CSCI 2720
Data Structures

Project 3

Sorting Algorithms

Fall 2016

Team Members

1. Name	Andrew Durden	UGA ID	811729528	CRN 25654
2. Name	Carter Hart	UGA ID	810158526	CRN 25654
3. Name	Connor Kirk	UGA ID	811462479	CRN 25654

1. Introduction

The objective of this Project was to empirically measure the complexity of sorting algorithms described by math in the field of Computer Science. By using these sorting algorithms on sufficiently large or small data sets, we can understand Big-O notation as a real phenomenon as well as observe the strengths and weaknesses of each algorithm with respect to time and space under various conditions.

The Experiment involves 6 sorting algorithms:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

In the next section we will explore the theoretical pros and cons of these sorting algorithms. For the experiment, we run 3 different data sets (sorted, random, or reversed) on each algorithm and calculate the elapsed time with 2 time stamps. This is repeated for different sample sizes (10, 100, 500, 1000, 10000, 20000, 100000, and 200000) to observe how these different algorithms scale with input size.

2. Theoretical Analysis of Sorting algorithms

Sorting algorithms are described by their "Big-O" complexity, which is a measure of the rate of grown in complexity on input of size n. For example, a simple linear search of a list to find if an element is a member of the list is considered O(n). You might think that best case, we find that the element is the first member, which would only require 1 check, not n checks. However, O(n) is the upper bound of complexity, meaning we have to consider the worst case in order to understand an algorithm's Big-O complexity.

Bubble Sort, Insertion Sort, and Selection Sort are all considered to be $O(n^2)$. This is because their sorting strategies do not take advantage of higher level data abstraction, but instead simply compare each element to every other element in order to figure out their order. For each element from 1 to n, we have to compare with each element 1 to n, and so their complexity is described as $O(n^2)$. However, this analysis is simplified because these algorithms still use small shortcuts to become more efficient under certain special cases. For example, Bubble Sort is $O(n^2)$ in its worst case, but best case (already sorted), no swaps have taken place and so fall under O(n), much better than Quick Sort, who's best case is O(n log n) complexity.

Quick Sort, Merge Sort, and Heap Sort all have average case O(n log n) complexity, which makes them more desirable in most cases. This efficiency comes at a cost,

however, because they exploit data structures or logical abstraction to perform quickly, however this typically takes more memory. For example, Merge Sort involves splitting up an array into bits and then merging them together in order. However, in the best case scenario, this takes an addition n places in memory. In contrast, Heap Sort does not use more space, but the list must already be a heap data structure which comes with computational overhead.

3. Experimental Setup

a. **Machine specifications**

Machine: 2015 Macbook Pro
Processor: 2.5 GHz Intel Core i7
Memory: 16 GB 1600 MHz DDR3

b. **Generating input**

Each test input was generated with a gen.cpp file, which takes several arguments including the number of elements in an output file, the name of that file, and a flag for the data relation (sorted, random, or reversed). We used this program to create the 3 files: inorder.dat, random.dat, and reverse.dat

After generating a each file with 200000 numbers inside, we used an ifstream to read each line of the file and parse the integer from that line into an array of equivalent size.

c. **Measuring time and gathering data**

We measured time using the clock_t data type. Before a sorting algorithm is run, we get the current time using the begin() method in the standard library, and repeat the same timestamp one the algorithm is over. We then store the difference into a double as time elapsed and print it out. We ended up taking the average of 3 executions and averaged them in an excel spreadsheet.

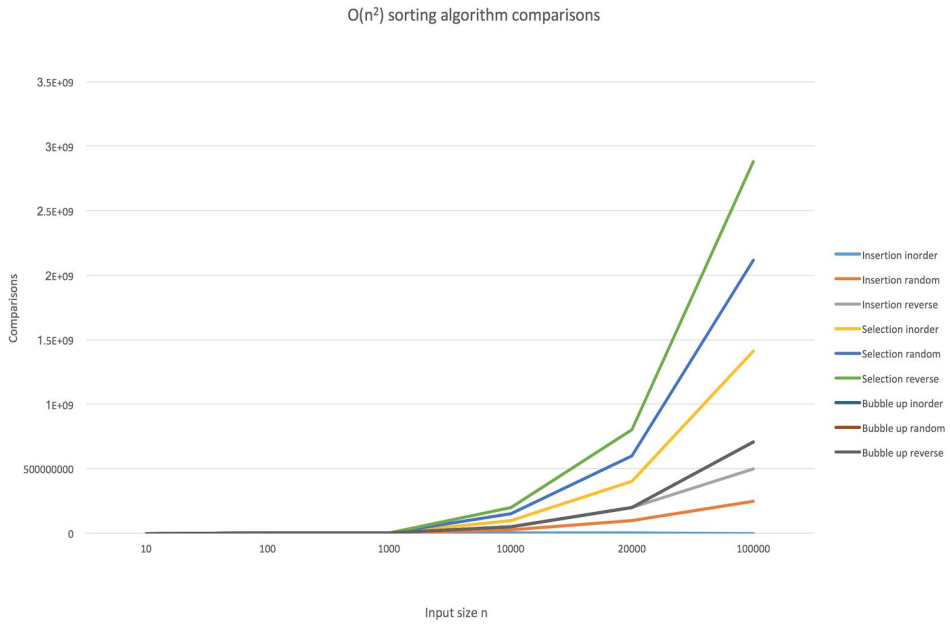
d. **Additional Memory Usage / Data Structures**

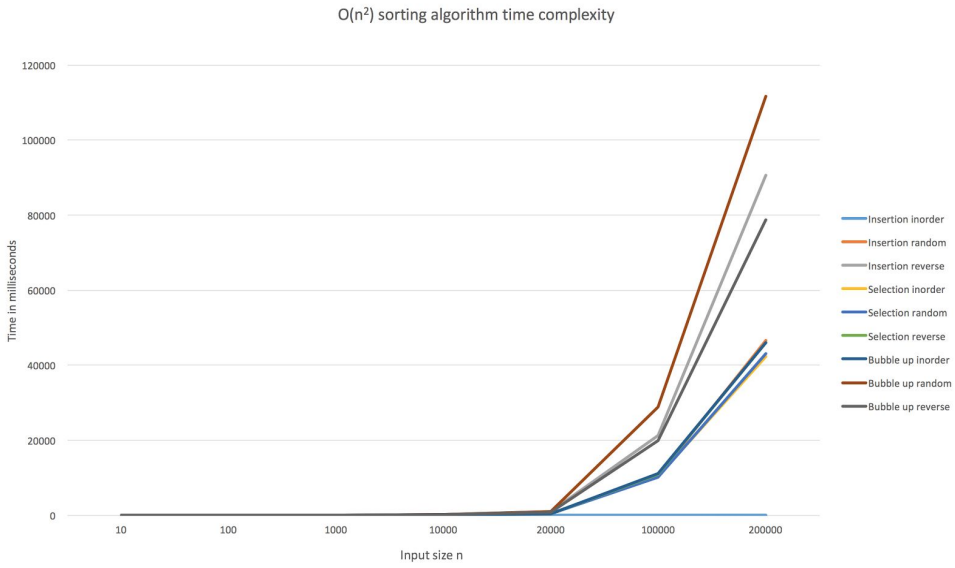
For the Heap Sort algorithm, the array as a head data structure is a precondition. This adds an additional O(n) time complexity to the algorithms, but does not take up any extra space. Merge sort, however, takes up an addition O(n) spaces in memory due to the need to merge every element in an array to a new array.

4. Experimental Results

4.1 O(n^2) Sorting Algorithms

As we can see from the data, large data sets grow in complexity very fast. The data suggests that selection sort performs worst on almost all accounts. Our implementation for insertion sort, however, simply inserts the next member, making it run at O(n) complexity when the data is already sorted (seen below as a flat blue line). Below we see the same complexity in terms of number of comparisons

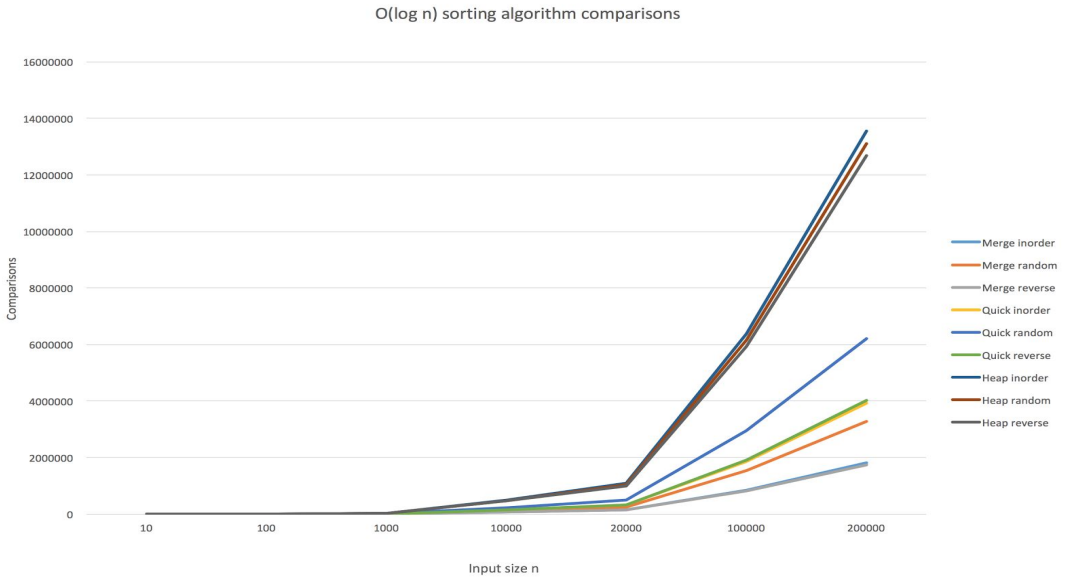


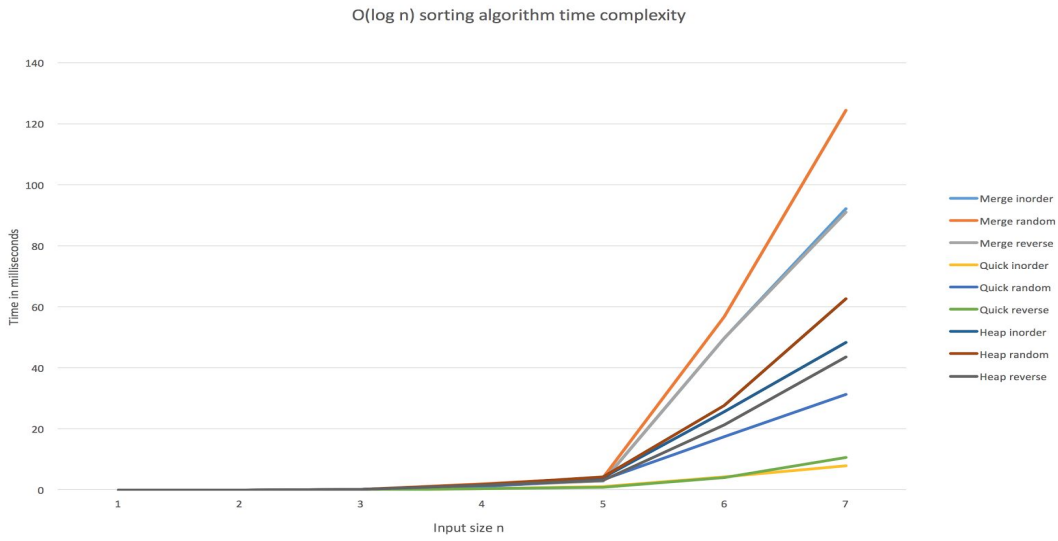


The poor results of selection sort match theoretical predictions exactly, since selection sort requires a finding the minimum of the remaining n elements, requiring a comparison every iteration, even when no swap takes place.

Bubble Up and Insertion reverse suffer from very similar issues, which is that if the data is reversed, the algorithm must do a the maximum number of comparisons under this condition. For bubble sort, this means swapping every iteration from 0 to n, then to 1 to n and so on until n-1 to n. For insertion sort, finding the next lowest value at the end of the array means traversing n spots to place it, exactly the opposite case from the sorted input.

4.2 O (n log n) Sorting Algorithms





Comparison complexity for heap sort is very high due to the requirement to buildheap from from a random array. If the array is already in order, building a heap actually un-orders the array quite dramatically before resorting, making it relatively inefficient for such cases.

5. Concluding Remarks

It is important to note that the theoretical analysis of the algorithms are computed in terms of Big O, which describes the limiting behavior of the algorithm when the input size approaches infinity. It does not take small input sizes into account, where the number of steps between algorithms are similar. Instead it looks at larger input sizes, where there is a large variance in the number of steps. The result of this is the Big O analysis being most reliable with large inputs.

From our experimentation, it is obvious that there is a difference in execution time between the $O(n^2)$ and $O(n \log n)$ sorting algorithms, as seen below. The difference in time increases as the size of the array increases, in accordance with the definition of Big O. However our experiment shows that not all algorithms with the same Big O are the same. For example quick sort was by far the fastest $O(n \log n)$ sorting algorithm for all orders of data, and there was a difference in time in the $O(n^2)$ algorithms as well. From this we can conclude that while Big O analysis does a good job at predicting execution times among algorithms of different Big O's, it is useless for predicting the execution times among algorithms with the same Big O. As far as machine independent algorithm efficiency analysis goes, Big O is still the best.

From our experimentation, insertion sort was the fastest sorting algorithm when working on data that was already sorted since it only iterates through the array once. Quicksort was the fastest when working with a large array of values and was even comparable to the other algorithms with a best case of $O(n)$, even on small arrays. In terms of number of comparisons, it was shocking to see how few merge sort used considering it was the slowest of the $O(n \log n)$ algorithms.

Please view the following diagrams from the data we have collected.

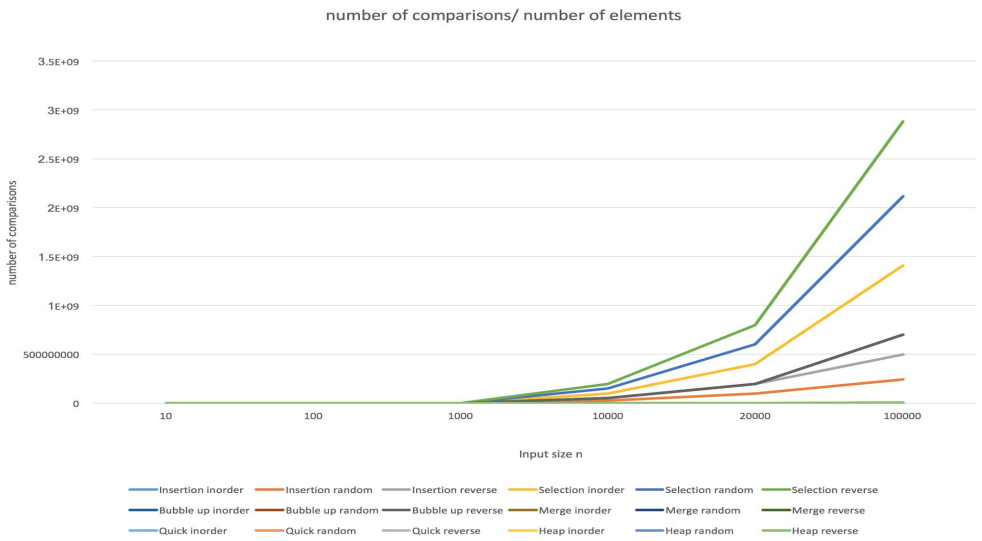


Table 1: Running time vs. number of elements

		Number of elements							
Sort Type	**data type**	**10**	**100**	**500**	**1000**	**10000**	**20000**	**100000**	**200000**
Bubble	Inorder	0.004	0.028	0.412	1.246	110.256	464.408	11218.4	46013.4
Bubble	Random	0.003	0.055	0.63	1.625	235.443	975.016	28841	111623
Bubble	Reverse	0.003	0.042	0.603	1.79	194.242	726.241	19898.6	78737.9
Insertion	Inorder	0.004	0.007	0.016	0.026	0.077	0.133	0.686	1.424
Insertion	Random	0.004	0.031	0.577	1.565	109.169	428.916	10664	44613.1
Insertion	Reverse	0.005	0.059	1.21	3.224	230.231	878.774	21472.9	90534.6
Selection	Inorder	0.004	0.028	0.569	1.587	107.406	419.753	11028.6	43261.9
Selection	Random	0.003	0.034	0.542	1.535	107.299	431.106	10829	43336.5
Selection	Reverse	0.004	0.031	0.405	1.469	108.719	481.84	11580.6	46098
Heap	Inorder	0.005	0.023	0.077	0.172	1.575	4.053	25.649	48.425
Heap	Random	0.005	0.023	0.09	0.187	1.815	4.3	27.722	62.625
Heap	Reverse	0.004	0.02	0.71	0.157	1.612	3.186	21.322	43.519
Merge	Inorder	0.007	0.04	0.129	0.206	1.446	2.884	50.507	91.958
Merge	Random	0.008	0.047	0.132	0.255	1.964	4.079	51.049	111.821
Merge	Reverse	0.007	0.042	0.102	0.169	1.477	3.222	44.039	89.249
Quick	Inorder	0.004	0.009	0.029	0.038	0.374	0.699	4.009	8.413
Quick	Random	0.004	0.019	0.066	0.126	1.148	2.299	13.595	28.937
Quick	Reverse	0.004	0.009	0.02	0.035	0.382	0.733	4.442	8.267

Table 2: Comparison count vs. number of elements

		Number of elements					
Sort Type	data type	10	100	1000	10000	20000	100000
Insertion	inorder	9	99	999	9999	19999	99999
Insertion	random	28	2525	245937	24538729	99903238	245387290
Insertion	reverse	45	4950	499500	49995000	199990000	499950000
Selection	inorder	90	9900	999000	99990000	399980000	1409965408
Selection	random	135	14850	1498500	149985000	599970000	2114948112
Selection	reverse	180	19800	1998000	199980000	799960000	2880705187
Bubble up	inorder	45	4950	499500	49995000	199990000	704982704
Bubble up	random	45	4950	499500	49995000	199990000	704982704
Bubble up	reverse	45	4950	499500	49995000	199990000	704982704
Merge	inorder	19	356	5044	69008	148016	853904
Merge	random	22	542	8711	120414	260836	1536355
Merge	reverse	15	316	4932	64608	139216	815024
Quick	inorder	55	858	11053	149055	318096	1862156
Quick	random	69	1277	18073	237397	509623	2950985
Quick	reverse	66	911	11559	154071	328113	1912169
Heap	inorder	99	2318	36604	503704	1086500	6369812
Heap	random	91	2182	34824	482685	1044996	6153968
Heap	reverse	80	1988	32772	461738	1004332	5937451

References:

[1] All of the code for the sorting algorithms was adapted from *C++ Plus Data Structures* by Nell Dale.