Caspar Lant

One `sort()` To Rule Them All?

Sorting algorithms tend to each perform differently under varying conditions. It is for this reason that Java's built-in `Arrays.sort()` method implements a quadratic sort for the purposes of ordering arrays of relatively short length, and more complex, logarithmic sorts for arrays which contain more elements. It is also for this reason that it is necessary for a programmer to be well-versed in several different sorts of sorts, and not just one with infinite applicability.

In our convention of "big-O" notation, we tend to ignore any coefficients involved in the time-function if a search algorithm. We data-nerds are interested in how these sorts *scale* with time, rather than the durations for which they run on an absolute timescale. Any sorting-program will run for varying lengths based on the machine that it running on, the language in which it is written, and the ordering of the dataset that it was given. The runtime of a sort algorithm will even vary based on the physical conditions under which it is housed. The presence of these intense and seemingly-stochastic fluctuations in runtime render any "more precise" measure–one which included coefficients and constants, for instance–of a program's runtime meaningless. I will attempt to illustrate this variability in sort performance under various computational conditions.

In this project I will explore the merits of three well-known sorting algorithms: selection sort, merge sort, and quicksort, and explore the possibilities of optimizing them.

The `SelectionSort` class implements a, you guessed it, non-recursive selection sort algorithm. Due of the algorithm's quadratic time-scaling (order $O(n^2)$), unsorted arrays of length > 40000 produced a `java.lang.StackOverflowError`. For completeness, the recursive solution has been kept in the `SelectionSort.java` file and commented out.

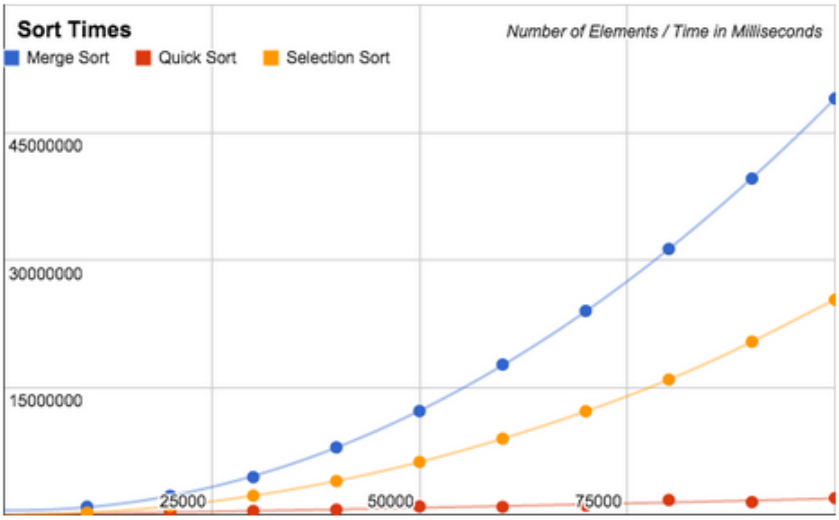
Following my own tautological convention, the SelectionSort.java file contains a selection sort algorithm. Despite the fact that selection sort is of category $O(n\log_2 n)$, my program took far longer to

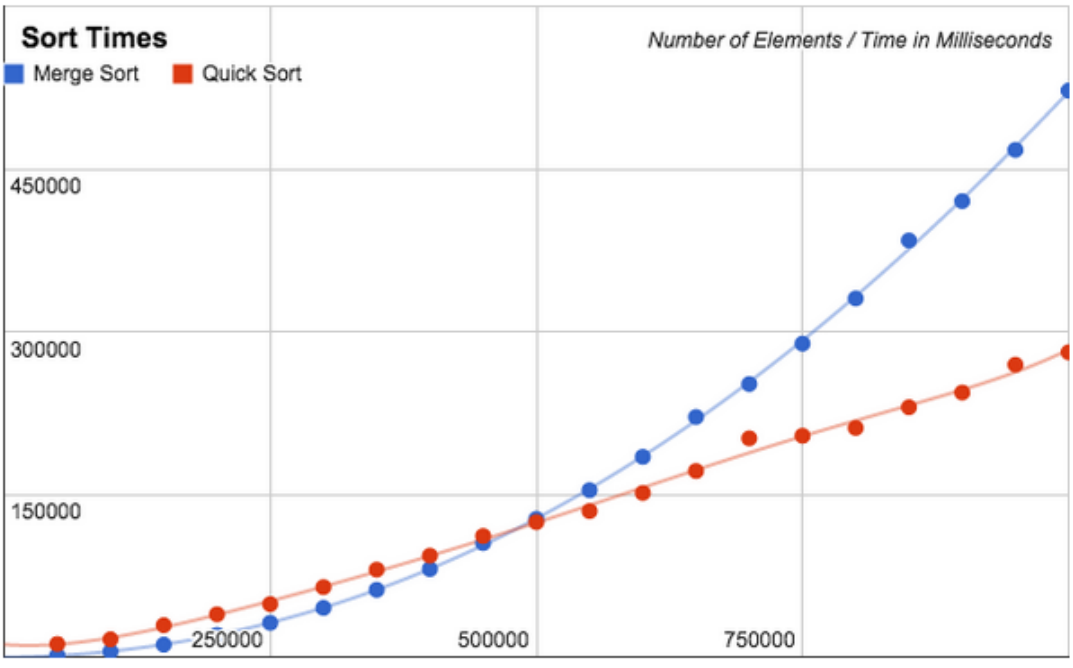
run than my implementation of quicksort, and appeared to scale with a function somewhere in between it's alleged $n\log_2 n$ proportionality and merge sort's n^2 time-scaling. This dubious result is likely due to the nested for-loops apparent when one juxtaposes the `minIndex(int, int)` and `selectionSort(int)` methods.

Lastly comes `QuickSort.java`'s quicksort: the holy grail of comparison sorts. In this implementation, the initial unsorted array is stored as a class variable of type `E[]`, along with the int `SIZE`–the array's length. I suspect this speeds things up a bit. Quicksort adheres to it's famed $O(n\log_2 n)$ time-scaling with the resoluteness of a Spartan warrior; a beauty to behold.

DATA

Elements	Merge Sort	Quick Sort	Selection Sort
10000	949551	143610	252177
20000	2221792	281720	990839
30000	4428073	445030	2246586
40000	7930074	600340	3974401
50000	12204675	975950	6215915
60000	17676066	924760	8956631
70000	23989667	1099170	12177534
80000	31275398	1740180	15906864
90000	39560419	1493090	20362759
100000	48978231	1962010	25302255





Elements	Merge Sort	Quick Sort
50000	1491	11923
100000	5028	16245
150000	11327	29251
200000	20235	39315
250000	31464	48711
300000	45251	64406
350000	61885	80543
400000	80992	93355
450000	104994	111485
500000	127321	124298
550000	153732	134474
600000	184520	151238
650000	221243	171467
700000	251628	201573
750000	288875	203932
800000	330519	211136
850000	384044	230216
900000	420229	243790
950000	467518	269490
1000000	522146	280913

[Upward divergence from the trendline most likely signifies an instance of `GarbageCollector`]

As previously stated, it is difficult to make the claim that `MergeSort` runs in $n\log_2 n$ time. This becomes blindingly apparent when one compares it to the time-scaling of `QuickSort` in the second graph. The fact that my merge sort program falls somewhere between `QuickSort and SelectionSort` is shown by the first graph. Interestingly enough, my implementation of merge sort shows better performance than my quicksort until halfway through the arrays. I suspect that this has something to do with my choice of pivot in quicksort[1]. In the hopes of optimizing the quicksort program, I will change my choice of pivot point to the median of the first three values present in each unsorted array. The algorithm (made simpler by assuming an array of `int` primitives) is as follows:

```
int median(int first){
	a = values[first  ];
	b = values[first+1];
	c = values[first+2];
	if ((a - b) * (c - a) >= 0)
		return a;
	if ((c - b) * (b - a) >= 0 && (=b < c || (b >= c && b <= a)))
		return b;
	return c;
}
```

This algorithm[2] produces perhaps a better estimate of the array's median (the ideal pivot-value), whilst keeping the number of comparisons to a minimum. This algorithm assumes an array of length greater-than or equal-to three, which will be dealt with in the form of another conditional statement directly before the function is called in the `split(int, int)` method. The `median(int)` method

[1] As we know, the variability in computation time for quicksort is quite large. Depending largely on the means of determining the pivot-point, as well as the specified dataset, the time range for quicksort can anywhere between $n\log_2 n$ and n^2.

[2] Coincidentally, this median algorithm forms part of the solution to determining the Euler line of a triangle.

stores the three values as local variables to bypass duplicate evaluations. The function call, wrapped by the aforementioned conditional statement, is as follows:

```
int split(int left, int right){
	E pivot;
	if (right - left >= 3)
		pivot = median(left);
	else
		pivot = values[left]; //leftmost object in subarray
	...
}
```

This change has a minor impact on the runtime of my `QuickSort` implementation: it marginally decreased the rate of change of the time function. This means that the more subarrays my program had to evaluate through, the greater of an impact my optimization had. It seems logical that the median of the first three values of a large array would be a better representation the array's median than in the case of a smaller array. This method is not included in the current implementation of `QuickSort.java` because it does nto support generic types. Instead, one could use this mess of conditionals:

```
E median(int first){
	E a = values[first  ];
	E b = values[first+1];
	E c = values[first+2];
	if (a.compareTo(b) > 0) {
		if (b.compareTo(c) > 0) {
			return b;
		}
		if (a.compareTo(c) > 0) {
			return c;
		}
		return a;
	}
	if (a.compareTo(c) > 0) {
		return a;
	if (b.compareTo(c) > 0) {
```

```
                return c;
            }
            return b;
        }
```

This implementation, however, does nothing to speed up the runtime of my quicksort algorithm, no double due to the overwhelming frequency of `if` statements and `compareTo(E)` method calls.

If one assumes that the unsorted array represents experimental results bound by some distribution, it would be computationally-efficient to choose the middle value of the array as a pivot. When dealing with large arrays filled with random integers, however, this change has no measurable or consistent impact.

When working with very large arrays that are random by design, the quicksort algorithm certainly shines brightest. With smaller, similarly randomized arrays, mergesort comes out on top. It is only in instances of very small unsorted arrays that selection sort becomes practical.