Arthur Liou
CS325

HW 1

1) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8n2 steps, while merge sort runs in 64nlgn steps. For what values of n does insertion sort run faster than merge sort?

Note: lg n is log "base 2" of n or $\log 2n$. There is a review of logarithm definitions on page 56. For most calculators you would use the change of base theorem to numerically calculate lgn.

Using less than because (faster = less time)
8n2 < 64nlgn
N^2 < 8nlgn
N < 8lgn

Solution is: n > 0, and n < (8 log(n)) / (log(2))
So when (8 log(n)) / (log(2)) > n > 0.

2) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.
I find this question to be ambiguously questionable. Are we looking at the strick 1-1, or are we comparing the worst-case time time of f and g? I'm going with #2 as it is an analysis of algorithms class, not a graphing class.
a. f(n) = n^0.25; g(n) = n^0.5
f(n) = Θ(g(n); f and g run times are part of the same polynomial big O classes
b. f(n) = n; g(n) = log^2n
f(n) is Ω(g(n)); g is in the logarithmic (log n) class, which is faster than linear time, so Ω
c. f(n) = log n; g(n) = ln n
f(n) = Θ(g(n)); ln n is the same as log n. Thus asymptotically equal
d. f(n) = 1000n^2; g(n) = 0.0002n^2 – 1000n
f(n) = Θ(g(n)); We ignore the constants and focus on the first polynomial n^2. We see that both functions are quadratics and thus asymptotically equal
e. f(n) = nlog n; g(n) =n√n
f(n) is O(g(n); refactoring g would get n^1.5, polynomial (or somewhere between linear / quadratic. "Refactoring" both by dividing by n would result in f(n) = log n & g(n) = n^0.5. As square root of n takes longer than log n, f < g.
f. f(n) = en; g(n) = 3n
f(n) = Θ(g(n); e and 3 are constants. When analyzing the runtime of both functions, we find that the runtime of both functions are linear O(n).
g. f(n) = 2n; g(n) = 2n+1
f(n) = Θ(g(n); While technically 2n+1 will always be greater than 2n, we drop constants (the 2x and the + 1) when comparing functions and the comparison would be n vs n.

h. f(n) = 2n; g(n) =22n

f(n) = Θ(g(n); While technically 22n will always be greater than 2n, we drop constants when comparing functions and the comparison would be n vs n.

i. f(n) = 2n; g(n) = n!

f(n) is O(g(n); g will take the most time, so thus is O / f less than g

j. f(n) = lgn; g(n) = √n

f(n) = O(g(n); square root of n is n^0.5, which is greater than logarithmic. Thus, f < g / O

3) Let f1 and f2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If f1(n) = Θ(g(n)) and f2(n) = Θ(g(n)) then f1(n)= Θ (f2(n)).

Counter Example: Let f1(n) = n, g(n) = n^2. F2(n) = n^2

Looking at the Reflexivity property, we can look at this for O and this problem is same as a lecture example, but in Theta

b. If f1(n) = O(g1(n)) and f2(n) = O(g2(n)) then f1(n)+ f2(n)= Θ(g1(n) + g2(n))

Start: f1(n)+ f2(n)= Θ(g1(n) + g2(n))

Then: O(g1(n)) + O(g2(n)) = Θ(g1(n) + g2(n))

Counter Example: Θ (g1(n)) + Θ (g2(n)) = Θ(g1(n) + g2(n))

= g1(n) = n^2 & g2(n) = n^2

= Θ (n^2) + Θ(n^2) = Θ(n^2 + n^2)

= n^2 + n^2 = Θ(2n^2)

= 2n^2 = n^2

When we calculate the addition of g1 and g2 on the right, we will always ignore the constants when calculating runtime. But when we calculate on the left, we calculate the combination of the two runtimes, which will lead to a constant (or inequality if g1 and g2 are not same) that is not present on the right side.

4) See Attached Code

5) Merge Sort vs Insertion Sort Running time analysis

a) Modify code- Now that you have verified that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for n = 5000, 10000, 15000, 20,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. Output the array size n and time to the terminal. Name these new programs insertTime and mergeTime.

See attached code. Currently n is 10000k. I manually manipulated the number n for each run time that I collected

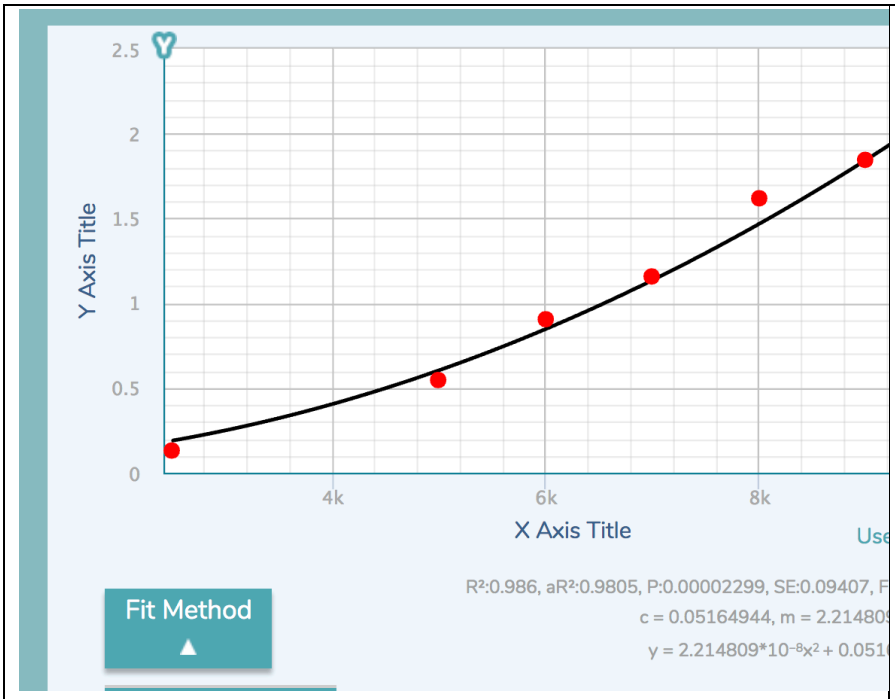
b) Collect running times - Collect your timing data on the engineering server. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of

the same algorithm you may want to take the average time of several runs for each value of n. Create a table of running times for each algorithm.

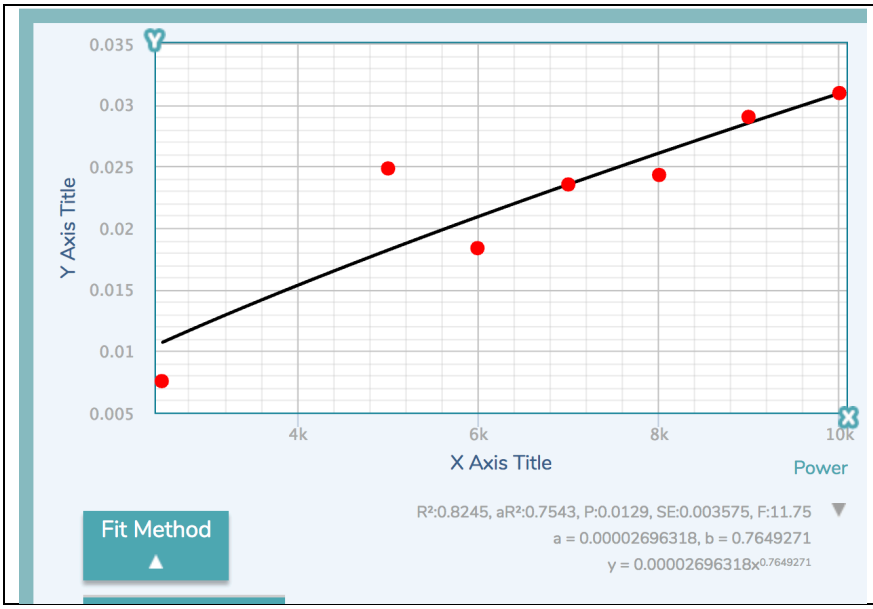
Array Size	MergeSort (runtime in seconds)	InsertionSort (runtime in seconds)
2500	0.00755405426025	0.130571842194
5000	0.0248291492462	0.54644203186
6000	0.0183548927307	0.905103206635
7000	0.0235290527344	1.15743088722
8000	0.0243008136749	1.61838698387
9000	0.0290191173553	1.8448369503
10000 (Max)	0.0309538841248	2.15977096558

Process: Used 7 array sizes and ran them both through MergeSort and InsertionSort to get their respective average runtimes.

c) Plot data and fit a curve - For each algorithm plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. What type of curve best fits each data set? Give the equation of the curves that best "fits" the data and draw that curves on the graphs.

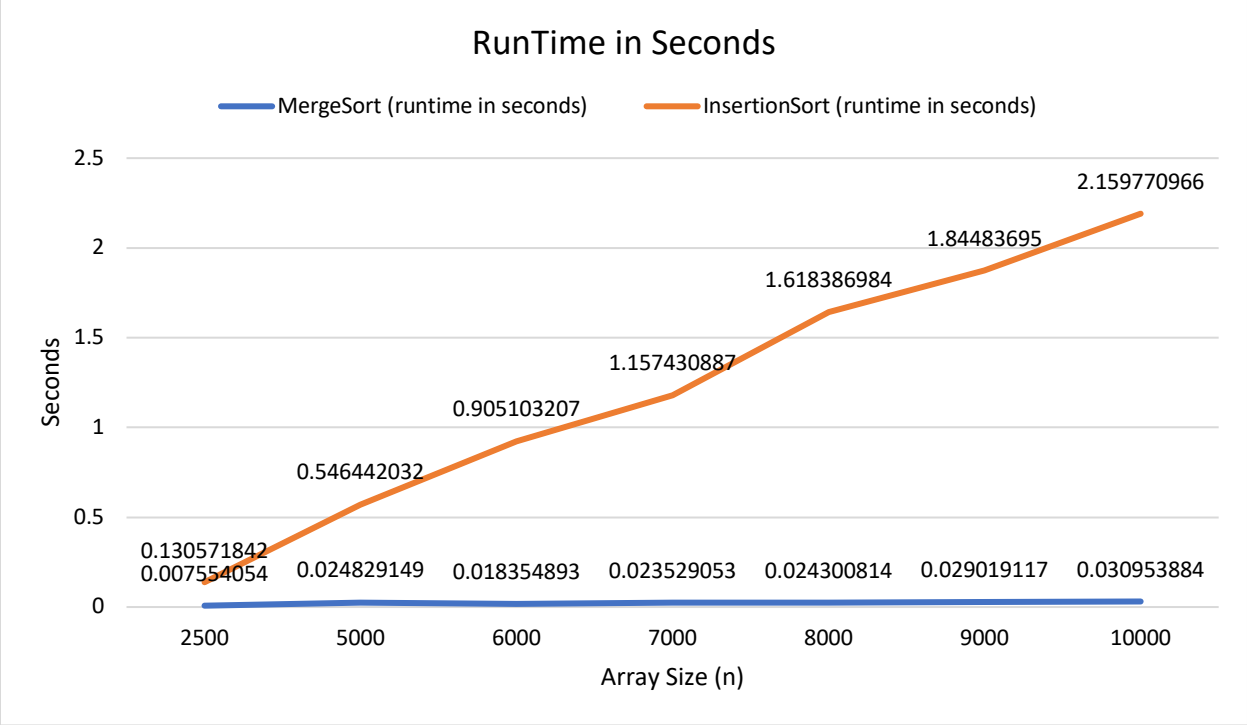


InsertionSort
Y = 2.21x4809*10^-8 * x^2 + 0.05

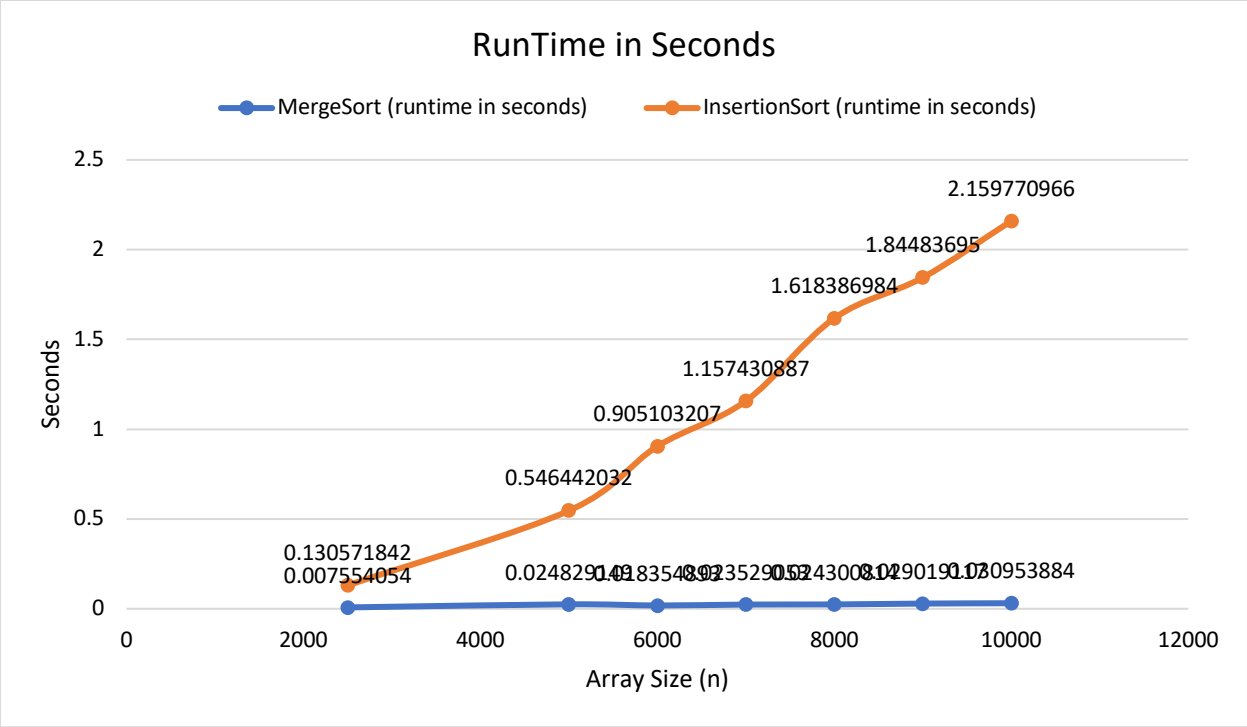


MergeSort
Y = 0.000026969318x^0.7649271

d) Combine - Plot the data from both algorithms together on a combined graph. If the scales are different you may want to use a log-log plot.



For continuity, I converted it into a log-log plot below.



e) Comparison - Compare your experimental running times to the theoretical running times of the algorithms? Remember, the experimental running times were the "average case" since the input arrays contained random integers.

MergeSort
Average Sort: Θ(n log(n))
0.000026969318x^0.7649271
In the scaled down version of my graph in C, I can see the n log n when it's starting to curve upward and reach a limit. This is apparent in a bigger picture graph in D, where the time "seems" more linear compared to InsertionSort and does not get very large

InsertionSort
Average Sort: Θ(n^2)
y = 2.21x4809*10^-8 * x^2 + 0.05
In both the scaled down version and in the bigger picture version, I can see the runtime exponentially increase as it goes up. While it seems more linear at the moment, I'm considering that I'm using relatively small sample sizes, and at larger array sizes (think 100k+, 1MM+, 1B+) the curve of the exponential function will be apparent.

EXTRA CREDIT: A Tale of Two Algorithms:
Generate best case and worst case inputs for the two algorithms and repeat the analysis in parts b) to e) above. To receive credit you must discuss how you generated your inputs and your results. **Submit your code to TEACH in a zip file with the code from Problem 4 & 5.**

Extra Credit

I generated my inputs by figuring out what would constitute a best case and worse case given how each algorithm sorts. For the most part, best case is when the array is already sorted & worst case is when you have to sort the most.

b) Best and worse case inputs

Best Case Inputs:

Merge Sort: array is already sorted, so O(n) comparisons. [0,1,2,3,4,5,6,7]

Insertion Sort: array is already sorted, so O(n) comparisons. [0,1,2,3,4,5,6,7]

Worse Case Inputs:

Merge Sort: [4,0,6,2,5,1,7,3], you have to sort each time

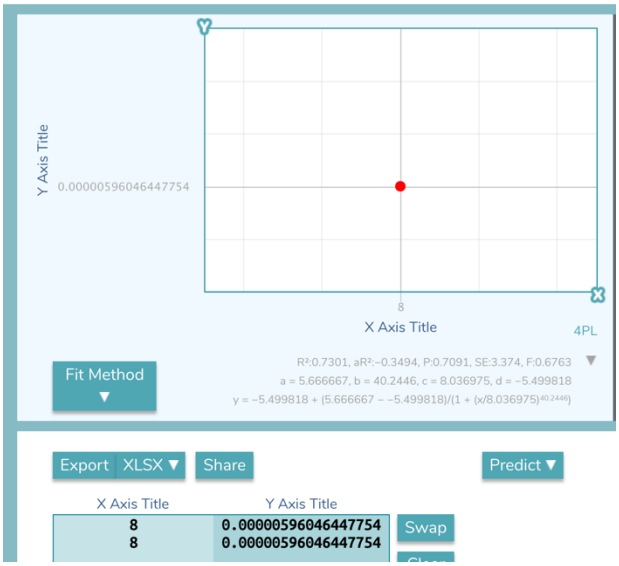
Insertion Sort: The simplest worst case input is an array sorted in reverse order. [7,6,5,4,3,2,1,0]

Array Size	MergeSort (runtime in seconds)	InsertionSort (runtime in seconds)
[0,1,2,3,4,5,6,7]	1.4066696167e-05	5.96046447754e-06
[4,0,6,2,5,1,7,3]	1.21593475342e-05	Not Applicable
[7,6,5,4,3,2,1,0]	Not Applicable	5.96046447754e-06

c)

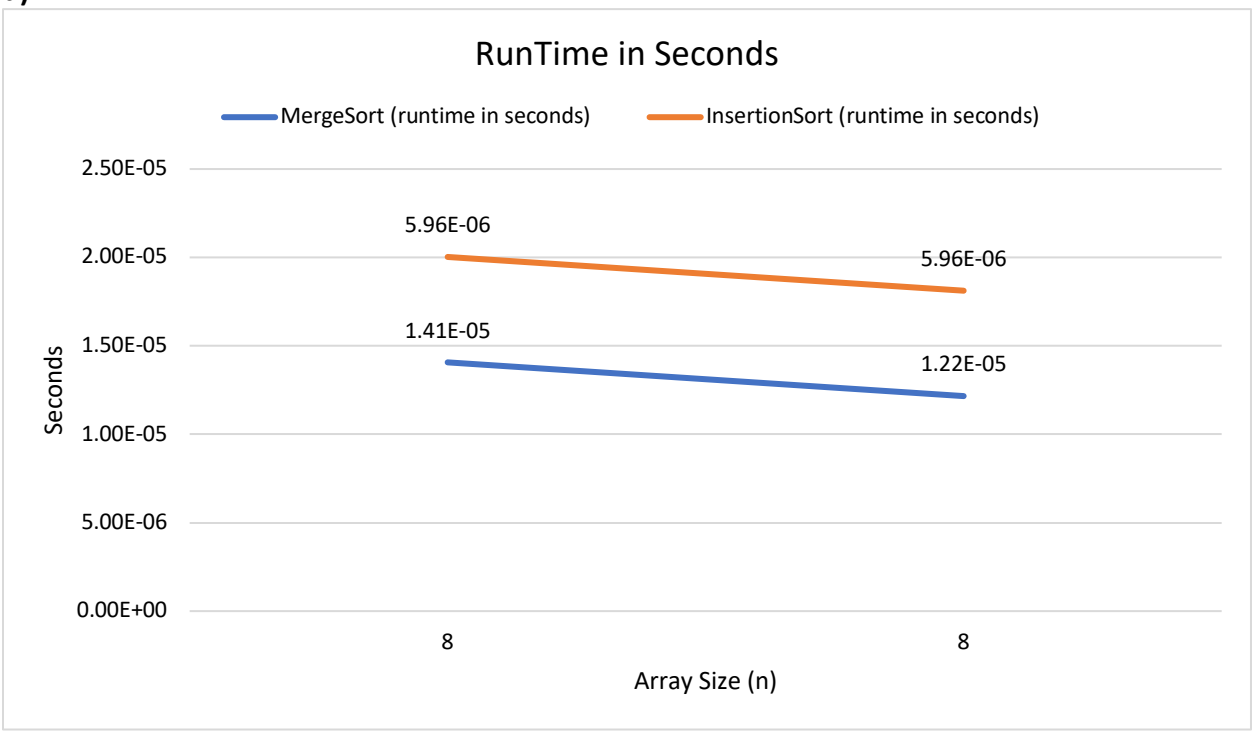
MergeSort

Y = -5.499818 + (5.6667 - -5.499818)/(1+ (x/8.036975)^40.2446)



y = -5.499818 + (5.666667 - -5.499818)/(1 + (x/8.036975)^40.2446)

d)



e)

<u>MergeSort</u>

Worst Case: O(n log(n))

Average Sort: Θ(n log(n))

Best Case: O(n)

I thought the theoretical runtime vs the best and worst run times was interesting because I got I got a slightly shorter runtime for worstcase. This was likely due to a small sample size (I didn't use numbers like 10k plus) as they would be difficult to manually calculate and write them out).

However, if we theoretically look at the runtimes where in best case, no sorting is done (it just iterates through the array), and in the worst case, where every single check requires a switch. The theoretical run times would relatively match the real runtimes since log n (removing n since n is constant and small) is faster than O(n)

InsertionSort
Worst Case: Θ(n^2)
Average Sort: Θ(n^2)
Best Case: O(n)

I thought the theoretical runtime vs the best and worst run times was interesting because I got the same time for both. This was likely due to a small sample size (I didn't use numbers like 10k plus) as they would be difficult to manually calculate and write them out). However, if we theoretically look at the runtimes where in best case, no sorting is done (it just iterates through the array), and in the worst case, where every single check requires a switch. The theoretical run times would match the real runtimes.