Tristan Santiago

CS325_400

September 30, 2018

## Question 1

Problem: Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in 64n lg n steps. For what values of n does insertion sort run faster than merge sort?

Insertion sort = $8n^2$

Merge sort = 64n $\log n$

Objective: To determine the values of *n* where insertion sort is faster than merge sort.

To start, I simplified $8n^2$ = 64n $\log n$ and came up with: n = 8 $\log n$. This allows me to plug in values for n to determine where n is less than 8 $\log n$.

To test the values, I used the Windows calculator to find the value of 8 $\log n$ where n is the value I want to test. I created a table to show the results:

I used the following formula to generate these results with the Windows calculator: $\log_2(n) = \ln(n)/\ln(2)$ * 8.

When n = 2, for example, 8 $\log n$ = 8.

$\log_2(n) = \ln(2)/\ln(2)$ * 8

= 0.69 * 0.69 * 8

= 8

So when n = 2, 8 $\log n$ =0

When n = 4:

$\log_2(n) = \ln(4)/\ln(2)$ * 8 = 8

= 1.39 / 0.69 * 8

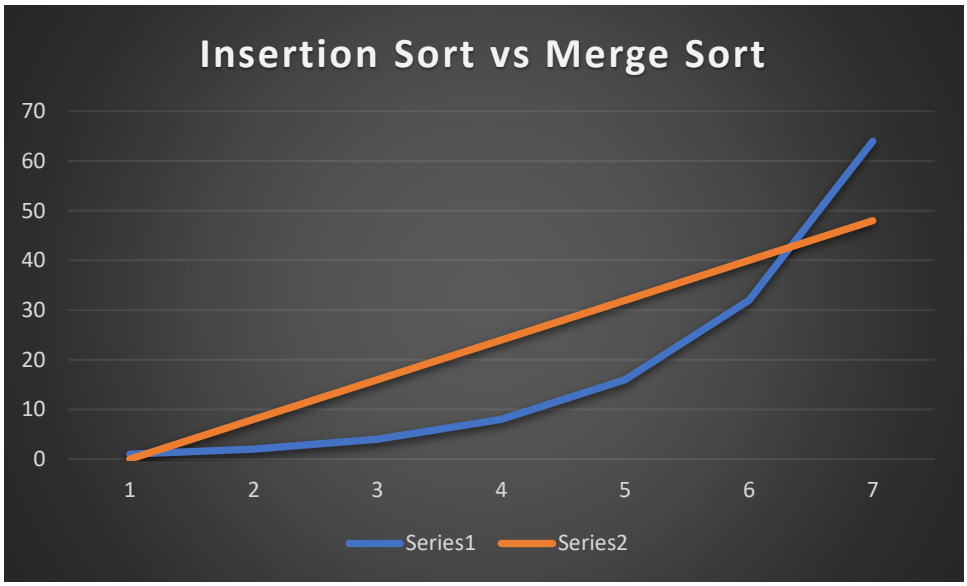
= 2 * 8

= 16

| n | 8 $\log n$ |
|---|---|
| 1 | 0 |
| 2 | 8 |
| 4 | 16 |
| 8 | 24 |

| | |
|---|---|
| 16 | 32 |
| 32 | 40 |
| 64 | 48 |

When testing 64, I noticed that merge sort was operating faster than insertion sort and the only way I could think of to figure out at what value merge sort started become faster was to test values between 32 and 64. Starting at 33, I worked my way up until I came to around 43.5, which is where I started to notice that merge sort started running faster. Therefore, my answer is 1 < n < 44 (since the number needs to be an integer).




## Question 2

Using Wolfram Alpha to calculate the limit, I received the following results:

A. f(n) is O(g(n); (Big-O)

Input: n^0.25 / n^0.5

Result:

Limit:

$$\lim_{n\to\pm\infty} \frac{1}{n^{0.25}} = 0 \approx 0$$

B. f(n) is Ω(g(n); Omega
Input: (n / log^2(n))

Limit:

$$\lim_{x \to 0} \log^2(x) = \infty$$

$$\lim_{x \to -\infty} \log^2(x) = \infty$$

$$\lim_{x \to \infty} \log^2(x) = \infty$$

C. f(n) is Θ(g(n); Theta
Input: limx->inf((log(x)/ln(x)))
Result:

Limit:

$$\lim_{x \to \infty} \frac{\log(x)}{\log(x)} = 1$$

D. f(n) is Θ(g(n)); Theta
Input: limx->inf((1000x^2)) / ((0.0002x^2)-1000))
Result:

Limit:

$$\lim_{x \to \infty} \frac{1000\,x^2}{0.0002\,x^2 - 1000} = 5\,000\,000$$

E. f(n) = O(g(n); Big-O
Input: limx->inf(nlog x / sqrt(x))
Result:

Limit:

$$\lim_{x \to \infty} \frac{n \log(x)}{\sqrt{x}} = 0$$

F. f(n) is O(g(n); Big-O
Input: (e^n/3^n)
Result:

Limit:

$$\lim_{n\to\infty}\left(\frac{e}{3}\right)^n = 0$$

G. f(n) is Θ(g(n)); Theta
Input: limx->inf((2^x/2^(x+1))
Result:

Limit:

$$\lim_{x\to\infty}\frac{2^x}{2^{x+1}} = \frac{1}{2}$$

H. f(n) is O(g(n); Big-O
Input: limx->inf(2^x/2^2^x)
Result:

Limit:

$$\lim_{x\to\infty}\frac{2^x}{2^{2^x}} = 0$$

I. f(n) is O(g(n); Big-O
Input: limx->inf(2^x/x!)
Result:

Limit:

$$\lim_{x\to\infty}\frac{2^x}{x!} = 0$$

J. f(n) is O(g(n); Big-O
Input: limx->inf(lgn/sqrt(x))
Result:

Limit:

$$\lim_{x\to\infty}\frac{\log_{10}(n)}{\sqrt{x}} = 0$$

## Question 3

a. If f1(n) = theta(g(n)) and f2(n) = theta(g(n)) then f1(n) = theta(f2(n))
By definition, f(n) = theta(g(n)) means that there exist positive constants c1, c2, and k, such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ k. The values of c1, c2, and k must be fixed for the function f and must not depend on n.
Show f1(n) = theta(f2(n))

Based on the definition above:
f1(n) can be written as f1(n) = c1.g(n) where c1 > 0

f2(n) can be written as f2(n) = c2.g(n) where c2 > 0

Using these two equations, we can now find the relationship between f1(n) and f2(n).

Since f1(n) = c1.g(n), if we substitute the value of g(n) from the second equation, we get:
f1(n) = c1.f2(n) / c2
After combining, we get:
f1(n) = (c1/c2).f2(n)
If we let (c1/c2) = c3
Then we assume:
f1(n) = c3.f2(n) (assuming c3 is a constant)
Which is the definition of Theta, therefore:
**f1(n) = Theta (f2(n)) is TRUE.**

b. If f1(n) = Big-O(g1(n)) and f2(n) = Big-O(g2(n)) then f1(n)+f2(n) = theta(g1(n) + g2(n))

Given f1(n) = O(g1(n)), f2(n) = O(g2(n))

By definition of Big-O, if f(n) and g(n) be two functions such that f(n) <= c.g(n) where c > 0 , n = n0, n0 > = 1, then we say that f(n) = O(g(n))
So using what's given and the definition of Big-O, we can assume:
f1(n) <=c1.g1(n) and f2(n) <=c2.g2(n)
By combining 1 and 2, we assume this to be true:
f1(n) + f2(n) <= c1.g1(n) + c2.g2(n)
When we combine the constants:
f1(n) + f2(n) <=c3(g1(n) + g2(n)) (We assume the constant here is c3)
The definition above matches that of Big-O f1(n) + f2(n) = O(g1(n) + g2(n))
f1(n) + f2(n) <=c3(g1(n) + g2(n)) = f1(n) + f2(n) = O(g1(n) + g2(n))
However, the reverse is not true.
f1(n) + f2(n) >= c4(g1(n) + g2(n)) is not true.
Let's suppose f1(n) = n , f2(n) = n2, g1(n) = n3, and g2(n) = n4, which would match the definition of Big Theta f1(n) <= c1g1(n) and f2(n) <= c2g2(n)

Yet the reverse (f1(n) >=c1.g1(n) and f2(n) >=g2(n)) is not true.

In order for functions to be Theta, the functions must be asymptotically equal to one another. In this case, the function on right side (g(n)) must either be equal to

or greater than the functions on left side(f(n)) and the function on the left side must never be greater than right side function. Therefore:
f1(n) + f2(n) = Theta (g1(n) + g2(n)) is NOT TRUE

## Question 5

As per the instructions, part A has been included in the zipped folder to be submitted via TEACH.

Part B, C, and D can be found in the accompanying PDF (imported from Excel).

Part E

When comparing the experimental running times of my Insert Sort and Merge Sort algorithms to those of their respective theoretical running times, I was not surprised at all. The algorithms performed exactly as expected and in line with their average performance.

The running times of both algorithms clearly increased with the number of inputs for each test, though those of the Insertion Sort algorithm grew much quicker and a much higher rate than those of the Merge Sort algorithm. The results were truly astonishing and as I was testing higher inputs, I could actually feel the difference in run time while running the Insertion Sort algorithm, whereas running the Merge Sort algorithm returned the results almost instantly.

I also ran tests on smaller inputs, though those results are not shown in the graphed data. It is a well-known fact that Insert Sort runs faster than Merge Sort with smaller sets of data (less than approximately 43.5), but for the purposes of this assignment, I was more concerned with speed and growth over larger sets of data.