Deznell Dixon
7/28/21
Algorithms CCNY

Approaching this project, I found it best to use functions for each algorithm. When programming on an everyday basis, prebuilt sorting algorithms tend to always be called as a function. To help write my algorithms I used programiz.com primarily among other online resources. The data structure used to store the numbers was C styled arrays. C styled arrays are fast and simple to use. They make it easy to store and delete things on the heap. In addition, since I am not resizing the arrays, the C styled arrays area perfect choice. I began coding by creating an array of pointers called numArr. This array stores pointers to arrays of varying length. This will store the original arrays which I will make copies of. I then used an array of functions to systematically call each sorting function on each array. I found the results and this project to be very helpful in understanding algorithms and creating them. Now I will move on to each algorithm.

The Insertion sort algorithm overall, can be considered the slowest out of the other algorithms when n is sufficiently large. The insertion sort uses a brute force method and has O(n^2) complexity. For n < 100, the insertion sort algorithm performed very similar to merge and heap algorithms. Once larger than 1000, the merge sort algorithm proved to be very slow. When N = 100,000, the merge sort algorithm took more than 11 seconds, when N = 1,000,000, the program did not finish. The Insertion sort algorithm was however, simple to implement and has O(1) space complexity. It is best suited for smaller numbers.

The merge sort algorithm was on of the most efficient algorithms. The merge sort algorithm works by use of the divide and conquer method. It divides the problem into it’s smaller components and then merges it all together. This method uses much less comparisons and has a run time of O(n*logn). The algorithm performed very similar to heap sort an quick sort. It had an almost linear growth rate. The longest runtime of which only took 0.2 seconds. The merge algorithm was harder to implement but it was very efficient when handling larger N’s. It has a space complexity of O(N).

The Heap sort algorithm works by creating a binary tree. This binary tree then swaps elements until the order is correct. This binary tree is called a heap, hence the name. The heap sort algorithm has O(nlogn) run time. The run time of the heap sort algorithm was slower than the merge and insertion sort. However, when N was sufficiently large, it had a slower run time than insertion sort. This algorithm also has a space complexity of O(1). This algorithm was the hardest to implement and understand as it is being represented as a binary tree. The longest run time of the heap sort algorithm was 0.4 seconds.

The Quick Sort algorithm is a very fast algorithm for values of smaller values of N. Across the aboard, it had a smaller run time than the other algorithms. The quick sort algorithm works by the divide and conquer approach. It divides the algorithm into smaller sub arrays, by the selection of pivot points. Each time, you cut the amount of comparisons you have to do in half. The algorithm has a O(nlogn) time complexity and a O(logn) space complexity. The quick sort algorithm can be effectively used for values < 100,000. When N is 1 million, the quick sort algorithm begins to break down with much longer run time compared to Heap sort, 3.5 compared to 0.4 seconds. Quick Sort also has a worst case run time of O(N^2). This means that depending on how the data is randomized, the run times can drastically change.

The Quicksort random algorithm is the same as the quicksort algorithm however, instead we are picking a random number to pivot on. This approach is stated to improve the run time, however, only a slower run time was observed. When N=10 the algorithm took 16,000 nanoseconds, almost ten times longer than quick sort. When N = 1 million the Quick sort algorithm took twice as long at around 6.0seconds. The worst time complexity is the same at around O(N^2) seconds.

The final algorithm is the Radix algorithm. The Radix algorithm sorts by comparing place values of the numbers. This algorithm was one of the fastest for this project because numbers were used with place values between 1 and 3. If all numbers had the same place value, this algorithm would not be as efficient. The opposite is true for numbers with a lot of place values. The algorithm has a time complexity of O(n+C), where C is a constant. It also has the largest time complexity. When N= 10, the Radix sort performed slower than the other algorithms, however, when N=1 million, it had the fastest run time of 0.1 seconds.

Overall this project has been a great learning opportunity. The results from my code run time is shown below.

N	Insertion	Merge	Heap	Quick	Quick Random	Radix
10	1600	2100	2000	1700	16000	2200
100	12700	27600	26700	11400	176700	12900
1000	2279300	234000	337800	183500	2017100	142000
10000	111974500	1624400	2458900	1514300	16514400	834600
100000	11752666900	19925100	33724000	49018300	235853100	8851900
1e+06	0	212845600	441313300	3501962200	6085895100	110835700

Note: All values in nano seconds. Having my laptop plugged in resulted in faster compiling and run times across the board.

Sources:

1. https://www.programiz.com/dsa/radix-sort
2. https://www.programiz.com/dsa/quick-sort
3. https://www.programiz.com/dsa/heap-sort
4. https://www.programiz.com/dsa/merge-sort
5. https://www.geeksforgeeks.org/quicksort-using-random-pivoting/