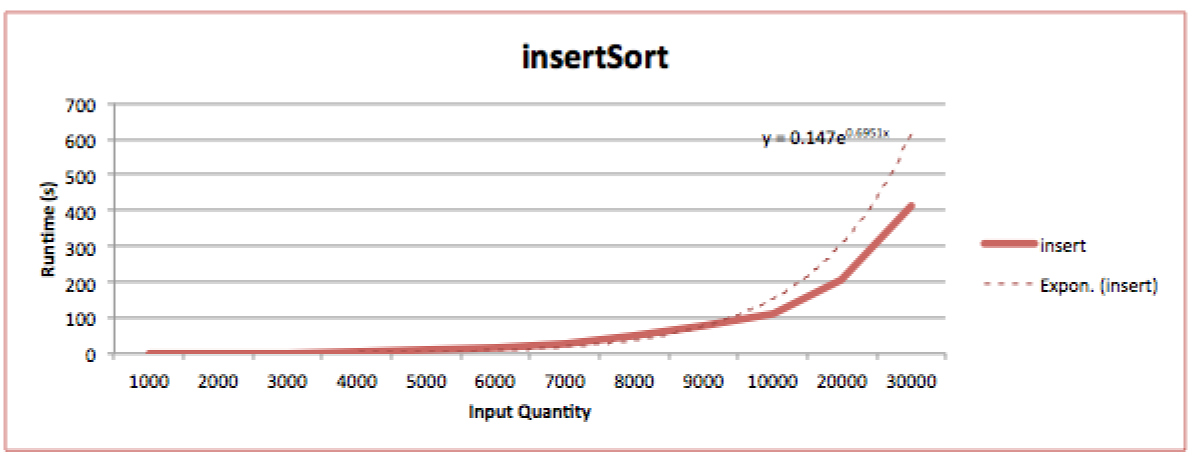
5. a) See python files mergesortTimedBW.py and insertsortTimedBW.py.
b)

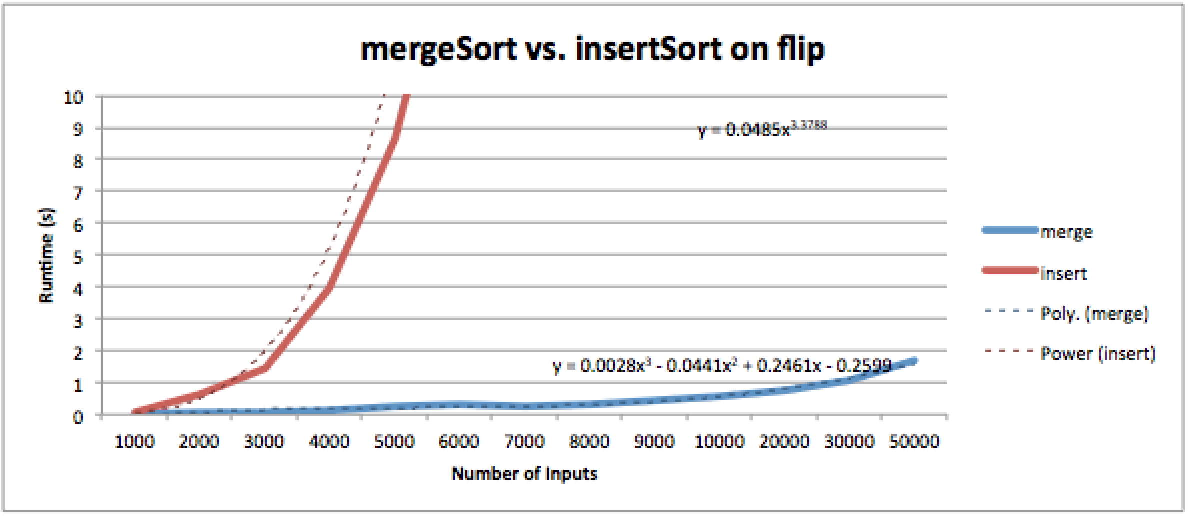
mergeSort		insertSort	
Input Size	Runtime	Input Size	Runtime
1000	0.011577845	1000	0.070154905
2000	0.039311886	2000	0.615178108
3000	0.084277868	3000	1.45300293
4000	0.146716118	4000	3.955150843

5000	0.227011919	5000	8.644432068
6000	0.327566147	6000	16.42769194
7000	0.258759975	7000	29.03759909
8000	0.34069705	8000	48.49501109
9000	0.434635878	9000	75.22172999
10000	0.53847003	10000	112.1867239
20000	0.756613016	20000	208.300894
30000	1.084466219	30000	413.1043992
50000	1.668538094		

c)







d) InsertSort looks to be exponential. When the input gets large, it quickly begins to take a really long time to sort. This makes sense that it would grow quickly as the input grows, given that each data point is run through twice, but this does not explain the exponential appearance.

MergeSort seems to be a polynomial on the order of $O(n^3)$. This is certainly larger than the expected $O(n\ logn)$.

It is possible that using the higher-level language python as well as running the program on flip (a shared server, yet only so many cores on a Sunday) has caused both algorithms to run more slowly than I would have thought.

I also noticed that the runtime of any n_x, where x is any of the Input Quantity values, is significantly slower when the program cycles through a large quantity of tests versus running a test specifically for any n_x. If I ran a set of tests [1000, 2000, 3000, 4000, 5000] then the runtime for 4000 inputs would be significantly longer than if I ran only the test with 4000 inputs. I suspect it has something to do with how the cache works.

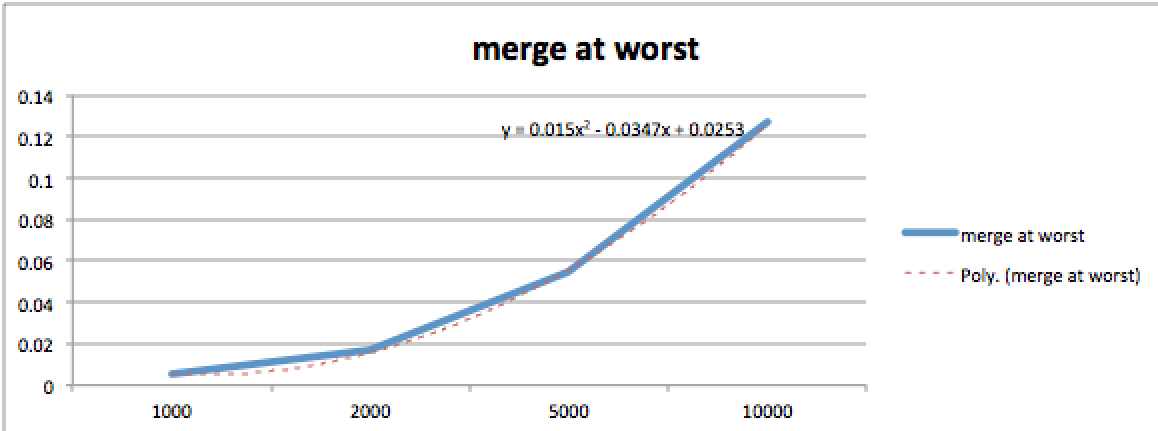
Extra Credit HW 1

Worst case for both algorithms is reverse sorted input. Best case is already sorted input.

mergeSort Worst appeared to run in $O(n^2)$ time.

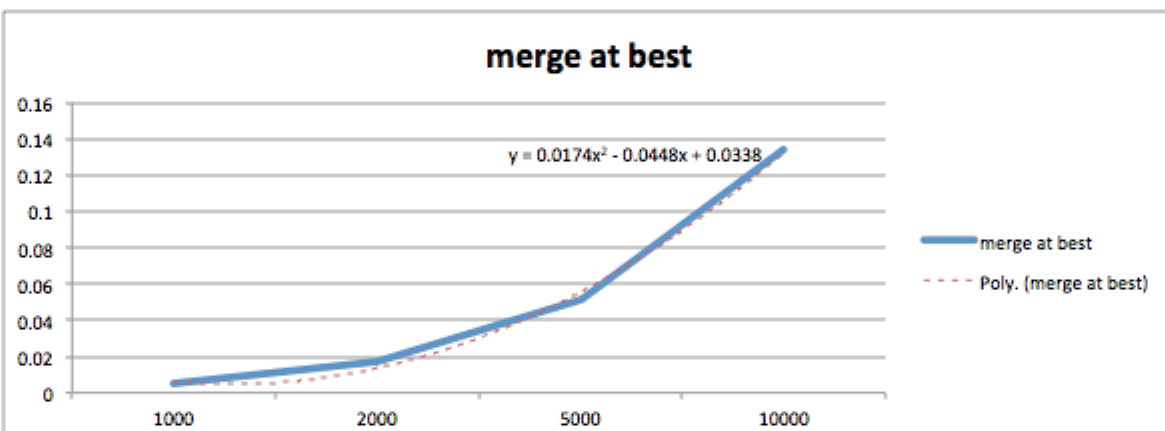
Input Size	Runtime
1000	0.005033016
2000	0.017289877
5000	0.054410934

10000	0.12659502



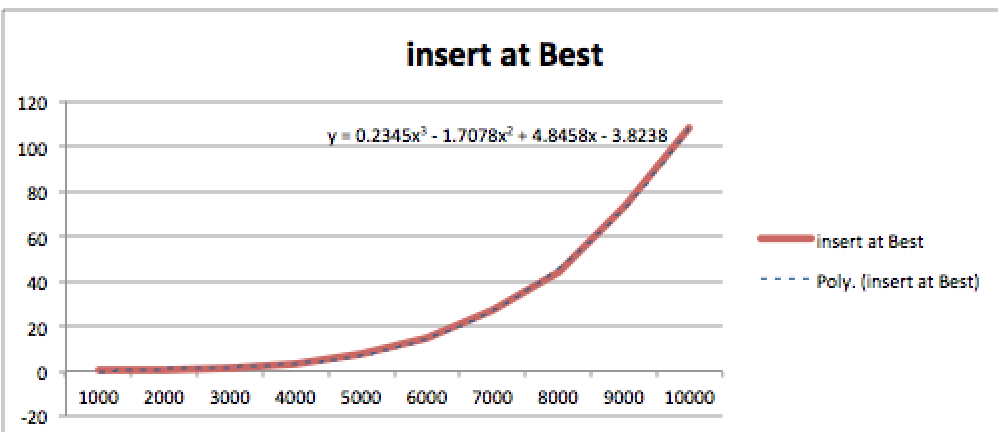
mergeSort Best appeared to run in $O(n^2)$ time.

Input Size	Runtime
1000	0.005034208
2000	0.01757884
5000	0.051733017
10000	0.133704901



insertSort Best appeared to run in $O(n^3)$ time.

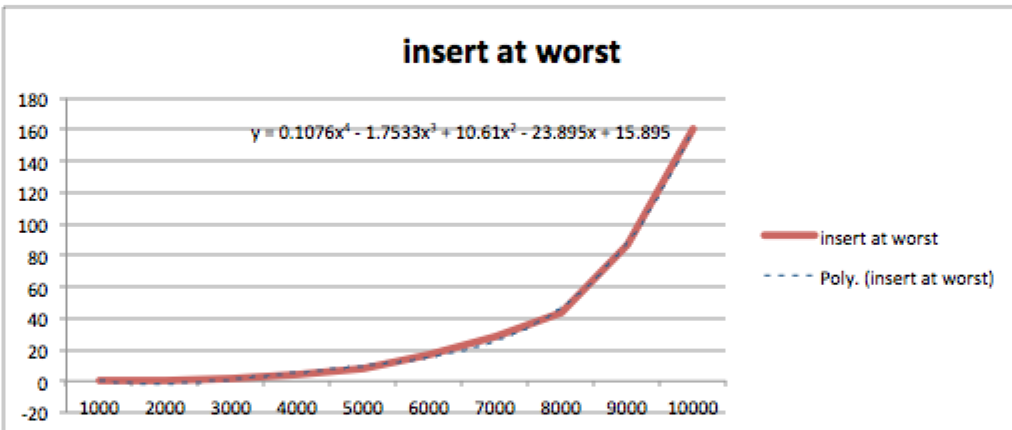
Input Size	Runtime
1000	0.032812119
2000	0.302526951
3000	1.230522871
4000	3.343257904
5000	7.466012001
6000	14.91609597
7000	27.56055188
8000	43.74303794
9000	73.11539197
10000	108.4792869



insertSort Worst appeared to run in $O(n^4)$ time.

Input Size	Runtime
1000	0.037779808
2000	0.305724144
3000	1.245493889

4000	3.599560022
5000	8.074874878
6000	16.45708489
7000	28.852036
8000	44.0317359
9000	87.31139112
10000	160.8897531



This all fits what I would expect. At best and worst, mergesort will still perform about the same as it has to cycle through the same quantity of data regardless of how well sorted it is. At worst, insertsort will perform worse than at best because it shifts data inverse to how sorted that data is.