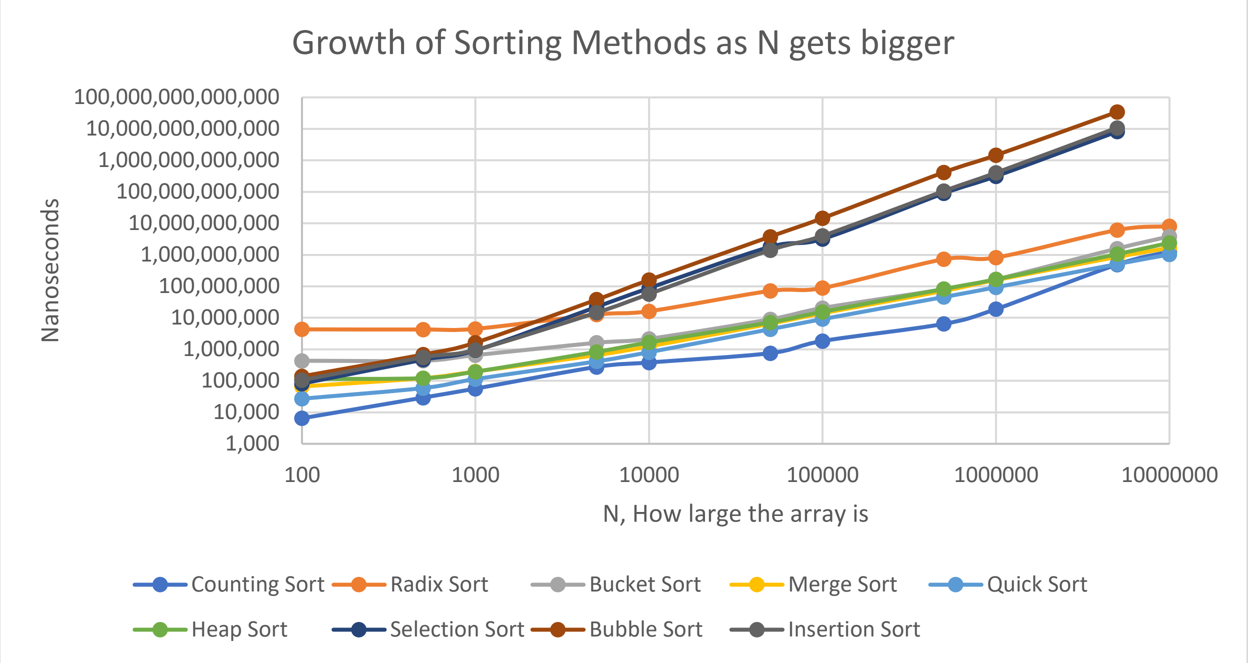
Chris LaFave Shomperithms HW3



As you can see, I decided to implement all of the algorithms, not just the three required. Also, I have the graph in logarithmic form on the axis's so that you can see all of the numbers (without it you could only see the lines separate when N was huge). I also did the time in nanoseconds because that's what the Java function I found uses, and the units don't matter a lot, we are just looking at the trend. Also as a matter of note, these points are the averages of 5 different runs of the sorts, except for the last points of the N^2 algorithms, which are only one run.

As expected, the N^2 algorithms skyrocketed to the moon. Logarithmic graphing actually doesn't do it justice; the highest point on the graph was 9.5 hours long. I was going to try to run these with the max N like the rest of the sorts, but seeing that number single-handedly changed my mind. Bubble sort is consistently and definitively the slowest sort out of the 9, basically doubling the other N^2 algorithms.

The Nlg_N algorithms look like they are growing linearly, which makes sense on this logarithmic graph. Heap was consistently the underperformer, and Quick always the second fastest behind Counting sort. Nothing seemed to be unexpected from this group; the overhead from Heap and Merge makes sense looking at how they were much slower than Quick, but not too much slower.

The N algorithms were interesting. Radix sort was always slower than the Nlg_N algorithms because of it's overhead, but it you can clearly see that it was growing at a much slower rate than those algorithms. It is clearly built to sort extremely long numbers, like social security numbers. Also, the elements in the array being from [0, N-1] does not show of its strengths compared to the other algorithms. Counting sort was consistently the fastest, being way faster than all of the other algorithms (except at the end, I don't know why it started to spike up, I'm chalking that up to human error). The elements being [0, N-1] definitely gave it a huge advantage over the other algorithms. If there had been a wider range of numbers (say, [0, 10*N]) it would have been slower due to needing to account for the bigger possibility of numbers, adding in more overhead.

And then there is Bucket sort. Bucket sort starts out like Radix, where it starts slower (because of overhead) but grows at a slower rate, but then it seems to become friendly with the Nlg_N algorithms and starts to grow at their speed. Here is what I think is going on. I probably didn't implement it optimally. The book used doubles for the Bucket Sort, but I wanted to compare all of the sorts on the SAME random array. Therefore, I had to figure out how to implement Bucket sort for integers. I landed on using N/10 buckets. Not because I did any math or thought it out, but because it seemed like a good arbitrary choice. With all that being said, I probably did not do Bucket sort (or Radix for that matter) justice, and this graph is probably not an accurate representation of its growth.