Gentry Atkinson
CS5329 Fall 2019
Programming Project Report: Problem 1

Introduction

Sorting algorithms are easily broadly classes by algorithmic complexity. But often times the performance of specific algorithms will not match those expectations in all cases. Algorithms which scale well may perform suboptimally on small data sets. Likewise some algorithms have unexpected best and worst case scenarios that cause them to run in wildly different execution times on datasets of the same size. Because of this it is important to test our algorithms (both sorting and otherwise) on actual data sets rathering than relying solely on mathematical analysis of the time and memory complexities.

This project has implemented a simple testing platform which will constructively test several sorting algorithms on generated data sets of several sizes and compositions. Several tests are run iteratively applying the sorting algorithms to the data sets in order to normalize the variations that can occur in operating system scheduling. These results are then organized in order to make effective assessments of the real-world performance of the algorithms

Data

9 integer data sets were automatically and randomly generated and written to text files. This allowed the 6 sorting algorithms to all be run on identical values. The data sets were generated in 3 different sizes and 3 different compositions. The 3 sizes that were chosen are 100; 1,000; and 10,000. These values cover a range which allows the scalability of the algorithms to be assessed but still aren't so large that they would take an excessive amount of time on a consumer PC.

The compositions of the datasets are fully random, random without duplicates, and nearly sorted. The random data sets consist of n values chosen uniformly at random from the range of 0 to n. The random without duplicates sets consist of all values from 0 to n arranged into a fully random order. This is done by making 2n swaps of random pairs of values in the set from 0 to n. Finally, the nearly sorted sets are produced in a similar fashion but rather than 2n swaps, only n/20 swaps are made. This insures that most values are in their proper positions. The randomness is introduced using the rand() function of the C std library. This function chooses pseudo-random values with equal likelihood over the range of integers.

3 visualizations are provided below in order to help understand the composition of the various datasets. They are plottings of the raw points from the 3 short datasets. The longer datasets have similar appearances but are too cluttered to be as easily interpretable.

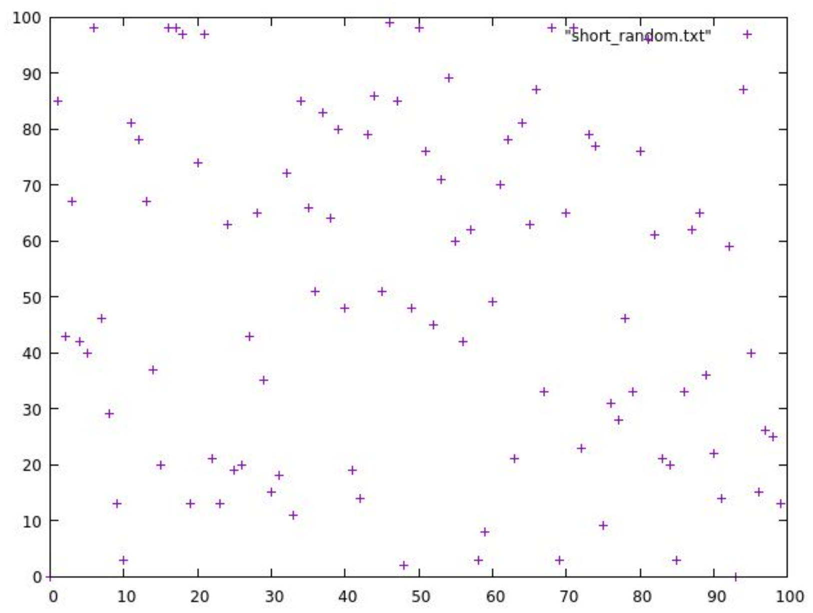


Figure 1: the 100 value random dataset.

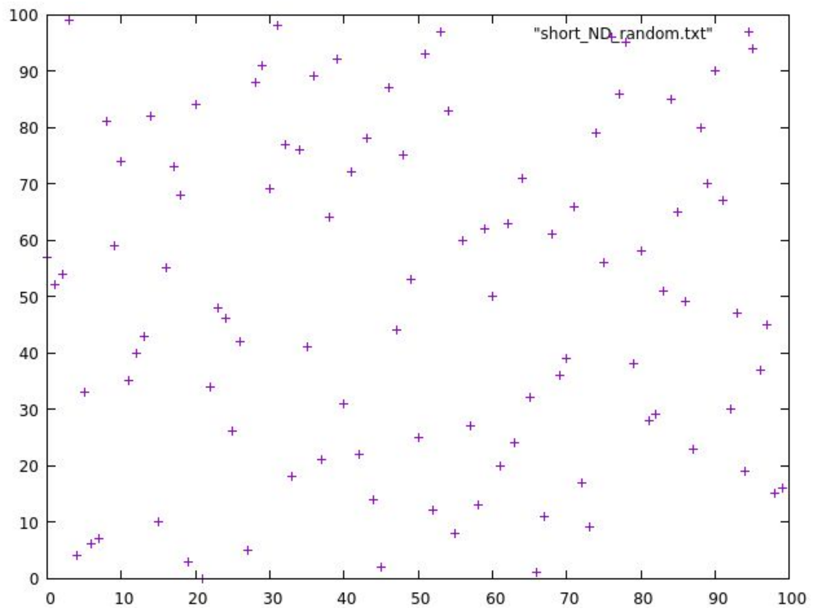


Figure 2: the 100 value random-without-duplicates dataset.

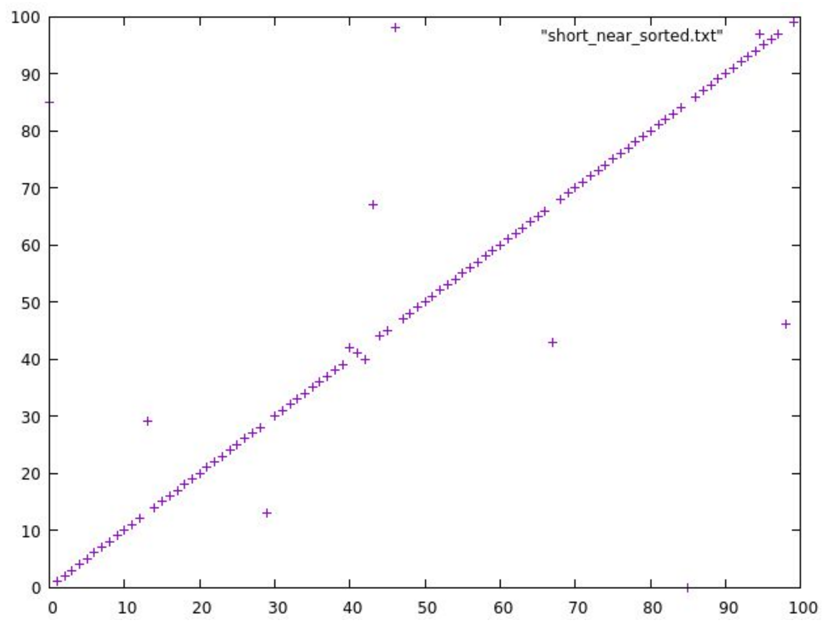


Figure 3: the 100 value nearly sorted dataset.

Algorithms

A base of 4 algorithms were selected for comparison in this project. These are Insertion Sort, Merge Sort, Quicksort, and Heap Sort. Furthermore 2 adaptations of Quicksort are included: Randomized Quicksort and Hybrid Quicksort. This brings the total to 6 different algorithms that are tested.

Insertion sort is the only one of the traditional sorting algorithms included in this project. It works by iteratively shifting values to the "left" until they are in the correct position. It maintains a growing list of correctly sorted values at the left hand side of the list. Analysis shows that Insertion Sort is a $O(n^2)$ order algorithm.

Merge sort works by dividing the list into n sublists and recursively merging the sublists back together. It is a "Divide and Conquer" algorithm which is in the O(nlogn) order of complexity.

Quicksort also applies a divide and conquer strategy. The algorithm recursively partitions the list into sublists by repeatedly picking "pivots" and then arranging the values such that the sublists are purely greater than or lesser than the pivot value. The first variation of Quicksort that has been included in this project is Randomized Quicksort which selects the pivot value randomly from the sublist rather than routinely choosing the left- or right-most value as the pivot. This improves the performance on nearly sorted lists. The second variant is Hybrid Quicksort which calls Insertion sort to sort a sublist whose size is less than 32. This takes advantage of Insertion Sorts relatively speedy performance on smaller sets. All variants of Quicksort are O(nlogn) in the average case. However, non-randomized variants degrade to $O(n^2)$ on near-sorted lists.

Finally Heap Sort applies the heap data structure to list sorting. A heap is a tree that guarantees that no subtree of a node contains a value greater than (or less than) the node value. This guarantees that the root node of a heap is always the greatest (or least value) of the set. Heap Sort iteratively extracts the root value of the tree and then "re-heapifies" the tree. It is also a O(nlogn) class algorithm.

Results

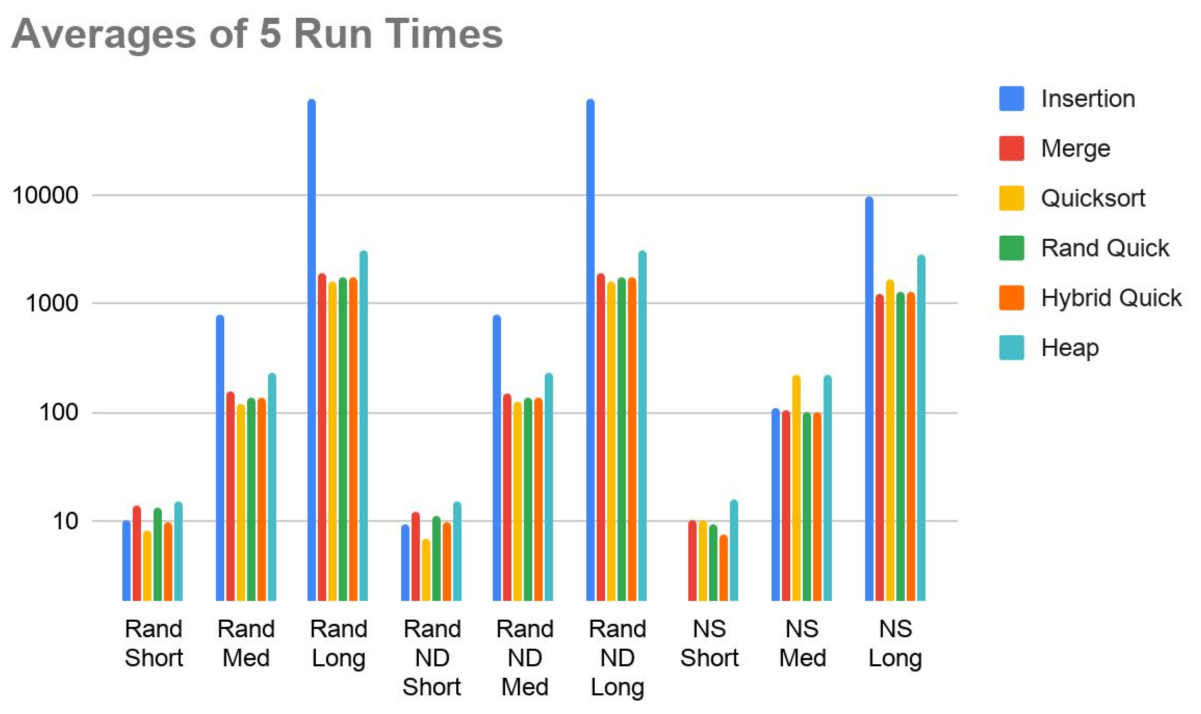


Figure 4: The runtimes of the six algorithms on the 9 datasets. The values presented in this graph are averages over 5 runs. ND is an abbreviation of No Duplicates. NS is an abbreviation of Near Sorted. All values are in microseconds.

Insertion	**Short Set**	**Medium Set**	**Long Set**
Random	10.2	791.8	76714.2
Random: No Duplicates	9.4	789.4	75293.4
Nearly Sorted	1.8	108.4	9579.6

Table 1: average runtimes of Insertion sort on the 9 datasets. All times are in microseconds.

Merge	**Short Set**	**Medium Set**	**Long Set**
Random	14	155.8	1906.4
Random: No Duplicates	12.2	151.8	1900.6
Nearly Sorted	10	105.8	1246.8

Table 2: average runtimes of Merge sort on the 9 datasets. All times are in microseconds.

Quicksort	**Short Set**	**Medium Set**	**Long Set**
Random	8.2	121.6	1571.4
Random: No Duplicates	7	124.4	1595.2
Nearly Sorted	10	218.4	1683.6

Table 3: average runtimes of Quicksort on the 9 datasets. All times are in microseconds.

Randomized Quick	**Short Set**	**Medium Set**	**Long Set**
Random	13	139	1740.6
Random: No Duplicates	11.2	138.8	1777.4
Nearly Sorted	9.4	102.4	1307.4

Table 4: average runtimes of Randomized Quicksort on the 9 datasets. All times are in microseconds.

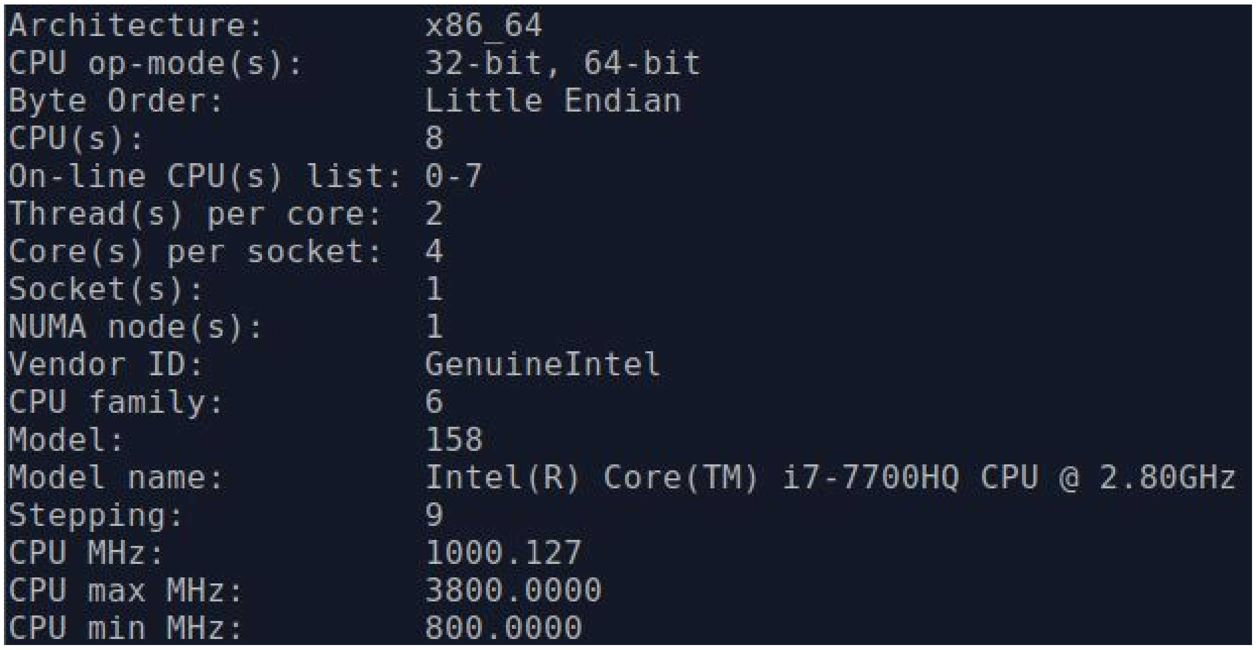
Hybrid Quick	**Short Set**	**Medium Set**	**Long Set**
Random	9.8	136.8	1745.2
Random: No Duplicates	9.6	138.4	1768.2
Nearly Sorted	7.6	101	1266.4

Table 5: average runtimes of Hybrid Quicksort on the 9 datasets. All times are in microseconds.

Heap Sort	Short Set	Medium Set	Long Set
Random	15.4	232.8	3090.6
Random: No Duplicates	15.4	229.4	3097.4
Nearly Sorted	15.6	217.6	2822.6

Table 6: average runtimes of Heap Sort on the 9 datasets. All times are in microseconds.

All of the tests were run on a consumer laptop with an Intel i7 8-core CPU running at 2.8 GHz. The tests were executed on the CPU of the machine rather than the GPU as no calculations were being made on the values in the lists. Since only comparisons were being made rather than calculations then there would be no advantage to running on a GPU.



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping:              9
CPU MHz:               1000.127
CPU max MHz:           3800.0000
CPU min MHz:           800.0000
```

Figure 5: a display of the output of the lscpu command on the test platform.

The values in the averaged runtimes cover range from 1.8 microseconds for the timing of Insertion Sort on the 100-value, nearly sorted data to 76,714 microseconds or the timing of Insertion sort on the 10,000-value, fully randomized list. Every dataset showed an increase in runtimes from 100 to 1,000 to 10,000 values. Insertion sort clocked the fastest average runtime on the 100-value lists at 7.1 microseconds. Hybrid Quicksort was the fastest on the 1,000-value lists with an average runtime of 125.4 microseconds. Hybrid Quicksort was also the fastest on the 10,000-value lists with an average runtime of 1593.3 microseconds. The complete set of these results is summarized in the figures below.

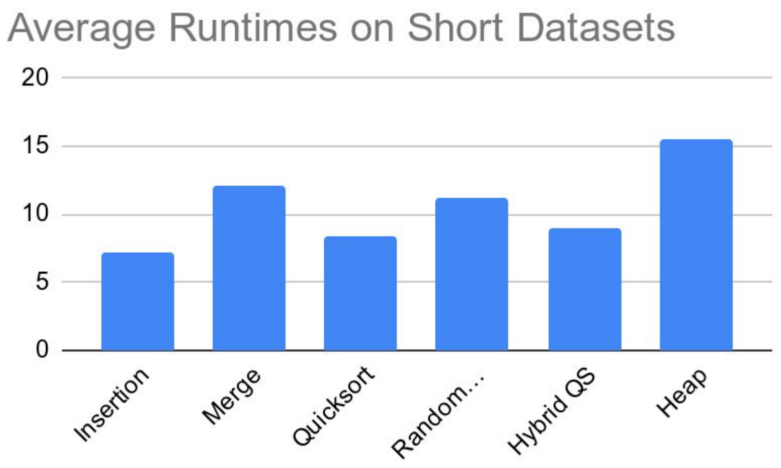


Figure 6: these values are all given in microseconds and represent the average time over 15 runs on 100-value lists with 5 being on fully random data, 5 on random no-duplicate data, and 5 on nearly sorted data.

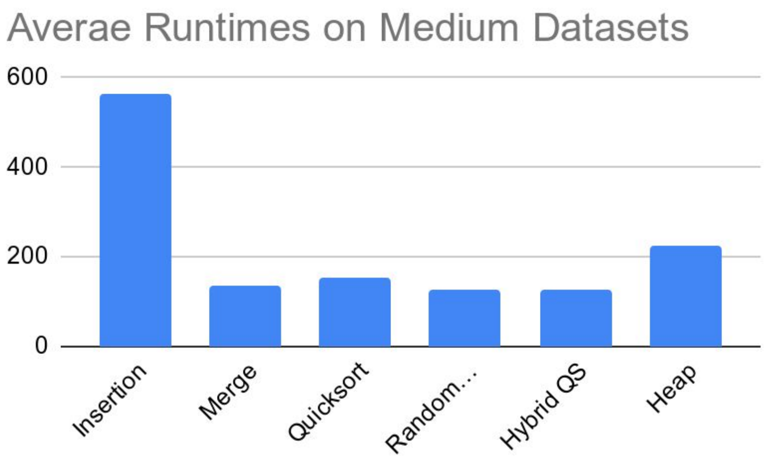


Figure 7: these values are all given in microseconds and represent the average time over 15 runs on 1,000-value lists with 5 being on fully random data, 5 on random no-duplicate data, and 5 on nearly sorted data.

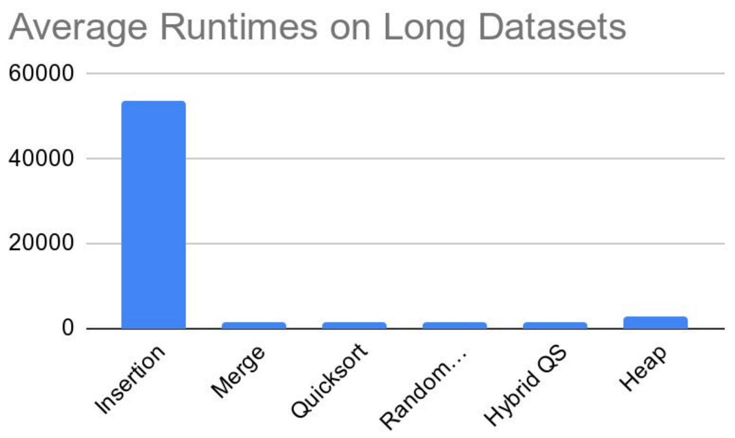


Figure 8: these values are all given in microseconds and represent the average time over 15 runs on 10,000-value lists with 5 being on fully random data, 5 on random no-duplicate data, and 5 on nearly sorted data.

Discussion

The collected data largely confirms what an analyst would assume about each algorithm. Insertion sort runs the most quickly of the 6 on the small data set and the slowest of the group on the larger datasets. It is therefore entirely reasonable that the Hybrid Quicksort algorithm, which calls Insertion Sort to sort sublists with a size smaller than 32, is the fastest of the 5 remaining algorithms. This is because it applies each algorithm in the condition that it performs most strongly in.

It is interesting to notice that the increase in runtime is much lower going from the n=100 sets to the n=1000 than simple analysis would suggest. We would expect to see a $O(n^2)$ algorithm increase it's runtime by roughly 100 fold and the O(nlogn) algorithms increase by roughly 33 fold on the same range. However, it is important to remember that there is an overhead to function calls that is not necessarily reflected in algorithmic analysis. This is what happens when code is executed on actual machines rather than in the idealized consideration of textbooks. Insertion Sort is therefore actually a $O(an^2 + bn + c)$ algorithm and the others O(a(nlogn) + c) but the solution is simplified to better divide algorithms into classes. We can solve these equations with our new data to calculate exact values for each algorithm:

Insertion Sort	$f(n) = 0.0005n^2 + 0.03n - 1.11$
Merge Sort	f(n) = 0.02nlogn + 12.4
Quicksort	f(n) = 0.02nlogn + 36.3
Randomized Quicksort	f(n) = 0.02nlogn + 6.6
Hybrid Quicksort	f(n) = 0.02nlogn + 6.4
Heap Sort	f(n) = 0.03nlogn + 1.4

Conclusion

This project has been able to confirm the conventional wisdom of sorting algorithms. Insertion sort is easier to implement than any of the other 5 algorithms and faster on smaller datasets. However the others capitalize on clever "divide and conquer" methodologies which allow them to sort larger sets much more quickly. Our calculations in the Discussion section confirm the empirical evidence presented in the Results section: Hybrid Quicksort accomplishes the lowest average runtimes by applying divide and conquer when it's most useful and by defaulting to traditional sorting when divide and conquer is no longer useful. We have also been able to produce a set of mathematical models to predict the runtimes of the 6 algorithms on datasets of arbitrary size.