Project Sorting Algorithms Report

Aaron Subichev

Kamran Khan

September 29, 2022

The following sorts performed best by order of original list based on comparisons:

* In Order - Insertion/Selection
* Reverse Order - Selection
* Almost Order - Selection
* Random Order - Selection

The following sorts performed best by order of original list based on movements:

* In Order - Insertion/Selection/Quick
* Reverse Order - Quick Sort
* Almost Order - Quick Sort
* Random Order - Quick Sort

The following sorts performed best by order of original list based on time:

* In Order - Insertion Sort
* Reverse Order - Quick Sort
* Almost Order - Quick Sort
* Random Order - Radix Sort

| Across ALL sorts | Best Time (ms) | Worst Time (ms) | Avg. Time (ms) |
| --- | --- | --- | --- |
| Insertion Sort | 2.17 | 1959.78 | 653.57 |
| Selection Sort | 2.04 | 1183.53 | 289.34 |
| Quick Sort | 2.48 | 7.94 | 4.60 |
| Merge Sort | 3.68 | 11.28 | 6.99 |
| Heap Sort | 10.05 | 73.13 | 31.9 |
| Radix Sort | 2.65 | 10.44 | 6.00 |

Interestingly enough, Selection Sort always contained N comparisons. This is likely because I counted the comparison of an element to the already sorted list as one comparison when in reality, it would probably be N comparisons again, if not log(N) + 2, only if you can traverse backwards. If my counts were more accurate, I’m sure radix sort would beat Selection in terms of comparisons because I believe Radix is, in fact, N comparisons.

Across the board, it is clear that Quick Sort wins. The only situation in which quick sort is genuinely, significantly worse, is a situation in which the given list is already sorted. While this is a possibility, performing 10x the amount of needed comparisons sounds terrible, but with a list of size 50,000, Quick Sort only loses to Insertion Sort by a millisecond. In fact, it actually performs twice as fast as Selection Sort despite the 10x comparisons.

This is clear evidence that there is more to a sorting algorithm besides comparisons and movements. Just to name a few factors that come to mind, a sorting algorithm also might have to manage memory allocation and increase code complexity to deal with special cases. The thing about optimization is that written code becomes less readable, but for that sacrifice, you gain increased performance.

Something else worth pointing out is Radix Sort’s performance across the board. In every situation, Radix Sort, in theory, should have performed the same amount of operations. After all, the idea behind Radix Sort is that it compares all digits of all 50,000 numbers. No matter if they’re in order, out of order, or whatever, it should always average around the same time. However, as per my extensive testing, we can tell that it performs significantly worse for a list in order than for any other. It’s interesting to think about since I didn’t even touch Radix Sort. To be clear, I didn’t change implementation in any of the sorts except for quick sort since it was giving me trouble, but besides that, I will trust that all implementations handed to me were correct and worked well. At the end of the day, I was able to collect results just fine and analyze them here.

One factor that is completely unpredictable and makes this project somewhat moot in my opinion is that the machines students perform the tests on will vary greatly. Some have gaming rigs while others light notebooks. Of course, the simple fact that no two students will have the same results is alright by itself, but when you consider the possibility of background processes running on certain computers, it makes it difficult to effectively predict the performance of some of these sorts. I’m not just referring to a constant load on the CPU in the background too, since certain processes run when they see fit and some are more taxing than others... there’s really a lot to consider. During my tests, I had a couple of weird results. Namely, Radix Sort performing poorly on In Order lists, and Heap Sort getting slower and clocking some abnormally high times on Almost Order. It’s just not feasible for me to shut everything down on my computer and run the tests in a sterile environment. Even at that point, there’s still room for improvement. Ideally, I shouldn’t go for a supercomputer since that would be *too* fast, but I need to get a new computer built to ensure it’s not full of malware or background processes, and that way I would be able to get the cleanest results possible.

Analysis of all 24 experimental results:

50k elements

In Order

* Insertion Sort - nothing special here, the algorithm simply iterates and finds nothing to swap. A little wasteful since it needs to compare every single pair of adjacent elements, but nonetheless fast since it doesn’t take any serious time cost to compare two elements.
* Selection Sort - works exactly like insertion sort in this scenario, and compares each and every element into the sorted sublist it’s “created.” A little more wasteful since rather than comparing each pair of contiguous elements, it compares each element to the subarray. Nonetheless, it works quickly.
* Quick Sort - the absolute best sorting algorithm still gets bested by one of the simplest (usable) sorting algorithms out there, but that’s simply because quick sort wasn’t made to win the best case scenario. Despite not being the fastest, it only really wastes 1 millisecond of time despite doing 10 times as many comparisons. Later on, we’ll see this guy really dominate.
* Merge Sort - here is where we start to perform very poorly on an In Order list. the reason Merge, Heap, and Radix sort perform so poorly is because they’re designed to where they will move around elements regardless of whether or not they are in order. Merge Sort takes this fully sorted list, breaks it down recursively into 50,000 separate lists, and then merges all of those lists into the correct order.
* Heap Sort - Heap Sort is even more wasteful than merge sort since it has to add every single element into a heap and then from that heap remove every single element back into the list. In terms of readability and ease of explaining the algorithm to someone without a technical background, this algorithm works well. However, it is most wasteful when it comes to a list already sorted.
* Radix Sort - finally, Radix Sort. Surprisingly, this algorithm performs worst of all in an In Order list, although I have no idea how to explain it. At the end of the day, it compares the digits of all 50,000 numbers, despite whether or not the list is in order at the beginning, so I have no explanation as to why it performs worse on In Order.

Reverse Order

* Insertion Sort - Unsurprisingly, this is where insertion sort performs worst of all. After all, it now has to go through N items in the list, but then N items again in the sorted subarray in order to find where that first element must lie. This is terribly slow on a large list as is, but it performs significantly worse on a list sorted backward.
* Selection Sort - Again, a similar situation to insertion sort, but having to iterate through each element to find the maximum N times, which eventually is a constant operation at a list size of 1, in the worst case, it would take N times to find the maximum value and thus is incredibly slow, again, especially so on a list sorted backward.
* Quick Sort - this is where I experienced some bumps. The pivoting system forced my counts to overflow for some reason, even though I didn’t change the implementation at all. Furthermore, I tested the original implementation, without any of my code, against a list of size 50,000 sorted in reverse, and I still ran into exceptions being thrown. Thus, I changed the pivot to length/2 as suggested in the project instructions pdf, and so Quick Sort worked just fine. Furthermore, it performed the absolute quickest specifically on a reverse ordered list.
* Merge Sort - this sort performs incredibly consistently since its algorithm is so simple yet so well optimized. It’s a method that never fails to work. I want to point out that throughout all 4 orders of lists, Merge sort had a range of 4 milliseconds, where it performed fastest on a Reverse Order list. I also want to make it clear that the list being sorted backward is likely a non-factor in why Merge Sort performs fastest on the list, but rather just a side-effect. As I mentioned before, a lot of external factors affect the results I gathered and thus medians are the true values that need to be looked at.
* Heap Sort - slowest “good” operation by far, and this again has to do with the fact that we’re adding every single element, that’s N elements, to a Heap and then removing all N elements just to sort the list. Much more effective than Insertion or Selection sort, but definitely lacking compared to Quick, Merge, or Radix.
* Radix Sort - Radix Sort did well overall but I, again, don’t fully understand why it didn’t do better. At the end of the day, this is one of two medians of the total time taken to sort the Reverse Order list between the six sorts, so its performance is still commendable.

Almost Order

* Insertion Sort - the performance here is as expected since it’s almost twice as fast as Reverse Order, despite being more than 50% sorted. Of course, the way I implemented Almost Order was random, so in theory, the percentage could be skewed and it’s even possible that the list wasn’t sorted at all.
* Selection Sort - once again similar to Insertion Sort, no surprises. It’s easier in a list that is close to being in order since there’s a high chance you won’t have to iterate an exact average of N/2 times.
* Quick Sort - taking the cake yet again is Quick Sort. It’s easy to see why this algorithm performs so much better than either of the first two, but sorting a list of 50,000 elements in a mere 5 milliseconds is very impressive indeed, and still happens to have fewer comparisons than Merge or Heap sort.
* Merge Sort - a consistent algorithm, it’s no surprise that Merge Sort finished the task in just 7 milliseconds.
* Heap Sort - wasteful again, but still performed significantly quicker than Insertion or Selection sort
* Radix Sort - consistent as ever; 8.3 milliseconds is impressive

Random Order

* Insertion Sort - being in a Random Order, it’s no surprise that this algorithm performed worse than it did on Almost Order, but it’s still very slow.
* Selection Sort - I suppose selection sort just got incredibly lucky since it’s almost 2.5 times faster than insertion sort in this instance.
* Quick Sort - as fast as ever; a time of just 7.8 seconds is very good.
* Merge Sort- pretty good. If Radix sort didn’t go absolutely crazy then this sort would’ve placed 2nd in Random Order, but at the very least, Merge Sort is consistently quicker and therefore better.
* Heap Sort - wasteful but effective. performed just slightly worse than it did on an In Order list, and this just goes to show that no matter the order of the list, Heap Sort, for the most part, does the same thing and therefore shines in its consistency.
* Radix Sort - this was a triumph. I’m making a note here - “huge success.” It’s hard to overstate my satisfaction. Radix sort won for once, even beating out Quick Sort. To be honest, I expected these kinds of results from Radix Sort all along, but to my dismay, it only performed this well once. Why? I have no clue.