Evan Sturtevant, Emma Pinto
Algorithms and Analysis
Writing Assignment 02
October 13, 2021

Introduction

The goal of this project was to experimentally test the runtime of various sorting algorithms. We did this for two theoretically $\Theta(n^2)$ algorithms and two theoretically $\Theta(n \lg n)$ algorithms. For n^2 algorithms, we selected insertion sort and selection sort; for $n \lg n$ algorithms we selected mergesort and quicksort. We tested the runtimes of these algorithms by performing the Ratio Test.

Theory

All pseudo-code is base 1 indexed, instead of standard base 0

Insertion Sort

Here is the pseudo-code for our insertion sort implementation:

```
1   for j = 2 to A.length
2   key = A[j]
3   i = j - 1
4   while i > 0 and A[i]  >  key
5       A[i+1] = A[i]
6       i = i - 1
7   A[i+1] = key
```

The line which runs the most in this insertion sort implementation is the `while` loop on line 4. In a best case scenario, where the data is already sorted, the `while` loop will run once per iteration of the for loop thus:

$$\sum_{j=2}^{n} 1 = n - 1 = \Theta(n)$$

For insertion sort a worst case scenario is one where the data is in reverse sorted order. This means the `while` loop will run j times for each iteration of j:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

On average, with randomly sorted data, the while loop will run about $\frac{1}{2}$ of j iterations:

$$\sum_{j=2}^{n} \left(\frac{j}{2}\right) = \frac{1}{2}\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Selection Sort

Here is the pseudo-code for our selection sort implementation:

```
1   for j = 1 to A.length - 1
2   smallest = A[j]
3   for i = j to A.length
4       if A[smallest] > A[i]
5           smallest = i
6       swap A[j] ⇆ A[smallest]
```

The line which runs the most in selection sort is the second `for` loop which will run j times for each iteration of the first `for` loop. Selection sort's worst, best, and average case all run the same amount of times, regardless of the data. This is because of the double `for` loop structure and since there is no way to break out of it, it will run through both every time.

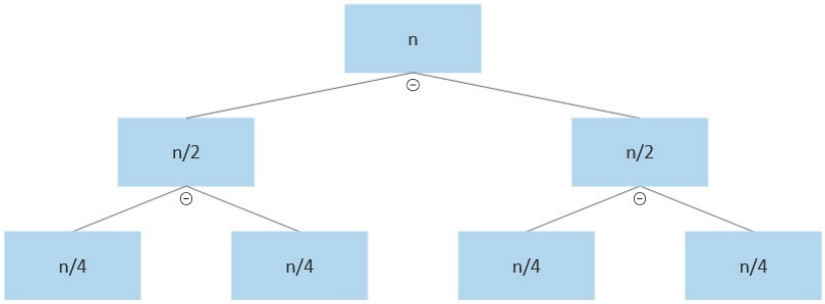
$$\sum_{j=1}^{n-1} j = \sum_{j=1}^{n} j - n = \frac{n(n+1)}{2} - n = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$$

Mergesort

Mergesort is a recursive algorithm with two parts. Here is the psudeo-code for `MergeSort` and `Merge`:

```
1   MergeSort(A,p,r)
2       if p < r
3           q = ⌊ (p + r)/2 ⌋
4           MergeSort(A,p,q)
5           MergeSort(A,p,q)
6           Merge(A,p,q,r)
1   Merge(A,p,q,r)
2       n1 = q - p + 1
3       n2 = r - q
4       let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
5       for i = 1 to n1
6           L[i] = A[p + i - 1]
7       for j = 1 to n2
8           R[j] = A[q + j]
9       L[n1 + 1] = ∞ and  R[n2 + 1] = ∞
10      i = 1 and j = 1
11      for k = p to r
12          if L[i] ≤  R[j]
13              A[k] = L[i]
14              i = i + 1
15          else A[k] = R[j]
16              j = j + 1
```

Line 11 runs the most in `Merge`, since there are no nested loops, we can say Merge runs on the order of $\Theta(n)$. Since `MergeSort` is recursive, the following recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, which can be modeled by the following recurrence tree:



The height of the tree can be calculated as: $\frac{n}{2^h} = 1$ or $\lg n = h$. Since n amount of work is being done on each level and since there are $\lg n$ levels there is $\sum_{i=1}^{\lg n} n$ or $n \lg n$ work being done. In other words: `MergeSort` is $\Theta(n \lg n)$. Mergesort, like selection sort does not change based on the data, so all cases will fit into $\Theta(n \lg n)$.

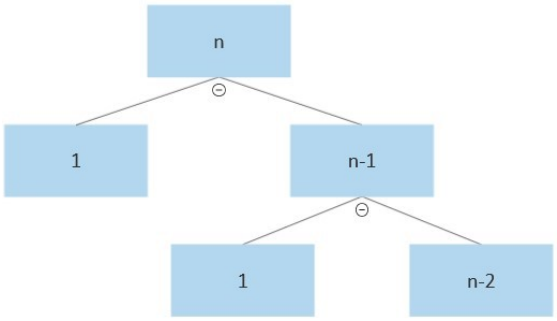
Quicksort

Quicksort is also recursive with two parts. Here is our pseudo-code for `QuickSort` and `Partition`.

```
1   QuickSort(A,p,r)
2       if p < r
3           q = Partition(A,p,r)
4           QuickSort(A,p,q - 1)
5           QuickSort(A,q + 1,r)
1   Partition(A,p,r)
2       x = A[r]
3       i = p - 1
4       for j = p to j ≤  r - 1
5           if A[j] ≤ x
6               i = i + 1
7               swap A[i] ⇆ A[j]
8       swap A[i + 1] ⇆ A[r]
9       return i + 1
```

In `Partition`, line 4 runs most frequently. Since there are no nested loops we can say `Partition` runs on the order of $\Theta(n)$. Since the index returned by `Partition` depends on the data, we must consider best, worst, and average case for `QuickSort`. Worst case for data for `QuickSort` is sorted

data, where `Partition` will return the index of r every iteration. This can be modeled with the following recurrence relation: $T(n) = T(1) + T(n-1) + \Theta(n)$ or as a tree:



The height of this tree is n. Starting from the bottom there is $1 + 2 + 3 + \ldots + n$ amount of work done on each level. This works out to:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Best and worst case for `QuickSort` are more complicated calculations. For simplicity we will just assume that `Partition` splits the array in half each time. That recurrence relation looks like this: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$. This is the same as the `MergeSort` derivation, but since we have some general assumptions, we will use an upper bound instead of the tight bound: $O(n \lg n)$.

Experimental

In order to test the theoretical run times of our algorithms, we needed to run the ratio test on each one, working up to large numbers for n. The ratio test is used to see if our algorithms converge as n grows to very large numbers. The formula for this test is:

$$\frac{T(n)}{F(n)}$$
$$where\ T(n) = run\ time\ and\ F(n) = the\ function\ to\ prove$$

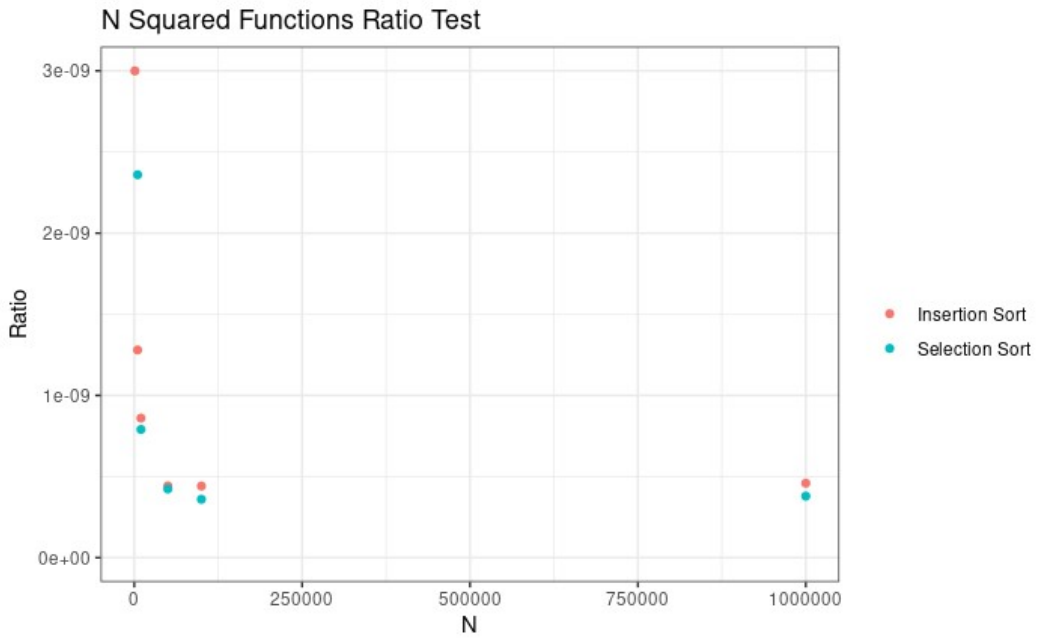
We ran our tests in Java using randomized data sets for every run-through.
For our n^2 algorithms, we were only able to go up to $n = 1,000,000$ because the amount of time it would have taken to run numbers any higher would have taken a very long time.

Insertion Sort

n	$T(n)$	$\frac{T(n)}{n^2}$
1000	0.003	0.000000003
5000	0.032	0.00000000128
10000	0.086	0.00000000086
50000	1.105	0.000000000442
100000	4.411	0.0000000004411
1000000	458.918	0.000000000458918

Selection Sort

n	$T(n)$	$\frac{T(n)}{n^2}$
1000	0.015	0.000000015
5000	0.059	0.00000000236
10000	0.079	0.00000000079
50000	1.058	0.0000000004232
100000	3.598	0.0000000003598
1000000	379.209	0.000000000379205



After graphing our results it appears as though both functions converge, indicating the algorithm in practice is $\Theta(n^2)$.

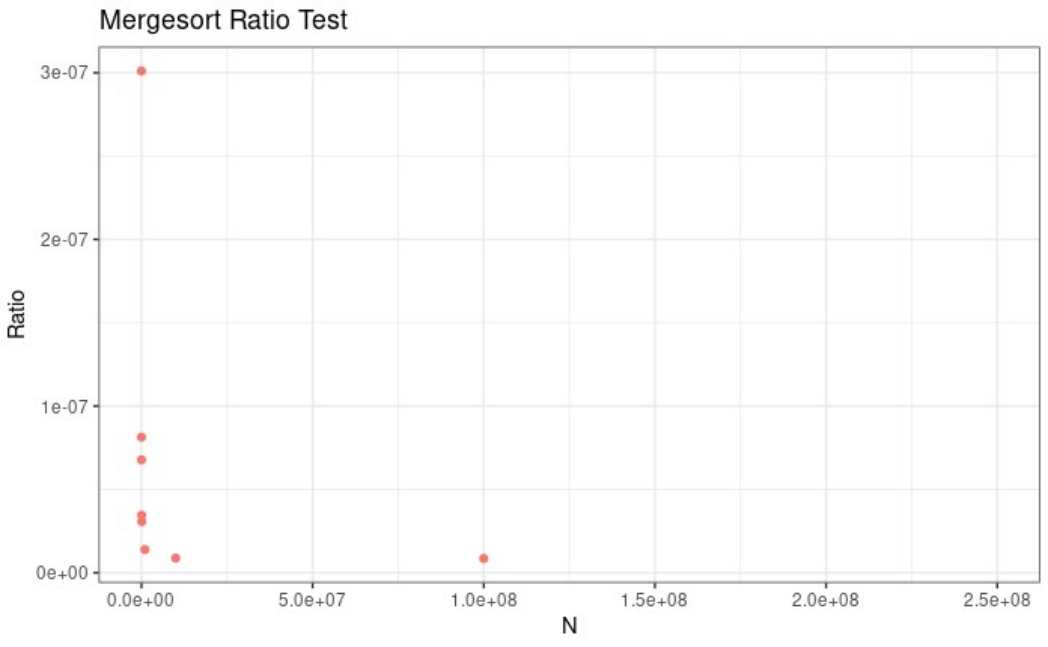
For our nlgn algorithms, we were able to use larger numbers, as they are much more efficient.

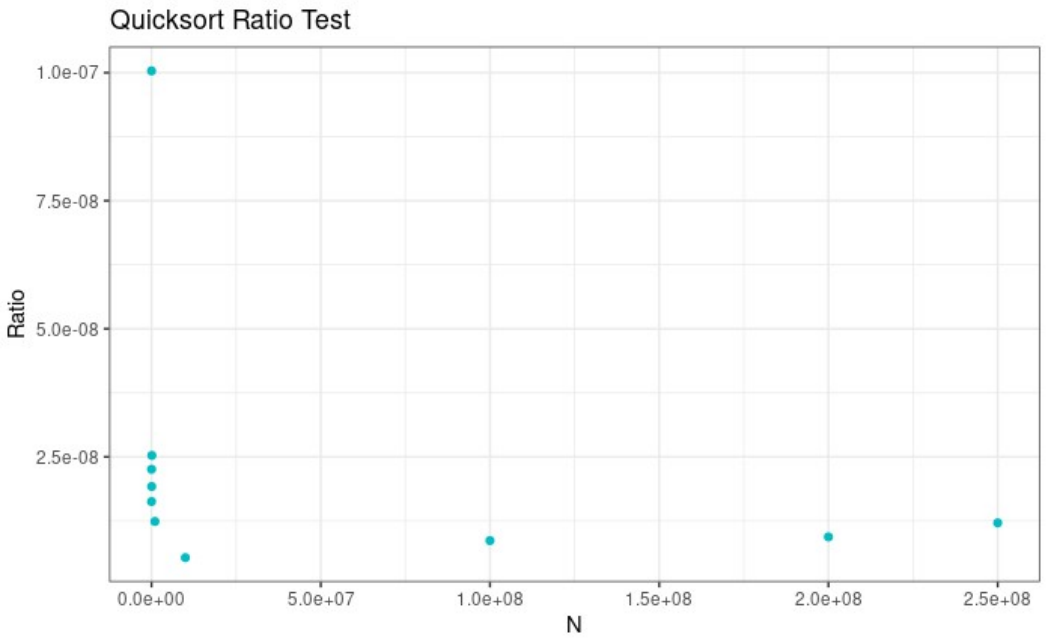
Mergesort

n	$T(n)$	$\frac{T(n)}{n \lg n}$
1000	0.003	0.0000003010299957
5000	0.005	0.00000008138211321
10000	0.009	0.00000006773174902
50000	0.027	0.00000003459400624
100000	0.051	0.00000003070505956
1000000	0.277	0.00000001389755147
10000000	2.05	0.000000008815878444
100000000	22.718	0.000000008548499302

Quicksort

n	$T(n)$	$\frac{T(n)}{n \lg n}$
1000	0.001	0.0000001003433319
5000	0.001	0.00000001627642264
10000	0.003	0.00000002257724967
50000	0.015	0.00000001921889236
100000	0.042	0.00000002528651964
1000000	0.247	0.00000001239240149
10000000	1.239	0.000000005328230923
100000000	23.002	0.00000000865536495
200000000	51.737	0.000000009380997836
250000000	84.52	0.00000001211871254





As is evident from the graphs, both Mergesort and Quicksort flatten out, indicating convergence. This shows that in practice these sorting algorithms are $O(n \lg n)$.

Graphics

Additionally to testing the algorithms experimentally, we decided to look at them all graphicly. This was done using OpenGl in C++. All four sorts can be seen at the same time, in order to get a good estimate of their speeds to one another. This is the configuration they are in:

Insertion Sort	Mergesort
Selection Sort	Quicksort

The way the graphics works is it creates a csv file for each sort, and every time the sort makes a comparison it appends the current array to the csv file. This means that during the "race" things like cache efficiency are not accounted for. For example, graphically, mergesort just barely beats quicksort with random data, but this has no way to account for the copy of the array mergesort makes, compared

to quicksort's in-line approach. Had we considered Heapsort, it would have likely differed from the experimental data drastically, considering how cache inefficient it is.

Another interesting observation between the graphics sorts, is that despite both being $\Theta(n^2)$, insertion sort finishes much faster than selection sort every time with random data. This is clearly due to the fact that it is able to break out of it's `while` loop while selection is stuck inside a nested `for` loop.