Who sorts the sorters?

Alexander R. Frieder*

March 31, 2017

1 Introduction

Any introductory programming class will cover the basics of different sorting algorithms. Sorting algorithms are the perfect introduction to algorithmic analysis for a multitude of reasons. First, there are straightforward proofs of the lower-bound in terms of performance, thus we can easily teach students that $O(n \log(n))$ sorts are optimal algorithmically. Second, the naïve sorts are fairly far from optimal and that is simple to show students. Finally, sorting is a deceptively easy task for humans. To expand on that final point, given a list of numbers, it is very easy for a human to order them correctly. Indeed, most introductory lessons will have the teacher prompting the student to describe exactly *how* they put them in order.

The important takeaway from the lesson is that sorting is not a trivial process and it must be done through some algorithm. **It does not suffice to take numbers and simply *put them in order***. The lesson will then proceed by walking the student through multiple algorithms and then displaying their performance on some large set of numbers, presenting how long each algorithm took. The teacher will then often conclude with some statement like "Thus we can see that merge sort is faster than selection sort". However, this directly contradicts the previous lesson: the teacher has taken performance numbers and somehow magically put them in order. Indeed, we need to use a sorting algorithm to determine which sorting algorithm is best. But which sorting algorithm is best for that task?

In this paper, we will review a variety of sorting algorithms and evaluate their performance on large randomly sorted arrays of numbers. We will then use the more general version of sorting to compare each sorting algorithm with each other to correctly evaluate which sorting algorithm is the best for sorting sorting algorithms. Finally, we will use the same algorithms to determine which sorting algorithm is best for determining which sorting algorithm is best for determining which sorting algorithm is best.

2 Sorting Algorithms

In this section we will review six different sorting algorithms that will be used throughout this paper and present, without proof[1], their algorithmic complexity. If you feel you are an expert sorter, feel free to proceed to section 3. I will not be offended. Not at all. I promise. These are not tears. I am just cutting onions.

2.1 Selection Sort

Selection sort is often the first sorting algorithm taught and is often the way humans intuitively sort. The algorithm behind selection sort is absurdly simple: find the smallest element left in the list, remove it, and put it in the output list. Repeat. Once the original list is empty, the output list will be sorted.

Since selection sort requires finding the max at each step, it runs in $O(n^2)$ time.

*email: alex.frieder@gmail.com

[1] Unless you count Wikipedia as proof.

Exercise 1

Teach your dog[2] how to perform selection sort.

2.2 Insertion Sort

Insertion sort is a similar idea to selection sort, but instead of putting in the effort to pull the correct element from the original list, we put in the effort to put a random element from the original list into the right location. For each element in the original list, we remove it, iterate through the output list, find the smallest element larger than it, and insert it immediately before that. Thus, at each step, the output list is sorted, and at the end, the original list is empty and the output list is sorted.

Since inserting an element may require iterating over the entire output list, insertion sort runs in $O(n^2)$ time.

Exercise 2

Put the words of this paper into alphabetical order using insertion sort[3].

2.3 Odd-Even Sort

Odd-even sort is a parallelizable version of the notorious and often politicized bubble sort[4]. In odd-even sort, we first iterate through the odd indices and for each element that is larger than the next element, we swap them. We then repeat for all the even indices. Finally, we repeat this pair of steps until no swaps are made. By definition, the list is now sorted. Since each step only swaps either exclusively odd or exclusively even indexed elements with its neighbor, the entire step can be completely parallelized.

Despite being parallelizable, odd-even sort is not more efficient than bubble sort and runs in a worst case of $O(n^2)$.

[2] Other pets may suffice for this exercise, but not cats. Cats are fundamentally incapable of learning algorithms.

[3] Ensure you have at least 15 minutes of free time before attempting this.

[4] https://www.youtube.com/watch?v=k4RRi_ntQc8

Exercise 3

There actually exists exactly one list of integers on which odd-even sort does not terminate. Find that list, treat the numbers as binary, read off the ASCII-encoded message, and continue your quest to find the Fountain of Youth.

2.4 Comb Sort

Comb sort is also a generalization of the notorious and often politicized bubble sort[5]. Comb sort is parametrized by a so-called shrink factor k, optimally around 1.3[6]. Comb sort begins by iterates over every pair of elements n away from each other, where n is the length of the list, and swapping the elements if they are not sorted relative to each other. Initially, that is only the first and last element. The distance is then set to n/k and the process is repeated. This process repeats, dividing the distance by k until the distance is 1, at which point we revert to bubble sort, swapping out of order pairs until the list is sorted.

Despite having faster run time than bubble sort, comb sort matches its complexity of $O(n^2)$.

Exercise 4

Do 30 push-ups, 30 sit-ups, and wall sit for at least 2 minutes.

2.5 Merge Sort

Merge Sort is the prototypical divide and conquer algorithm. First, recursively sort the first half of the list and the second half of the list, then merge them, by removing the smaller head of the two lists and inserting it into an output list. Repeat until both lists are empty. The output list is now sorted.

Merge sort takes exactly $\Theta(n \log(n))$ comparisons and is thus algorithmically optimal.

[5] https://youtu.be/MOzg_Cf4K4w?t=21s

[6] As cited from the Epic of Gilgamesh.

Exercise 5

Write an angry letter to the author explaining that the $O(n \log(n))$ lower bound only applies to comparison-based sorts and that some types can be sorted into linear time so therefore the author **is wrong**.

2.6 Quicksort

Quicksort takes, in the worst case, $O(n^2)$, but on average, $O(n \log(n))$.

Exercise 6

Enroll at Carnegie Mellon and take any of the following courses to learn in-depth about quicksort:

- 15-122
- 15-210
- 15-359
- 15-451

3 Methods/Implementation

For comparing sorting algorithms, the naïve method is to sort a large number of random lists of numbers and take the average time spent sorting. As mentioned originally, this method is antithetical to the entire purpose of sorting algorithms. But it will suffice as a first level approximation of the efficiency of sorting algorithms.

When comparing higher-order sorting algorithms we need to enforce a partial ordering. In other words, we need to define a $<$ operator on algorithms. There are multiple ways to do so, but we have chosen to use the most natural ordering: when comparing two algorithms A and B, we define $A < B$ to be true iff the total time spent sorting n random shuffles of some list L is less when sorting under A than when sorting under B. This is obviously a probabilistic total ordering, but in the limit, it is a deterministic total ordering. We will pretend our numbers are big enough to be considered "the limit".

We used $n = 3$ across all experiments. For sorting algorithms, we used $L = [1, 10^5)$ and defined $<$ as the normal integer $<$. For sorting sorting algorithms, we used $L =$ the list of all sorting algorithms discussed in section 2. As described above, we determined for two algorithms A and B if $A < B$ by running A on 3 random shuffles of $[1, 10^5)$ and B on 3 random shuffles of $[1, 10^5)$ and using normal integer $<$ on the time taken. Finally, for sorting sorting sorting algorithms, we use the same L, that is, all algorithms discussed above. We define $<$ in the logical expansion of the previous definition: $A < B$ if 3 runs of A sorting sorting algorithms takes less time than 3 runs of B sorting sorting algorithms.

These were all implemented in Python 3.6 but ported to and eventually run in PyPy2.7 v5.6.0 due to time concerns.

4 Discussion

All of the numeric results of these tests can be found in table 1 on the next page.

For the integer sorting algorithms, we see exactly the results usually presented in introductory classes and as expected. Quicksort, using probabilistic magic, is fastest, with merge sort following closely. Comb sort, using its heuristic advantage, is fairly fast, with the other $O(n^2)$ algorithms following behind. There is nothing too exciting here. The full ordering is quicksort $<$ merge sort $<$ comb sort $<$ insertion sort $<$ selection sort $<$ odd-even sort.

However, both the sorting sorting algorithms and the sorting sorting sorting algorithms have a different order from the integer sorting: merge sort $<$ insertion sort $<$ quicksort $<$ selection sort $<$ odd-even sort $<$ comb sort.

There are two interesting changes here. We can use merge sort as a good baseline since it does a relatively deterministic and static number of comparisons.

First, insertion sort is much much better for higher-order sorting than for integer sorting. For integer sorting, insertion sort is about 60x slower than merge sort. However, for sorting sorting,

Algorithm	0th Order Time	1st Order Time	2nd Order Time
Quicksort	0.06127 s	329.88 s	18.62 min
Merge Sort	0.06757 s	277.32 s	16.65 min
Comb Sort	0.09867 s	792.78 s	235.46 min
Insertion Sort	4.03476 s	314.31 s	17.67 min
Selection Sort	8.15739 s	565.02 s	25.33 min
Odd-Even Sort	15.79098 s	781.93 s	216.02 min

Table 1: The time take for algorithms to sort integers (0th order), sorting algorithms (1st order), and sorting sorting algorithms (2nd order)

it is only 1.13x slower, and for sorting sorting sorting, it is only 1.06x slower. It is conceivable that since insertion sort spends most of its time iterating over the beginning of the sorted list that when comparisons get more expensive, it gets more efficient.

Second, comb sort is much much slower for higher-order sorting than for integer sorting. For integer sorting, comb sort is only about 1.5x slower than merge sort, but for sorting sorting, it is about 3x slower, and for sorting sorting sorting, it is 14x slower. It is the slowest sort for higher-order sorting and took almost 4 hours compared to merge sort's 17 minutes. It is conceivable that comb sort, with its decreasing window, spends more time doing redundant tests as it reduces to bubble sort.

We ran tests counting comparisons to test both of these hypotheses and the evidence is clear: there is no evidence for these hypotheses whatsoever. Thus we present no explanation and offer a 50¢ prize for the first plausible explanation submitted to the author.

5 Conclusions

Use quicksort, unless you need to sort algorithms, in which case use merge sort. Most sorts behave the same across orders, but some do not. Thinking about higher-order sorting algorithms may cause temporary[7] insanity.

[7]At least, I hope it is temporary...

6 Future Work

There are numerous ways this novel direction of research can be expanded:

- Higher-order functions including, but not limited to, 3rd, 4th, and ωth order sorting algorithms
- Something involving quantum computers working on big data in the cloud
- Determining if there exist sorting algorithms that excel at non-constant comparison algorithms[8]
- Other meta-analyses such as:
  - A search algorithm for searching for search algorithms
  - A database for managing databases
  - A container for containing other containers[9]
  - A cache invalidator for managing cache invalidation algorithms

7 Acknowledgements

The author would like to thank a handful of people, including Arley Schenker and Ben Lichtman for proofreading this and sharing in the madness, Thomas Bayes, for obvious reasons, the Academy, for making this all possible, and readers like you, without whom this whole ordeal would have been meaningless.

[8]This is actually an interesting problem that I could not find much research on.

[9]This may be made redundant by Docker.