cs361 Assignment 2

Ryan Scherbarth

February 15 2024

1 Repository

https://github.com/rysc3/cs361/tree/main/assignment2

2 Platform

During this project I worked on multiple differnet platforms to test my results. I started on my ubuntu server, which is Ubuntu 22.04 running on a 2019 macbook pro. This machine has a 12 core (6 physical cores with 2 threads each) cpu, running at 2.6GHz, and 16GB ddr4-2600 of memory. Specifically, this is the last version of the macbook pro that contained a cpu with an x86 instruction set, after which apple switched to their M series cpus, which are built with an ARM architecture. One that doesn't have a solid distribution of linux out at this time.

This is where I started running my program. While everything worked relatively well, and I did have ample memory available to fit the full $2,147,483,647$ element array in memory, such that I was able to do the insertion sort algorithm on it. This, however, takes a very long time to complete, especially so with my implementation that was running on a single core, so I did not wait for it to complete. The issue here was that I quickly ran into a space complexity issue.

Insertion sort has a space complexity of $O(1)$, since we are only moving values around in the pre-existing array. The issue is when we notice that merge sort has a space complexity of $O(n)$. Where as a problem of size mem-in-gb = num-integers * size-of-integer, in this case, $\frac{2{,}147{,}483{,}647*4}{1024^3} =\sim 8GB$. In the case of insertion sort, this remains constant. However, when we get to merge sort the memory we would require increases significantly, and was unable to fit into the memory on this machine.

Because of this, I found the largest value I could fit into memory to run was roughly two million. While this value fits in my memory and is able to run all three algorithms, to use insertion sort for a 2 million element array will take a very long time. I settled at a value of $200,000$ for a compromise of time and ample problem size.

I was interested in seeing the time continue to scale with large values, so I also created a slurm script and moved my code over the the Xena cluster at

UNM's Center for Advanced Research Computing, where I was able to request and run my program on a partition that has 1TB of memory/Node. The node I was running on has an Intel Xeon E7-4809 cpu running at 2.00 GHz, and just like the iteration on my ubuntu machine, we are still only running on a single core.

For my program, I chose to use C. Really, I just chose this over java because it is more light-weight and I knew from the start one of our constraints might end up being the ability to fit our problem into memory. Java carries a lot of extra weight and I thought it would be much more difficult to determine exactly where my memory was being used.

3 Runtime Analysis

Starting with my ubuntu machine, and a problem size of $n = 200,000$, I came to the following results by running each algorithm 20 times, each with completely unique arrays, and then calcualting the mean for the running time. The results, in seconds, were as follows::

	Insertion Sort	Merge Sort	Quick Sort
n=20	0.000001	0.000002	0.000001
n=2,000	0.002201	0.000261	0.000189
n=20,000	0.201056	0.002515	0.001839
n=200,000	19.800001	0.030309	0.023157
n=2,000,000	very long	0.362779	0.267257
n=2,147,483,647	very long	seg. fault	seg. fault
n≥2,147,483,648	seg. fault	seg. fault	seg. fault

There are a few things we can take away from these results. Firstly, we can see that for very very small problem sets, insertion sort actually ends up beating out merge sort. This could be applicable in a case such as a stock ticker. Imagine we have a list of, say, 3 stocks that we are watching. These will be updating very frequently, maybe once every second, with the most recent price. We can imagine these are all on a watchlist similar to the stocks app on an iphone, they will constantly be updating as far as their position on the list based off of a particular factor, maybe the percent change for the day, or your total value you have in a given position. This is an example where we might be sorting a very small list many, many times over, and it would actually be more efficient to use insertion sort.

There is one other issue with this data, as we begin to notice our limitations. I cannot realistically scale insertion sort across higher $n's$, however this is not as significant as insertion sort grows at an order of magnitude faster than the other two algorithms. The major issue we have is that, due to their recursive nature, I am only able to scale to roughly $n = 2,000,000$ before I run into segmentation

faults. Regardless, we can still see the direction we are going with the limited data we were able to gather, which is that as n increases, not only is merge sort always slower than quick sort, but also the difference between the two continues to increase. In other words, while the difference between a choice of merge sort and quick sort on relatively small datasets - realistically, anything that can fit into an integer, is negligible, as the value of n increases, the amount of time you gain from choosing quick sort over merge sort does the same. We are seeing that quick sort's runtime grows at a faster rate than quick sort.

I then gathered the results from running our recursive algorithms at CARC. The results, in seconds, here were as follows:

	Merge Sort	Quick Sort
n=2,147,483,647	1197.970000	886.650000

And here, as our problem size increases even further, we see this difference continue to grow. In this case, we have roughly a 13.5% difference between the algorithms. For reference, a google search for 'a' returns $25,200,000,000$ results, so there are no shortage of problems that contain $> 2,000,000,000$ elements.

4 Technical Analysis

Insertion Sort exhibits a runtime complexity of $O(n^2)$ in the worst case, making it less suitable for large datasets. However, for small problem sizes, it can outperform more sophisticated algorithms due to its simplicity and low overhead.

Merge Sort demonstrates a runtime complexity of $O(n \log n)$ in all cases, making it more efficient than Insertion Sort for larger datasets. Despite its superior performance for large problem sizes, Merge Sort incurs higher overhead due to its recursive nature and memory requirements.

Quick Sort's average-case runtime complexity is $O(n \log n)$, making it comparable to Merge Sort. However, it exhibits a worst-case complexity of $O(n^2)$, which occurs when the pivot selection is poor. Nonetheless, its average-case performance often surpasses that of Merge Sort due to lower constant factors and efficient partitioning.

Apart from runtime considerations, the space complexity of sorting algorithms influences their practicality, especially concerning memory usage. Insertion Sort has a space complexity of $O(1)$ since it operates in-place, making it memory-efficient. However, its quadratic runtime complexity limits its applicability to larger datasets.

Merge Sort's space complexity is $O(n)$ due to its recursive division of the array, which necessitates additional memory for temporary arrays during the merging phase. While this overhead can be significant for large problem sizes, it guarantees stable and predictable performance.

Similarly to Merge Sort, Quick Sort's space complexity is $O(n)$ due to its recursive partitioning. However, its in-place partitioning reduces memory overhead compared to Merge Sort's merging phase.