Te

****

**PA2 Report**

By

**Tempest Hopp**

**CSI 281**

**Data Structures & Algorithms**

**Fall 2021-22**

1. **Introduction**

The goal for this experiment is to test the speeds of different sorting algorithms: bubble sort, insertion sort, selection sort, shell sort, quick sort, and merge sort. These algorithms will then be tested in three different conditions: random order to represent average case, ascending order to represent best case, and descending order to represent worst case. My hypothesis is that quick sort will be the quickest overall.

1. **Background**

Bubble sort is the most basic algorithm, but also runs on the slower side. You compare the current value with the next value and swap it if it’s out of order. Under the best circumstances, it has a big O of O(n), but most often leans towards O(n2) with the worst case also being O(n2). Insertion sort is also very simple. It’s almost like a variation of bubble sort, since you take a value, and then place it into a portion of the list in the proper order, like how you would a hand of cards. Insertion sort’s performance also mirrors that of bubble sort with a best case of O(n), and an average and worst case of O(n2).

Selection sort is slightly different from these two. You start at the front of your list, and then search for the smallest value. You then swap your current value with the smallest value and continue down the list. No matter the size or ordering of the list, using selection sort will always yield a big O of O(n2). Shell sort, in my eyes, is an augmented version of bubble sort. Instead of looking at the next value in the list, you look halfway down the list and do the same processes as bubble sort, slowly narrowing down the gap between the current value and the value you’re looking towards, ending with simply using bubble sort. Shell sort’s performance depends on the size of the list, but the best case will always be O(n).

Quick sort and merge sort are the most complex sorting algorithms within the experiment, with quick sort using recursion, and merge sort breaking down the array and then reassembling. Quick sort takes the last value in the list, and then sorts the array into greater-than or less-than this value, with it residing in the middle. You then repeat the process for the chunks of the list on either side of this middle value, until you’ve gotten through the entire list. Quick sort has a best case and average case of O(n log n), and a worst case of O(n2). Merge sort simply disassembles the list, then reassembles it in order. No matter the condition of the list, merge sort will always have a performance of O(n log n).

1. **Implementation Detail**

For bubble sort, you start with a Boolean variable to implement a while to track if a swap occurred that iteration of the proceeding for loop. Within this for loop, you go through every value in the array and compare it to the next value. If the next value is less than the current value, you swap the values and set the Boolean to true. The code ends once the while loop is false, indicating a swap didn’t occur that iteration.

In insertion sort, you start by taking the first two values in the list and putting those two in order by swapping if necessary. From there, you take the next value in the list, and find where it properly goes within those two sorted values. You insert it in its proper place, and you continue the pattern for the rest of the list.

Selection sort revolves around finding the minimum value in the unsorted section of the list. You keep the index of the first value in the list, and then search for the minimum value in the list and log its index. Then, you swap your first value and the minimum value, and then repeat the process for the second, third, and so forth indices of the list.

Shell sort is implemented similarly to bubble sort, but instead of checking the next value, you have a variable “gap” that initially starts out as half the size of the list but gets cut in half every iteration of the for loop. Within the for loop, you do the same process as bubble sort, comparing the two values, and swapping if they’re out of order. You continue this process until your “gap” has a value of 1, and then you call bubble sort to finish off the sorting.

Quick sort is the first of our recursive sorting algorithms, which lends to its lower big O value. You start by taking the last index and its value, and then partitioning the list into two sections: smaller than this value, and larger than this value. You then place this last value into its proper place between these two sections, and recurse back through the function with the left section and the right section of the list, continuing until you’re left with single values.

Merge sort is the most complex of the sorting algorithms. We start by finding the midway point in the list, and then immediately recursing through the function with the two halves, until we’re left with lists of size 1. From there, we slowly reassemble the list by going one value at a time and comparing each of the two temporary lists. You slowly build up the list until one is empty, then you add the rest of the second list to the end. You continue this process until the recursive calls are over.

1. **Experimentation Detail**
   1. Memory on board: Corsair Vengeance RGB Pro 16GB (2x8GB) DDR4 3200MHz C16 LED Desktop Memory – White
   2. Processor type: AMD Ryzen 5 1600 Six-Core Processor, 6 Cores, 12 Logical Processors
   3. CPU speed: 3200 MHz (3.2 GHz)
   4. System type: 32 bits or 64 bits: 64 bit

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0003s | 0.00000063s | 0.0006s |
| 10,000 | 3.1228s | 0.8953s | 5.1147s |
| 100,000 | 414.4279s | 374.4002s | 511.8343s |
| **Algorithm: Insertion Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.000009s | 0.000001s | 0.00002s |
| 10,000 | 0.0926s | 0.00005s | 0.1806s |
| 100,000 | 7.3091s | 0.0005s | 14.5014s |
| **Algorithm: Selection Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.00003s | 0.00002s | 0.00003s |
| 10,000 | 0.1289s | 0.1310s | 0.1568s |
| 100,000 | 12.6604s | 12.6313s | 19.5842s |
| **Algorithm: Shell Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.00009s | 0.000004s | 0.00006s |
| 10,000 | 1.2467s | 0.9186s | 0.9222s |
| 100,000 | 378.1081s | 373.6875s | 864.5241s |
| **Algorithm: Quick Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.00003s | 0.00001s | 0.00002s |
| 10,000 | 0.0047s | 0.0016s | 0.0019s |
| 100,000 | 0.0599s | 0.0208s | 0.0251s |
| **Algorithm: Merge Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0001s | 0.00009s | 0.0001s |
| 10,000 | 0.0100s | 0.0094s | 0.0101s |
| 100,000 | 0.1026s | 0.0936s | 0.0919s |

1. **Discussion and Conclusion**

By looking back at the data, you can see that as the data set got larger, the time required grew exponentially. You can also tell the fastest case was always the ascending order, and slowest was always descending order. My hypothesis was correct, that quick sort would be the fastest.

If I had to rank the sorting algorithms, I’d put quick sort at the top of the list, followed by merge sort. Even if they’re more complex, they make up for it in speed. I’d put insertion sort next, as it was among the fastest of the rest of the algorithms. The last three would be selection sort, then shell sort, then bubble sort. Even though shell sort had a much larger average for one of the cases, it was due to an outlier in running the code.

Selection sort has always been my go-to sorting algorithm to implement, as the concept has always been the easiest for me to understand. Now, however, I may switch to quick sort as I like the speed of the algorithm a lot and it is still very easy for me to understand and implement.

1. **References**

Hammad, Jehad, “A Comparative Study between Various Sorting Algorithms,” International Journal of Computer Science and Network Security vol. 15, no. 3, pp. 11-15, Mar. 2015.

Vaughan R. Pratt, “Shellsort and Sorting Networks,” Stanford Univ. California, Stanford, Rep. 10, Feb. 1972.

1. **Appendix**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0004s | 0.0003s | 0.0003s | 0.0003s |
| 10,000 | 3.2144s | 3.1318s | 3.0223s | 3.1228s |
| 500,000 | 423.0897s | 407.9457s | 412.2483s | 414.4279s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0. 0000005s | 0.0000008s | 0.0000006s | 0.00000063s |
| 10,000 | 0.8968s | 0.9123s | 0.8767s | 0.8953s |
| 500,000 | 379.9111s | 372.7451s | 370.5445s | 374.4002s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0005s | 0.0007s | 0.0005s | 0.0006s |
| 10,000 | 5.0750s | 5.1165s | 5.1026s | 5.1147s |
| 500,000 | 520.3586s | 507.8411s | 507.3032s | 511.8343s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.000009s | 0.00001s | 0.000009s | 0.000009s |
| 10,000 | 0.0904s | 0.0731s | 0.1142s | 0.0926s |
| 500,000 | 7.527s | 7.1957s | 7.2045s | 7.3091s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.000002s | 0.0000007s | 0.0000008s | 0.000001s |
| 10,000 | 0.00008s | 0.00004s | 0.00003s | 0.00005s |
| 500,000 | 0.0008s | 0.0003s | 0.0003s | 0.0005s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00002s | 0.00002s | 0.00002s | 0.00002s |
| 10,000 | 0.1627s | 0.1463s | 0.2328s | 0.1806s |
| 500,000 | 14.8280s | 14.2918s | 14.3845s | 14.5014s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00003s | 0.00003s | 0.00003s | 0.00003s |
| 10,000 | 0.1283s | 0.1284s | 0.1301s | 0.1289s |
| 500,000 | 12.9109s | 12.5082s | 12.5622s | 12.6604s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00002s | 0.00002s | 0.00003s | 0.00002s |
| 10,000 | 0.1278s | 0.1376s | 0.1277s | 0.1310s |
| 500,000 | 12.9746s | 12.3821s | 12.5373s | 12.6313s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00002s | 0.00005s | 0.00003s | 0.00003s |
| 10,000 | 0.1562s | 0.1561s | 0.1581s | 0.1568s |
| 500,000 | 19.9679s | 19.2485s | 19.6363s | 19.5842s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0001s | 0.00008s | 0.00008s | 0.00009s |
| 10,000 | 1.2639s | 1.2161s | 1.2602s | 1.2467s |
| 500,000 | 386.3816s | 374.0674s | 373.8752s | 378.1081s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.000006s | 0.000003s | 0.000004s | 0.000004s |
| 10,000 | 0.9621s | 0.8748s | 0.9189s | 0.9186s |
| 500,000 | 384.0071s | 369.3094s | 367.7460s | 373.6875s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0001s | 0.00004s | 0.00005s | 0.00006s |
| 10,000 | 0.9612s | 0.8766s | 0.9288s | 0.9222s |
| 500,000 | 1853.8338s | 370.4154s | 369.3230s | 864.5241s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00003s | 0.00003s | 0.00003s | 0.00003s |
| 10,000 | 0.0049s | 0.0046s | 0.0047s | 0.0047s |
| 500,000 | 0.0610s | 0.0596s | 0.0590s | 0.0599s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00001s | 0.00002s | 0.00001s | 0.00001s |
| 10,000 | 0.0013s | 0.0022s | 0.0013s | 0.0016s |
| 500,000 | 0.0203s | 0.0219s | 0.0201s | 0.0208s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.00002s | 0.00002s | 0.00002s | 0.00002s |
| 10,000 | 0.0021s | 0.0018s | 0.0018s | 0.0019s |
| 500,000 | 0.0250s | 0.0251s | 0.0252s | 0.0251s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** 1 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0001s | 0.0001s | 0.0001s | 0.0001s |
| 10,000 | 0.0100s | 0.0100s | 0.0099s | 0.0100s |
| 500,000 | 0.1028s | 0.1073s | 0.0976s | 0.1026s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** 2 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0001s | 0.00008s | 0.0001s | 0.00009s |
| 10,000 | 0.0087s | 0.0087s | 0.0109s | 0.0094s |
| 500,000 | 0.0910s | 0.0973s | 0.0926s | 0.0936s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** 3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0002s | 0.00008s | 0.0001s | 0.0001s |
| 10,000 | 0.0117s | 0.0082s | 0.0103s | 0.0101s |
| 500,000 | 0.0930s | 0.0866s | 0.0960s | 0.0919s |