CSE 100 Project #1

Ian Camp

May 8, 2015

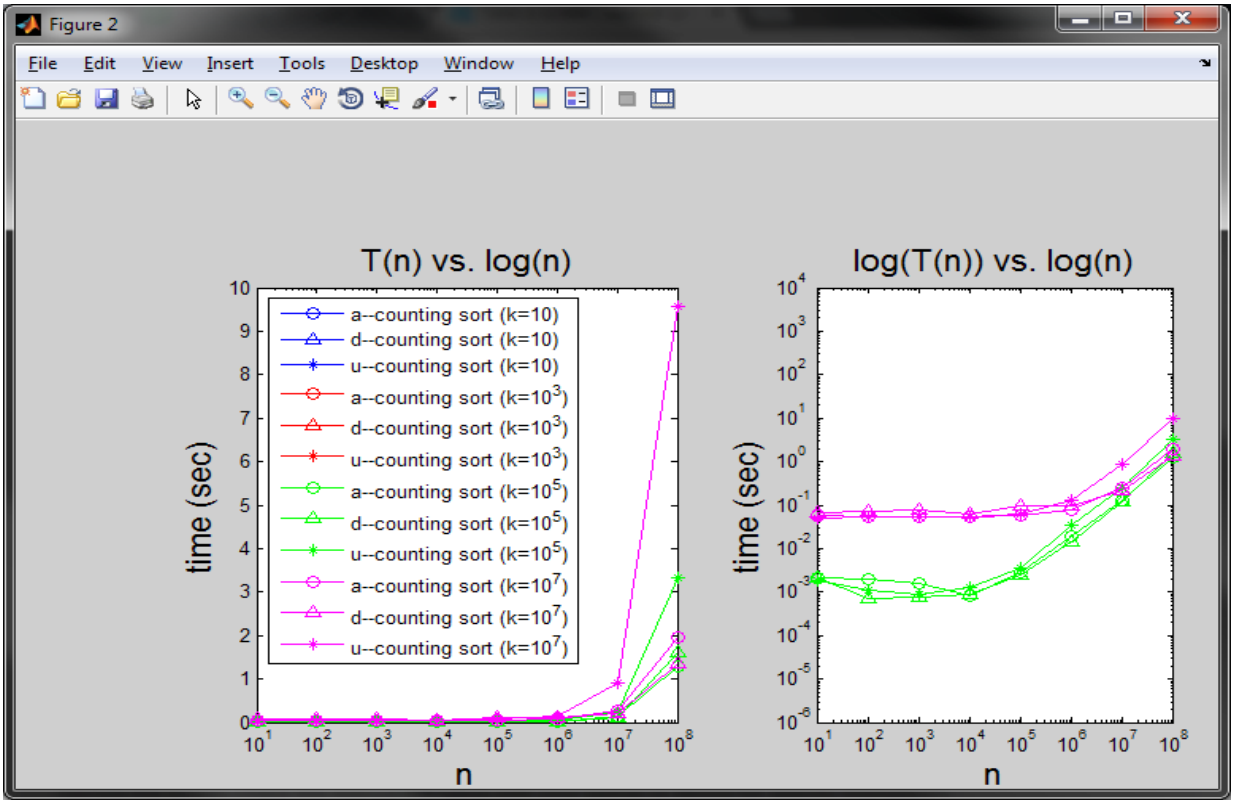
PC Specs

Intel(R) Core(TM) i3-2370M CPU @ 2.40GHz
(RAM) 4.00 GB
Linux Mint 12 (64-bit)

Intro

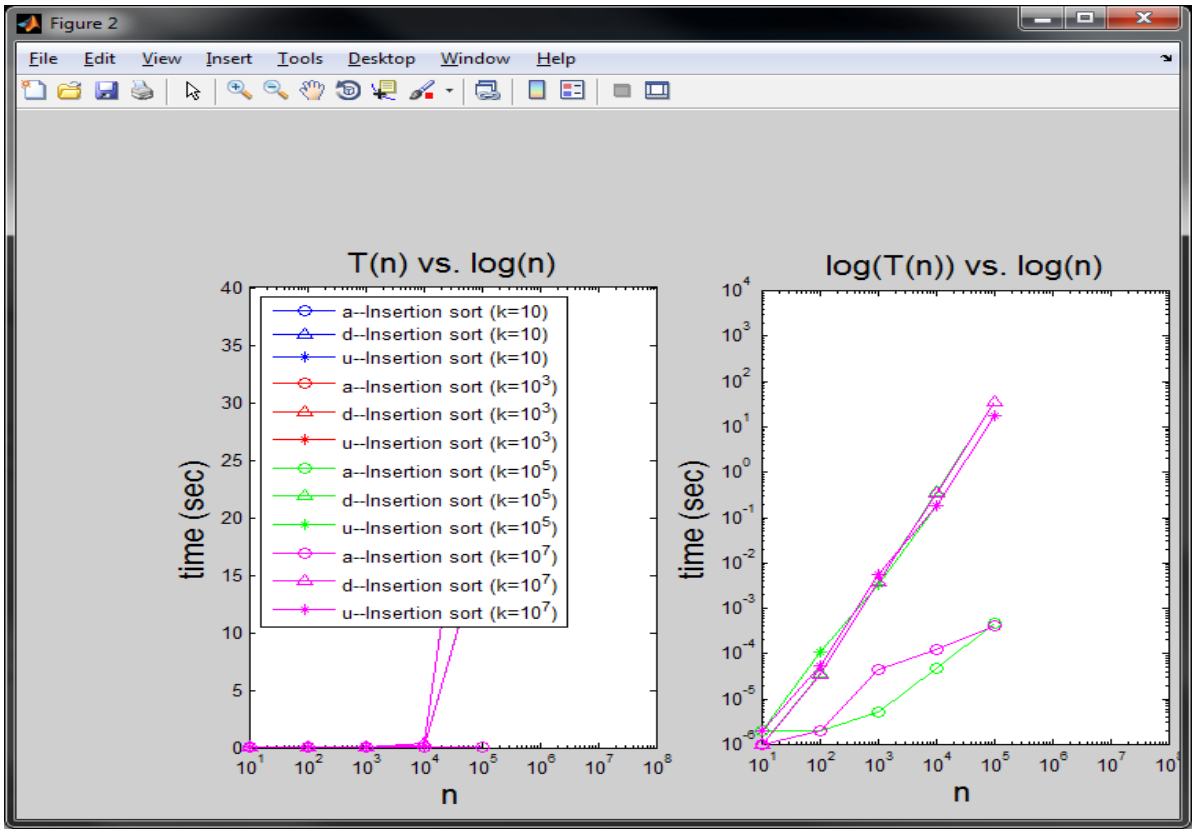
When I first started running the sorting algorithms I was confused by the time returned for some of my tests. I was waiting sometimes minutes for a set to complete and then getting a run time of a few seconds. It took me a while to realize that most of the time was being taken up by the initial generation of numbers and setting them in ascending or descending order. I also noticed the smaller sizes for k with large n seemed to take even longer for this initialization than the larger values of both k and n. So for the sake of saving myself time waiting for these algorithms to finish I cut a few sizes out of my runs.

Counting Sort



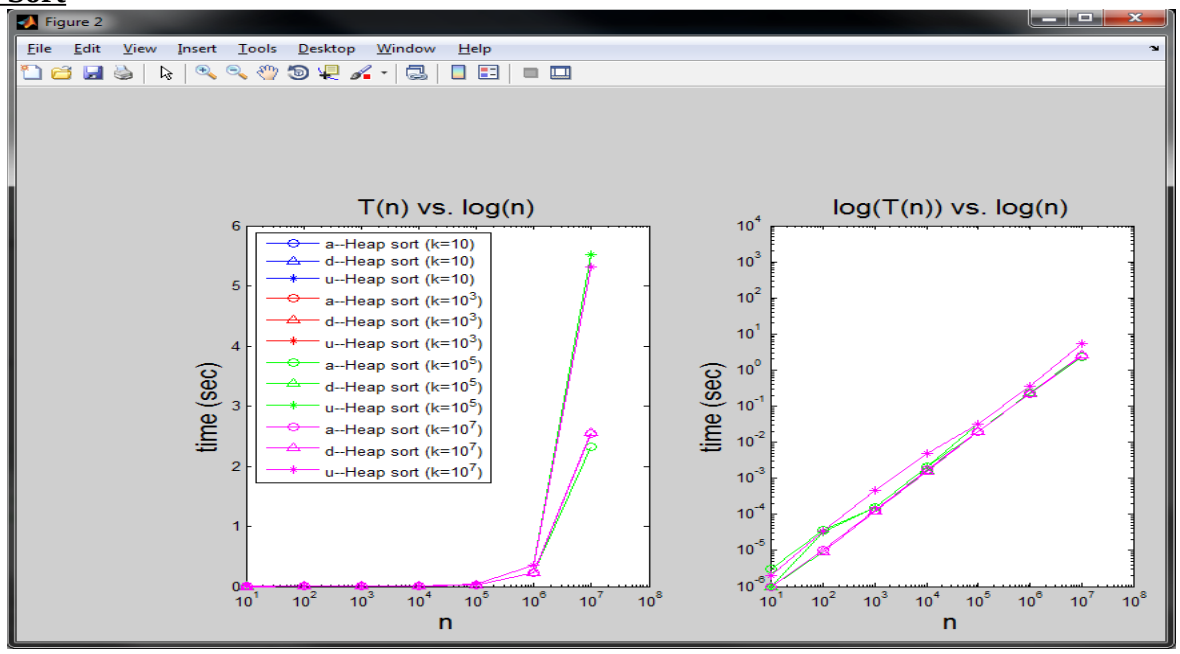
I noticed a lot of fluctuation in the smaller values of n. I figure this was due to other things happening on my pc at the time of running. The time complexity here should be O(n+k). The results proved that seemed to be the case. When the size of k increased the graph jumped up respectively. I was surprized how quickly this one worked at first thinking that this was a simple algorithm so it should have been a slower.

Insertion Sort



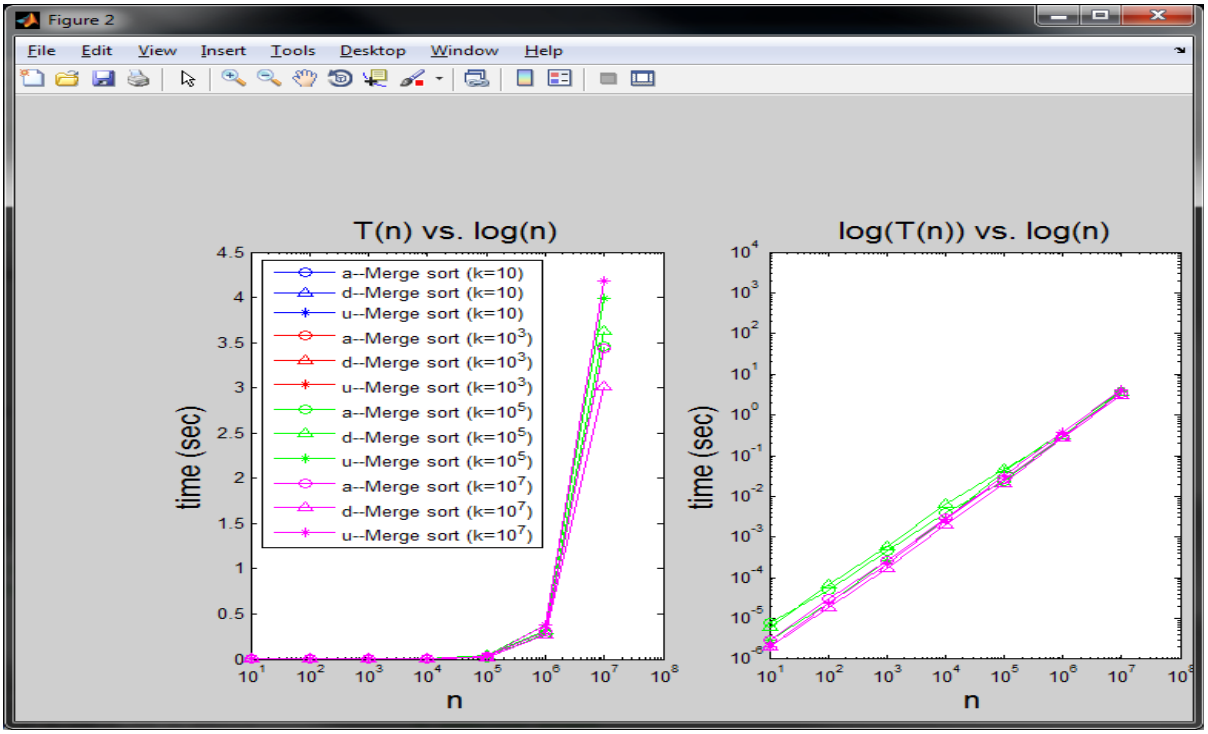
This Algorithm should have a best case of O(n) and an average/worst case of O(n^2). I found that having an ascending list provided the best times for this algo while the unsorted and descending lists gave a worse time. Results match what they should.

Heap Sort



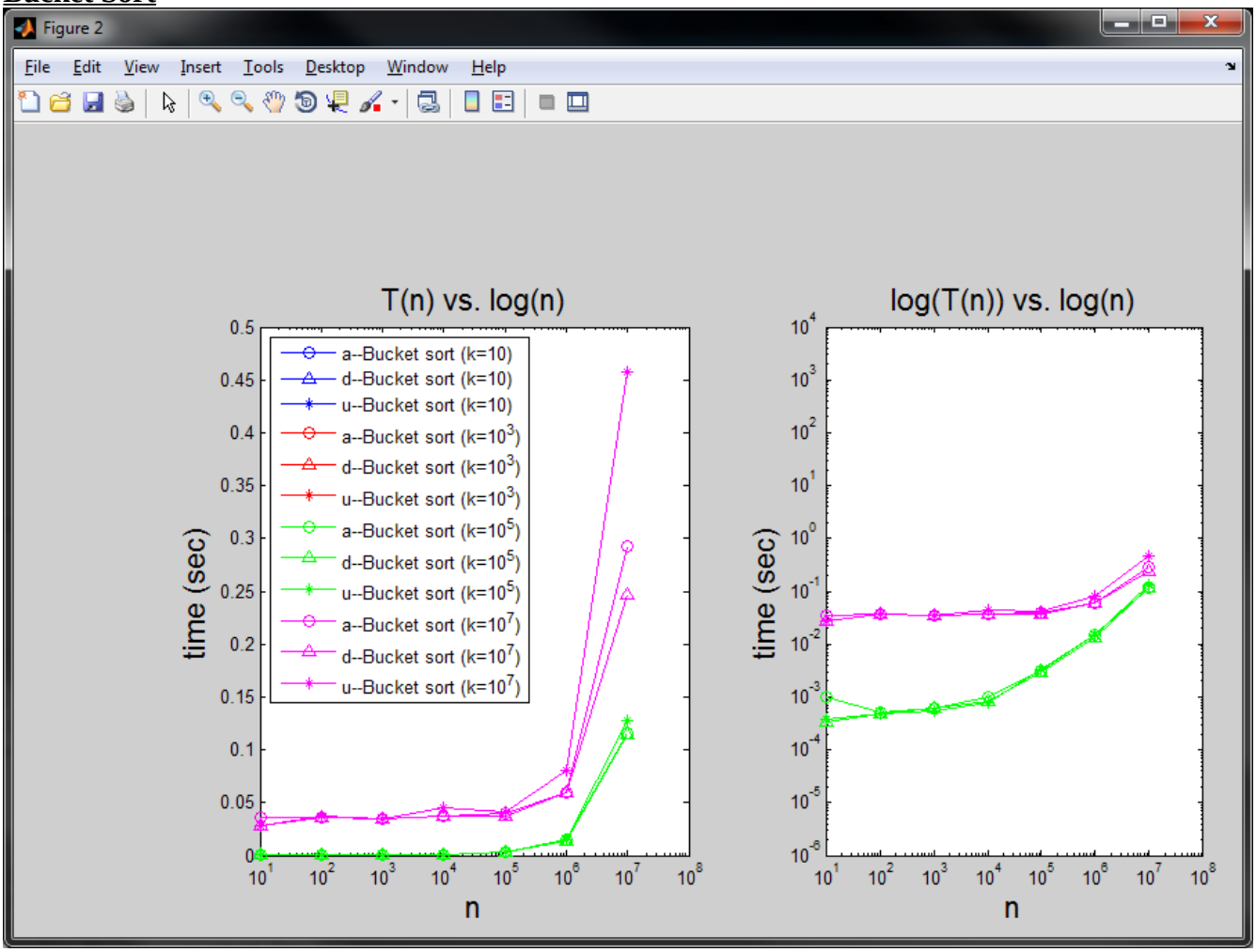
The heap sort should have a best, average, and worst case all of O(n log(n)). The results I got seem to match as all types of sorted data going in gave close to the same times.

Merge Sort



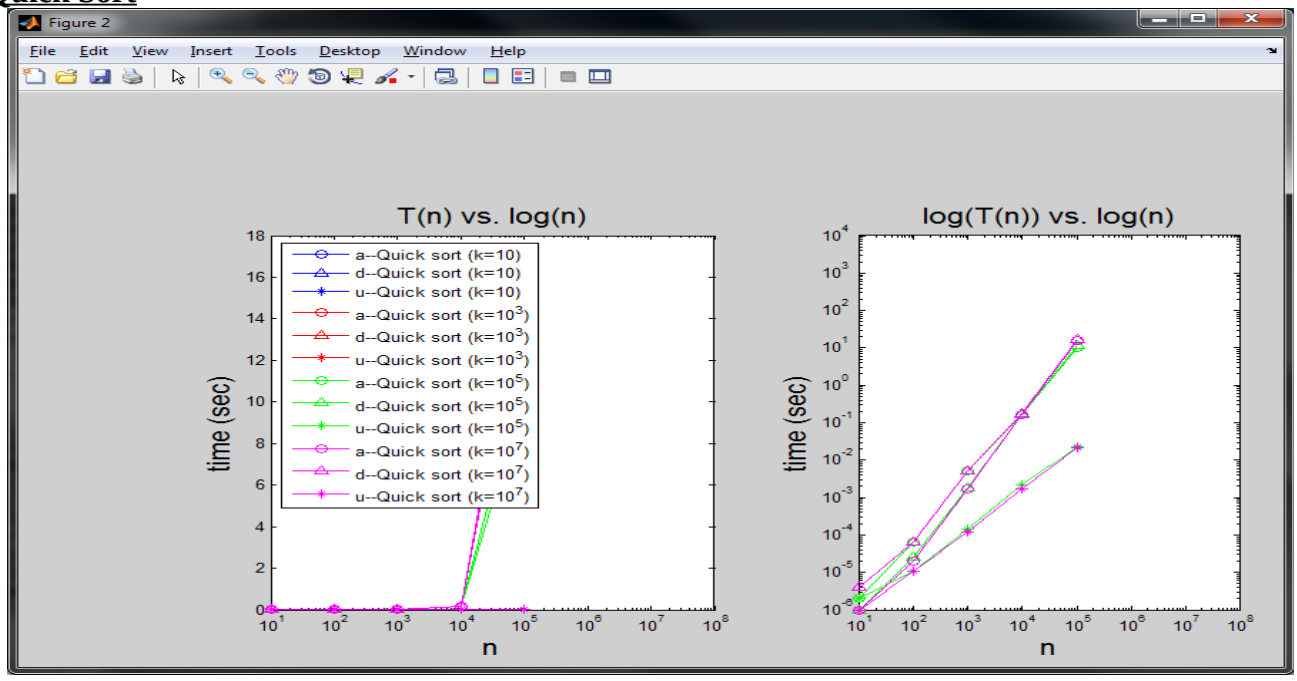
Merge sort has the same time complexity as Heap sort with O(n log(n)). Again results were very similar to that of heap sort. When I looked into it a bit more to discover what reason anyone would have to choose one over the other I noticed this one had a larger space complexity than that of heap sort.

Bucket Sort



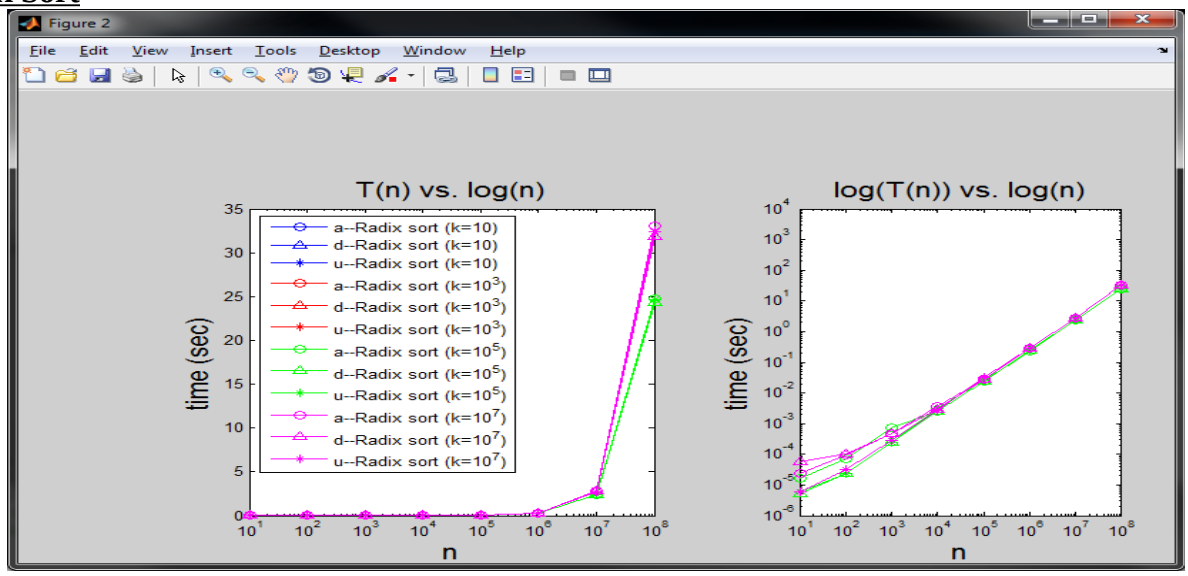
This was very similar to Counting Sort. It should have a best and average of O(n+k) and a worst of O(n^2). I was able to see where the k values affected it as it did in counting sort but didn't notice the worst case. Maybe larger values of n are needed to see the worst case better.

Quick Sort



Quick sort has a best and average of O(n log(n) and a worst case of O(n^2). It was interesting to see that the unsorted data gave better times than the sorted data. Both ascending and descending gave similar times and I found those cases to be the worst.

Radix Sort



Radix sort has a time complexity of O(n*k). This one had the most noticeable trend in the graph and shows nicely the time complexity.

Conclusion

The first two algorithms I ran were counting sort and then quick sort. I noticed that counting sort had a much faster run time than quick sort and was perplexed by this at first. For some reason I had it in my mind that quick sort should have been faster. I began wondering why if counting sort is one of the fastest algorithms then why pick any of the other ones. It's even a much simpler than all the rest. Then I realized that if the largest value in the list was a huge size this would mean counting sort would need to make space for a list with a size of that number. This quickly became a problem for space complexity and made sense why you wouldn't want to use counting sort in many cases. This project helped me understand the importance of knowing not only time being important but also space required to run. Before one chooses an algorithm for sorting it is not just important to think about how much data there is but also the kind of data that is being used.