CSCE 221 Cover Page
Programming Assignment #3
Due Date: Wednesday October 30, 11:59pm
Submit this cover page along with your report

First Name: Ty Last Name: Conway UIN: 626002786

Any assignment turned in without a fully completed coverpage will receive ZERO POINTS.
Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (URL)	Other
1.	1. Yahui	1.	1. bigocheatsheet.com	1. Heaven
2.	2. Charlie	2.	2.	2.
3.	3.	3.	3.	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: 10/28/19

Printed Name (in lieu of a signature):

Introduction

There are certain fundamental processes which the computing industry routinely finds itself performing, one of which is sorting. There have been numerous strategies devised to achieve a sorted list of items as quickly as possible. Surely, with the rise of Big Data and the world's ever-increasing information, sorting *faster* is of greater demand than ever. This publication reports the results of a project conducted on three sorting strategies: selection sort, insertion sort and heap sort. These sorts will be implemented in three respective priority queues, have certain quantities of numbers inserted and removed, then have their performances in these areas graphed and discussed. Both a description of the experiment's structure and a thorough theoretical analysis of the sorts will precede the results however.

Experimental Setup

The experiments were tested on a Windows Surface Pro Laptop with 4 GB RAM and an Intel Core i506300U CPU at 2.40GHz.

The two experiments followed this procedure: (1) A script initializes the data structures, (2) 10,000 numbers are inserted (or in the case of the second experiment inserted and subsequently removed, (3) runtime is logged to a .csv file at intervals of 10 operations. The times were measured from the beginning of the 10,000 operations until

the end for the insert-only experiment. For the insert-and-remove experiment, successive trials were timed for a push of 10 numbers, then a push of 20, etc. until 10,000. This allows for a more complete picture of inserting and removing together.

Theoretical Analysis (insert and sort)

Unsorted Priority Queue

The unsorted priority queue insert is perhaps the simplest operation discussed in this project: it appends the inserted item to a list. This is where its name is derived: the list within the implementation is not sorted and does not ever become so; only upon removal are items sorted. This is known as *selection sort*, because the queue selects the items as it goes. The advantage is obvious: O(1) complexity for insertion.

The unsorted PQ's disadvantage lies on the removal, as stated previously. Needing to search through all N elements to find the minimum upon each remove operation will yield O(n). The two together, insertion and remove, when performed on N number of items will produce a sorted list of those items at the overall complexity of $O(n^2)$.

However, a key distinction ought to be made: the unsorted PQ will check *every* element, *every* time, lending to precisely N operations *every* time. This will be contrasted with the sorted PQ below.

Sorted Priority Queue

The sorted PQ appears by technical means to be the opposite of its unsorted counterpart. By implementing an *insertion sort* strategy where the insert method begins traversing the list, comparing each element to the incoming item until its appropriate spot is found, the queue resolves an insert in O(n) complexity, or the length N of the list.

The beauty in this implementation is the removal strategy: the smallest element is the first element, making retrieving it a O(1)-sized task. When sorting a list however, the gain is overshadowed by the insert process and will produce the same $O(n^2)$ complexity, via O(n) * N operations.

This queue maintains a slight advantage over the unsorted version. The sorted PQ technically inserts at O(n) worst-case, but on average the complexity is somewhat less. Indeed, this follows intuitively since the sorted PQ will only sort *until* the element's spot is found, not continuing to compare each of the N elements. Whereas the unsorted PQ will check all N regardless even in the best-case, the sorted PQ may well outperform the unsorted.

Heap Priority Queue

The heap PQ has neither a constant insert nor a constant remove. Both operations are instead less complex than O(n). For insert, an item is added to the end of the array (which will be assumed resizeable) and then a method is called to 'fix the heap'. Likewise for remove, the top of the heap is plucked off and the last element

becomes the first element before calling ‘fix the heap’. Both the insert and remove functions minus ‘fix the heap’ would yield a constant runtime. It is in fixing the heap that the process begins to grow less performant--yet, there is a favorable bound to this decrease in performance. When fixing a heap in which only the first or last element has changed, the worst-case has an element traveling from the bottom level to the root, *upheaping*, or swapping places with its parent node, each time. The maximum levels it could climb is the height of the tree, or *log n*. This is good for the heap, since in sorting N elements, the worst it could perform would be N * O(log n) or O(n log n).

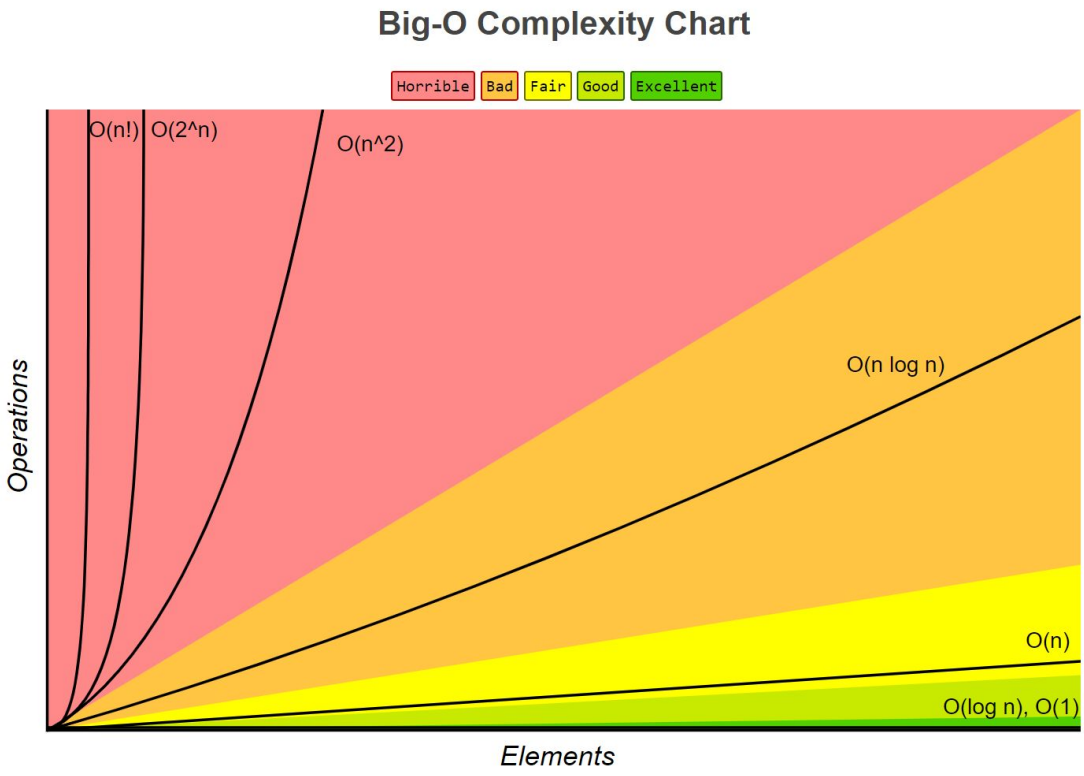
The plot thickens though, since on average the heap PQ does not perform *log n* upheaps with each operations. In fact, the average is something closer to constant time, performing some few upheaps each time, independent of the height of the tree. It can then be reasonably expected to observe an O(n) runtime for sorting with a heap PQ.

Experimental Results

An Overview of Algorithmic Complexity

The primary measure of comparison in this examination is Big-Oh complexity, or the performance of a routine at high numbers of input. As we approach such large numbers, trends in their plotted functions develop and allow for straightforward analysis of who’s producing results faster than the other. Courtesy of bigocheatsheet.com, the following chart has been included for reference (and the report itself will refer back to this chart):

Figure 1:



The subsequent discussion will be centered on three of these function sets: $O(n^2)$, $O(n \log n)$ and $O(n)$; with each in different conventional categories of performance: horrible, bad, and fair, respectively.

Inserting

Figure 2 depicts the reality of the insertion sort (Sorted): it pays its costs upfront. That gentle curve up of the red line is characteristic of $O(n^2)$, as it grows quadratically. This behavior was predicted by needing to walk through a list, taking $O(n)$, and needing to do it n-times, totaling at $O(n^2)$.

Figure 2:

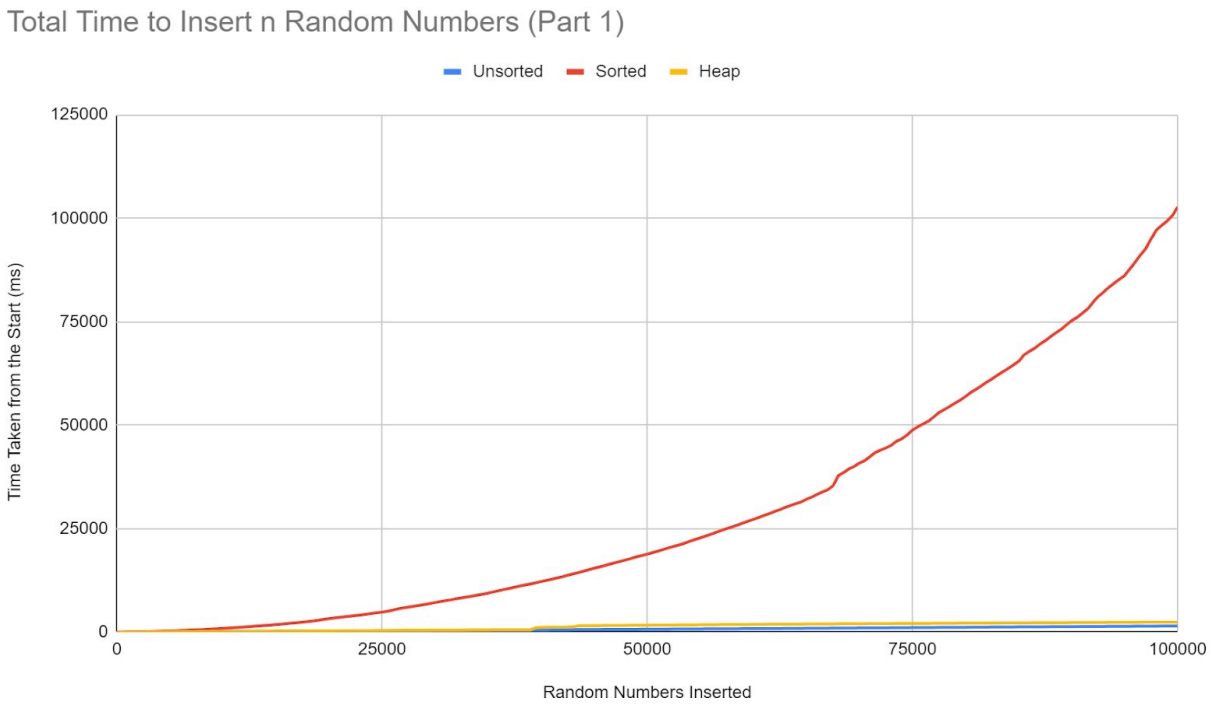


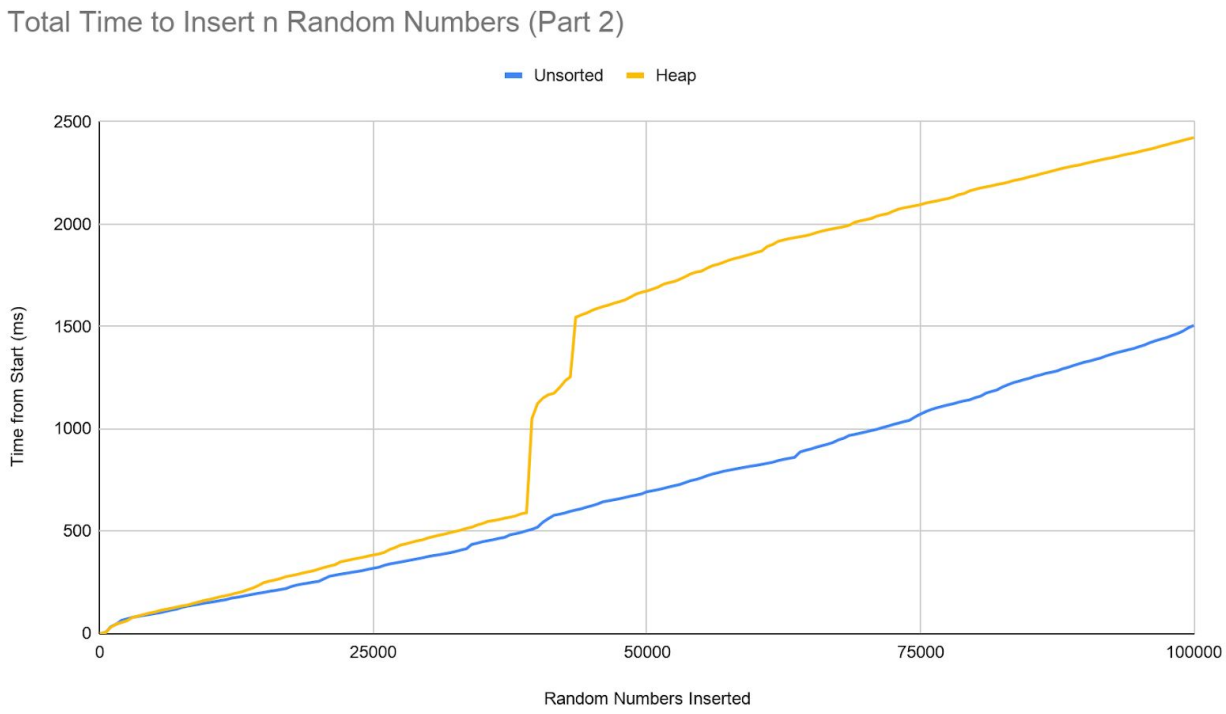
Figure 2 fails to provide much comparison on the Unsorted or Heap however, so Figure 3 zooms in. As pictured, the Unsorted in blue grows at an expected O(n), since it needs merely append a number to a list O(1) n-times. This is how selection sort pays later, saving time now as it seems to get ahead far ahead of insertion sort. It will be shown that this is only one piece of the picture and that the hare did not win the race.

The graph for Heap yields some concerns, largely what on earth that grotesque hump halfway could indicate. When examining the underlying implementation, it could

be hypothesized that the vector storing the heap is needing to resize at that point in the program, greatly affecting the curve. Further experimentation utilizing a statically-declared array would likely clear the air, and should be considered moving forward. The data represented is not useless though, as the curves before and after are largely untouched. Moving the latter half of the yellow curve down to where it would have been, an adjustment so to speak, suggests something very similar to and just above the performance of O(n), namely O(n log n). The theoretical analysis holds for inserting on all three accounts.

Insertion, however, is only the first test of the experiment.

Figure 3:



Sorting

The primary test is that of sorting: inserting numbers and removing them in a sorted order. Figure 4 shows the results of sorting sets of numbers ranging in size from 0 to 100,000, with times taken for the total time to sort the set.

Unsorted finally pays its dues, and in excess of Sorted, despite both having a theoretical $O(n^2)$ complexity. What could contribute to this discrepancy? There are at least two possible considerations. (1) As discussed above, insertion sort does not have to traverse the whole list every time, but stops when the slot needed is found. Selection sort on the other hand, walks through and does all the work all the time. This added work no doubt compounds over time. (2) The Sorted PQ in the experiment also employs a useful check: if the inserted item is greater than the last item in the sorted list, the item is appended and the function returns, effectively *skipping* the need to walk through the list. This would further enhance the Sorted's performance over the Unsorted.

In the end, both of these implementations pale in comparison to the yellow line running along the bottom, which is seen better in Figure 5.

Figure 4:

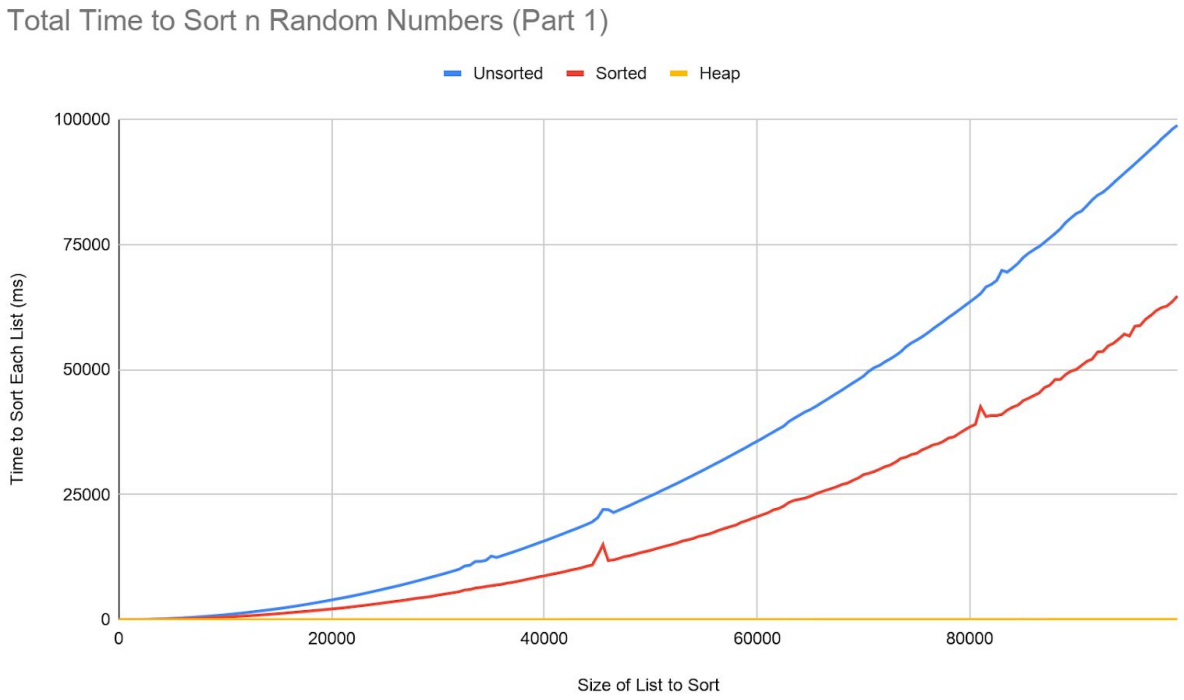
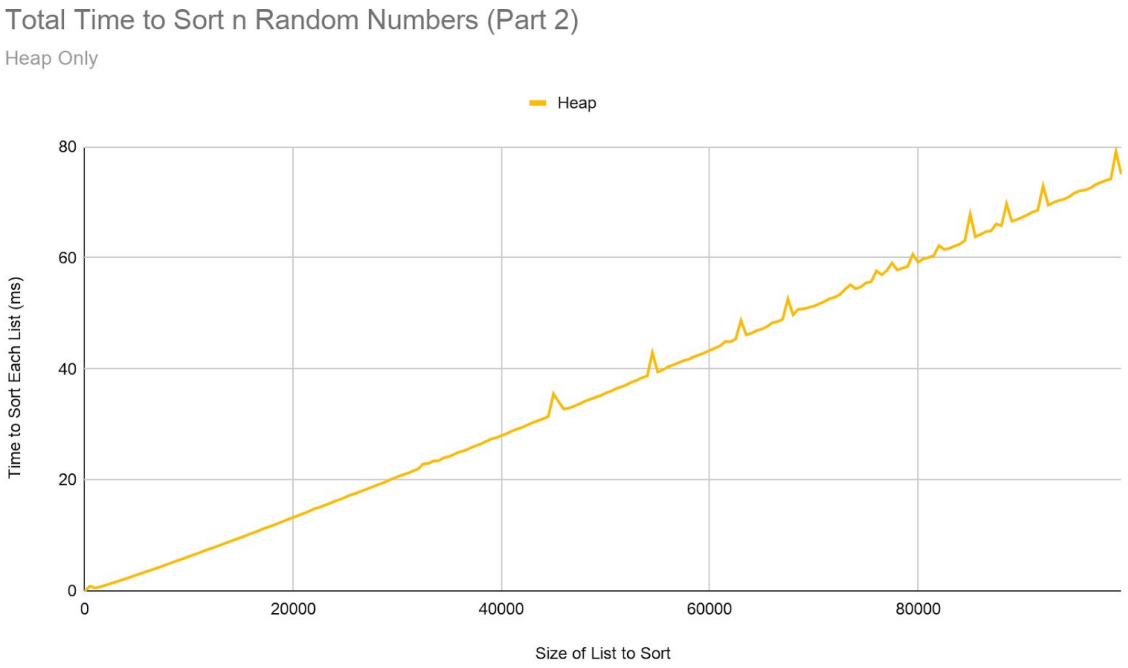

Total Time to Sort n Random Numbers (Part 1)
Unsorted
Sorted
Heap
100000
75000
50000
25000
0
Time to Sort Each List (ms)
0
20000
40000
60000
80000
Size of List to Sort

Figure 5:



Here the heap demonstrates its tell-tale O(n log n) complexity by comparing its slope to Figure 1. There are a few bumps in the line, especially towards the end. This is likely from those sets containing a random sequence that tended towards heap sort's worst case: numbers in decreasing order. A further examination of this phenomenon follows.

Sorting a Reverse-Sorted Set

If algorithms are to have their resilience truly tested, then a look at their worst cases would offer good information. What might be the worst case for a sorting algorithm? Perhaps a set which is sorted in reverse order, ensuring that each insert will require the full use of algorithmic muscle.

Figure 6:

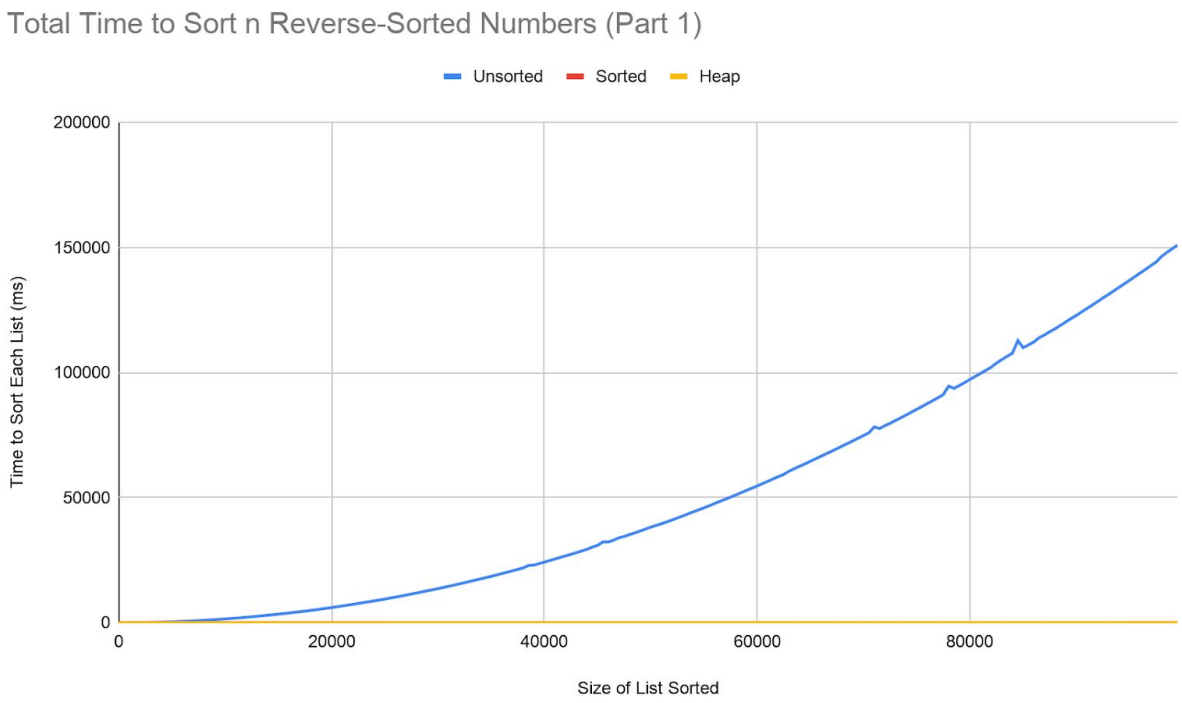
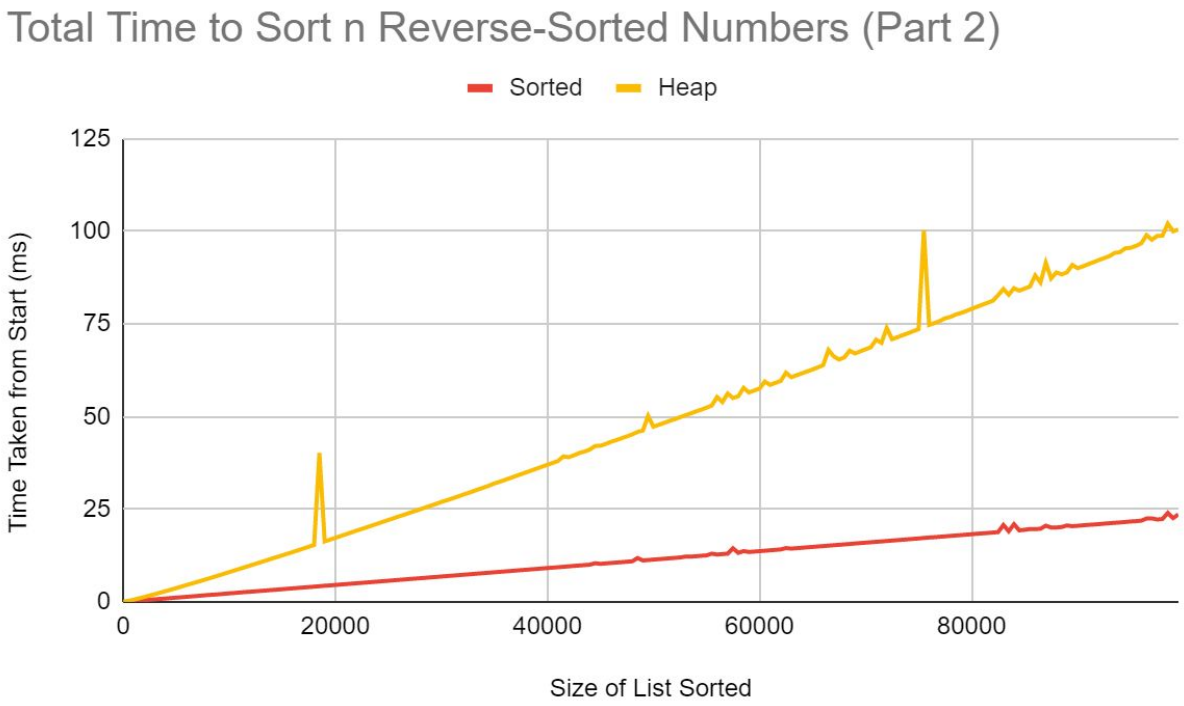


Figure 6 shows a lonely blue line, reaching for the sky--just as a villain caught by the gun-slingin' hero of a western movie might do upon surrender. That is because selection sort did what it does every time: walk the whole length of the list each and every function call. Do note the absence of its normally $O(n^2)$-sorting partner named insertion sort. As it turns out, a reverse-sorted list happens to be the Sorted's best case.

Figure 7 shows that Sorted outperforms the Heap, though a reverse-sorted list does not bode well for the Heap. If the minimum is added to the end of the Heap each time, then the maximum number of upheaps will be required upon insertion. While this decrease in performance is palpable, it is not fatal: Heap still performs at its steady O(n log n).

Why then should Sorted perform so fantastically? Because if each number being inserted is the new minimum, then it will go in at the beginning of the list, resulting in a liquid and buttery smooth O(1) per insertion, or O(n) for n insertions.

Figure 7:



The same test was conducted for a pre-sorted list but the results were almost identical, negating the need of another chart. Again, Sorted's smart-check of an inserted item being greater than the last element resulting in an append leads it to victory there again. That victory is for purely theoretical reasons however because sorting a sorted list yields no real-world value.

Conclusion

If given the choice for a particular purpose, none of these sorting algorithms should be used as there are faster available. But if limited to these three, then the heap sort is to be preferred. While its performance depends upon the input, as shown in the case of a reverse-sorted list, such a list could be sorted by trivially reversing it in O(n) time. The heap sort clearly outperforms selection and insertion for the general-use case of a random set of numbers, $O(n \log n) < O(n^2)$.