Selection Sort

Algorithm Benchmark Program Analysis

Stephanie Anderson

CMSC 451 Project 2

10/14/18

The selection sort algorithm sorts an array by selecting the minimum element from the unsorted portion and swapping it with the leftmost element. That element then becomes part of the sorted array, and the process is repeated on a smaller and smaller unsorted portion until the array is fully sorted.

A high-level pseudocode for the recursive version of the selection sort is as follows:

```
recursiveSort(list)
      recursiveSortInner(list, 0)


recursiveSortInner(list, j)
      if (j >= n - 1) return;
      min = j;
      for (i = j + 1; i < n; i++)
            if (list[i] < list[min])
                  min = i;
      swap list[j] and list[min];
      recursiveSortInner(list, j + 1);
```

The high-level pseudocode for the iterative version of the selection sort is as follows:

```
iterativeSort(list)
      for (j = 0; j < n - 1; j++)
            min = j;
            for (i = j + 1, i < n; i++)
                  if (list[i] < list[min])
                        min = i;
            swap list[j] and list[min];
```

Since the selection sort searches through the unsorted portion of the array of size *n* an *n* number of times, the selection sort in both versions of the algorithm have a quadratic time complexity, which makes it inefficient on large lists.

$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1)-(i+1)+1 = \sum_{i=0}^{n-1} n-i-1 = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1$

$= n^2 - \sum_{i=1}^{n-1} i - n = n^2 - \frac{n^2-n}{2} - n = \frac{n(n-1)}{2} \approx n^2$ for large *n*, so the Big-O is $O(n^2)$.

My approach to avoid the problems associated with the JVM warm-up was to load a large number of dummy classes in the main before running the sorting program. I tested this method myself by running the warm-up once in the static block and once in main. The one in main was faster when the one in the static block was included, versus when only the one in main was included.

The critical operations I chose to count in the selection sorts were both comparisons and assignments. My initial Big-O analysis only counted comparisons (`if (list[i] < list[min]))`, but with true Java implementation, the three assignments that execute the swap are also critical operations.

Figure 1 shows one example of the output with the array sizes {2500,5000,7500,10000,12500,15000,17500,20000,22500,25000}.

	Recursive				Iterative			
Size	Average Count	Coefficient of Count	Average Time	Coefficient of Time	Average Count	Coefficient of Count	Average Time	Coefficient of Time
2500	3136225.24	2.57e-06	5070840.60	0.15	3133725.24	2.57e-06	5178233.18	0.03
5000	12522470.50	7.10e-07	20623556.46	0.11	12517470.50	7.10e-07	20946872.66	0.03
7500	28158718.52	3.59e-07	46390831.72	0.08	28151218.52	3.59e-07	46874420.60	0.03
10000	50044967.44	1.52e-07	83460138.58	0.07	50034967.44	1.52e-07	84225903.30	0.03
12500	78181217.38	1.12e-07	133779849.70	0.06	78168717.38	1.12e-07	134731643.50	0.02
15000	112567466.84	7.33e-08	191034607.58	0.05	112552466.84	7.33e-08	192293116.88	0.02
17500	153203714.62	6.91e-08	258706723.20	0.04	153186214.62	6.91e-08	259996572.64	0.01
20000	200089964.38	4.18e-08	338131932.86	0.04	200069964.38	4.18e-08	339650192.58	0.01
22500	253226214.44	4.22e-08	428481610.76	0.04	253203714.44	4.22e-08	430314458.66	0.02
25000	312612464.02	2.91e-08	526592642.24	0.03	312587464.02	2.91e-08	528654996.68	0.01

Saving...

Figure 1

Figure 2 plots the average counts for both the recursive and iterative sorts. One can see the parabolic curve as the array size increases. The lines appear to overlap, but the average count for each array size in the recursive sort is exactly *n* higher than the average count for the same arrays in the iterative sort.

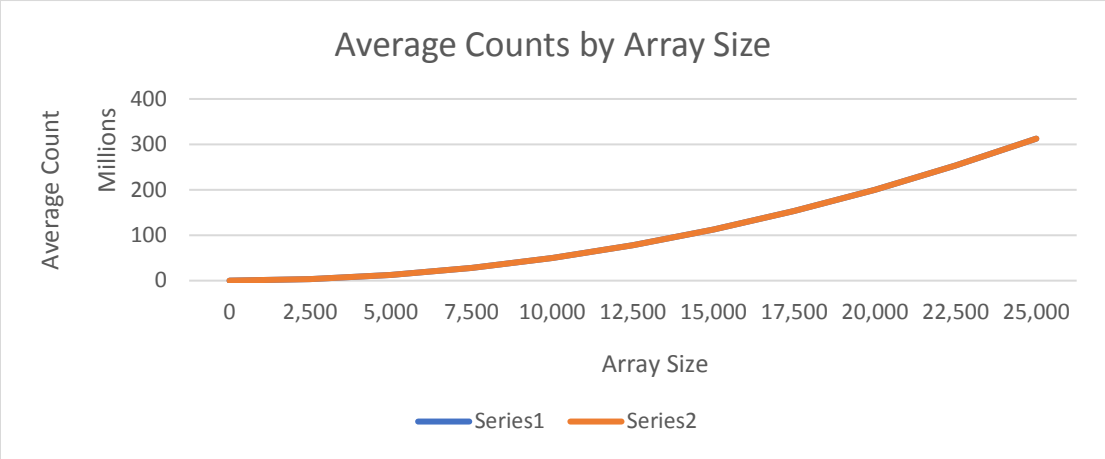

Figure 2

An equation to get approximately the same output would be $\frac{n(n-1)}{2} + 4n$ for iterative and $\frac{n(n-1)}{2} + 5n$ for recursive. Both theoretical equations show that the average counts for both algorithms is indeed quadratic. The difference between the equation and actual output is due to the algorithm not performing the swap when the leftmost element in the unsorted portion is already the minimum, which

makes it a tad more efficient than the equations for lower data sets. The standard deviation is due to this; during the few times that the element is already in the correct place, the program saves on a couple superfluous instructions, and fluctuates the count. The best-case scenario with this implementation would be zero swaps on an already sorted array. Figure 3 shows that in big array sizes, the coefficient of variance asymptotes to zero, which correlates to how large data sets will follow the average-case scenario.

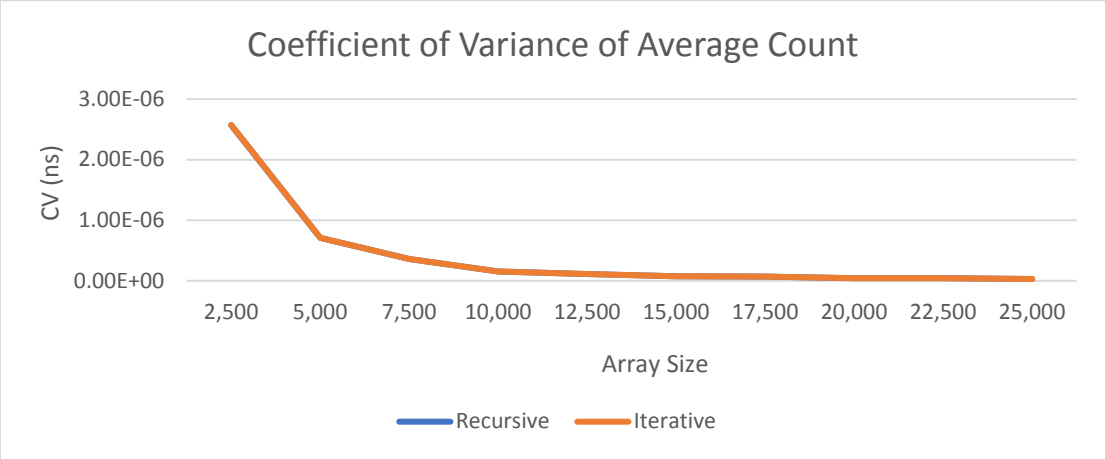

Figure 3

Figure 4 plots the average times for both the recursive and iterative sort. The parabolic curve is similar to that of the average counts which supports the quadratic time complexity. Also, the lines once again nearly overlap, with the iterative taking slightly more time than recursive for each array size.

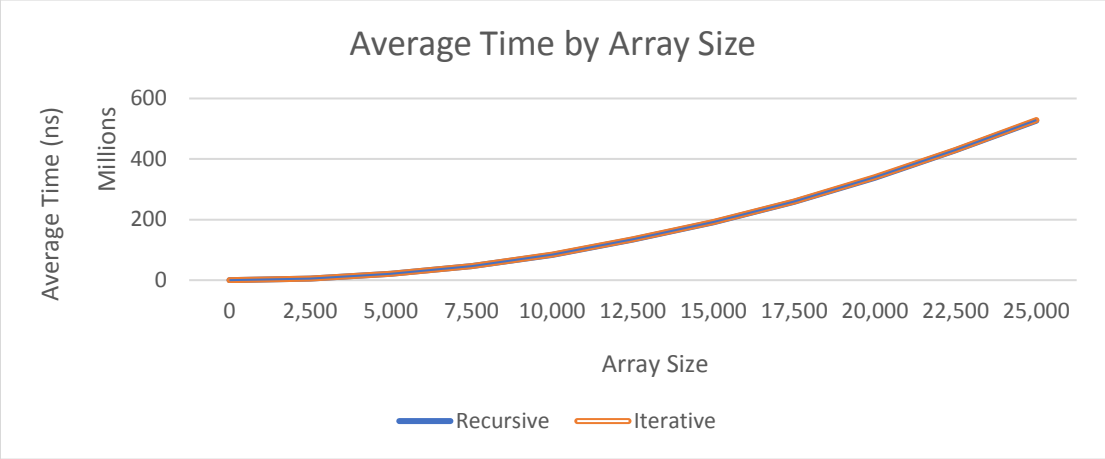

Figure 4

Figure 5 plots the coefficient of variance for the average times for both the recursive and iterative sort. The recursive has a similar curve to that of the count's coefficient of variance, but asymptotes a little above zero. Even in high data sets, there will be a fluctuation of time because of the computer's own processing. Unlike the recursive sort, the iterative sort's coefficient of variance sits linearly just above zero for all array sizes, showing a consistent average-case time complexity across all data sets. The

higher coefficient in the smaller arrays' iterative sort average counts does not reflect on the real time data.

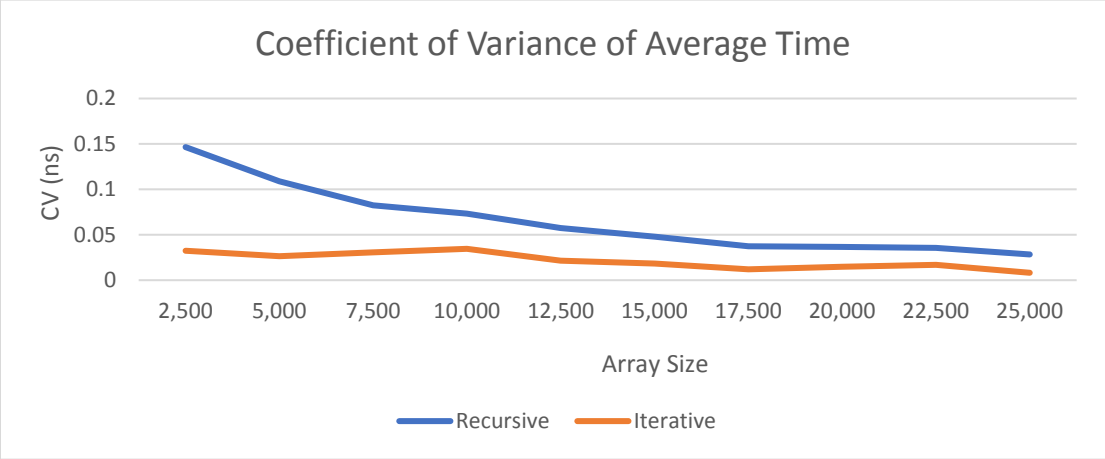

Figure 5

The recursive sort had a higher average count, but the iterative sort took longer on the same sets of data, implying the recursive sort is actually more time-efficient in a Java environment. However, the iterative sort takes much less memory, which makes it more practical, especially for larger data sets.

The average counts and times for both versions of the sort all show a quadratic time complexity, which correlate to the previous theoretical analysis of the selection sort. With support from these benchmarks, the Big-O of a selection sort is $O(n^2)$, which is inefficient for large data sets.