Stephen Jones
CMSC 451
Project 2 (Final)
Due: 14OCT2019

Introduction

The sorting algorithm I chose is the bubble sort algorithm. This algorithm is as asymptotically inefficient as it is simple, which will be shown in this project. Bubble sort is similar to selection and insertion sort, in that they are all relatively slow, but also that they all involve scanning through a sequence of values (such as an array) and sorting them. Bubble sort does this by comparing every index pair and swaps the greater value toward the end of the array (for an increasing order sort). Once the last index is the greatest value, the bubble sort repeats the process at the first index again and moves the next greatest value to the index just before the last index. This continues until the array is sorted.

The two versions of the bubble sort presented in this project are the recursive and iterative versions:

The recursive version contains the method `bubbleSortRec()`. This method continually calls itself with a decrementing value that reflects the last index of the array to be compared with the conditions that there are enough values left to compare and when a swap occurred in the last scan. As long as the two conditions are met, this method also calls the `bubbleLargestRecursive()`. This method recursively scans through all the values of the array, up until the last index to be compared, and calls a utility function (`swapElements()`) that swaps any two values that are out of order. This version also includes an initializing method, `recursiveSort()`, that is called once.

The iterative version, `iterativeSort()`, contains two nested loops. The inner for loop compares a pair of indices, swaps them if they are out of order, then incrementally moves up the array. The outer do/while loop decreases the portion of the array that the for loop scans accounting for the last index that was fully sorted. This version also uses the same `swapElements()` method to swap the unordered values.

Pseudocode

Here is the pseudocode for the recursive sorting algorithm where n is the last index in the array to be compared.

```
recursiveSort(array)
   Assign array to global variable
   bubbleSortRec(array.length - 1)

bubbleSortRec(n)
   If there is less than two values in array to be compared
      Return;
   Assign swap variable to false;
   bubbleLargestRecursive(0, n);
   If two values were swapped
      bubbleSortRec(n-1);
```

```
bubbleLargestRecursive(i, n){
   If the current index (i) is equal to the last index to b
   compared (n)
      return;
   If the current index value (i) is greater than the next index
   value (i+1)
      swapElements(i,i+1);
      Assign swap variable to true;
   bubbleLargestRecursive(i+1 , n);
```

Here is the pseudocode for the iterative sorting algorithm where, again, n is the last index in the array to be compared.

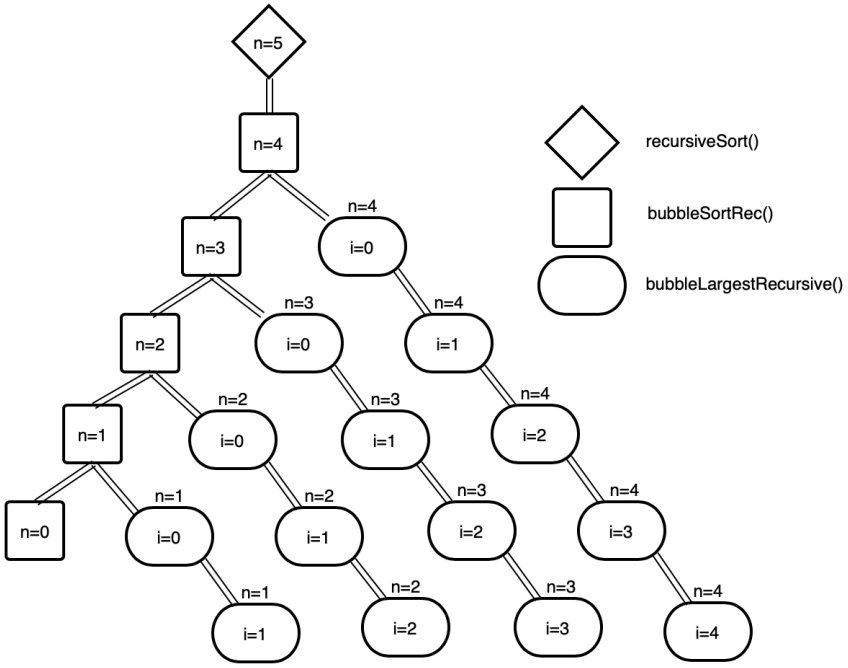
```
iterativeSort(array)
   Assign array to global variable
   If there are less than two values in array to be compared
      Return;
   Do once and then while the swap variable is true
      Assign swap variable to false;
      For each index (i) in the array up until the last index to
      be compared (n)
         If the current index value (i) is greater than the next
         index value (i+1)
            swapElements(i,i+1);
            Assign swap variable to true;
      n - 1

swapElements(x, y)
   Increment critical count variable
   Swap values at index x and y using an intermediate variable
```

Big-Θ Analysis

Here is the Big-Θ analysis for the **recursive algorithm**. It includes an example activation tree, a summary of the time values from that example, the summation produced from that summary, the simplification of the summation, solving the summation, a proof of that solution, and finally the Big-Θ growth rate that corresponds to the recursion.

Activation Tree Where input (n) = 5



Assuming that each call to a method above is some constant, we can use the tree to determine a summation pattern:

T(5) = 6 + 5 + 4 + 3 + 2
T(4) = 5 + 4 + 3 + 2
T(3) = 4 + 3 + 2
T(2) = 3 + 2
T(1) = 2

Note: The 2 above is from the initial method `recursiveSort()` which is always called and the last call of `bubbleSortRec()` which results in a return from the method.

Here is the summation from the above pattern with some manipulation to turn it into the arithmetic series form and solving that summation:

$$T(n) = \sum_{i=3}^{n+1} i + 2 = \sum_{i=3}^{n+1} i + 2 + 1 - 1 = \sum_{i=1}^{n+1} i - 1 = \frac{(n+1)((n+1)+1)}{2} - 1$$

Here is the simplification of that solution:

$$= \frac{(n+1)((n+1)+1)}{2} - 1$$

$$= \frac{(n+1)(n+2)}{2} - \frac{2}{2}$$

$$= \frac{n^2+3n+2}{2} - \frac{2}{2}$$

$$= \frac{n^2+3n+2-2}{2}$$

$$= \frac{n^2+3n}{2}$$

Here is the proof by induction:

Basis Case:

$$T(1) = \frac{(1)^2 + 3(1)}{2} = \frac{4}{2} = 2$$

Induction Step:

$$T(k) = \frac{(k-1)^2 + 3(k-1)}{2} + k + 1$$

$$T(k) = \frac{(k-1)^2 + 3(k-1)}{2} + \frac{2k + 2}{2}$$

$$T(k) = \frac{k^2-2k+1+3k-3}{2} + \frac{2k + 2}{2}$$

$$T(k) = \frac{k^2+3k-2k+2k-3+1+2}{2}$$

$$T(k) = \frac{k^2+3k}{2}$$

Finally we can keep the largest term and remove any coefficients and get:

$$T(n) = \Theta(n^2)$$

Here is the Big-Θ analysis for the **iterative algorithm**. It includes a summation, solving the summation, a proof of that solution, and finally the Big-Θ growth rate that corresponds to the iteration.

There are two loops in the iterative algorithm, a for loop inside a do/while loop. The do/while loop will run n times, with the n variable decrementing by one each pass. The for loop will run n-1 times for each pass of the do/while loop. This means that the for loop will be n-1 for the first pass of the do/while loop, while the second pass of the do/while loop will result in n-2 passes in the for loop to account for the decrement of the n variable. A truncated summation would look like this:

$$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

Writing this summation in reverse reveals the same form as an arithmetic series:

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{(n-1)(n)}{2}$$

Here is the simplification of that solution:

$$= \frac{(n-1)(n)}{2}$$

$$= \frac{n^2-n}{2}$$

Here is the proof by induction:

Basis Case:

$$T(2) = \frac{(2)^2 - 2}{2} = \frac{4 - 2}{2} = 1$$

Induction Step:

$$T(k) = \frac{(k-1)^2 - (k-1)}{2} + k-1$$

$$T(k) = \frac{(k-1)^2 - (k-1)}{2} + \frac{2k-2}{2}$$

$$T(k) = \frac{k^2-2k+1-k+1}{2} + \frac{2k-2}{2}$$

$$T(k) = \frac{k^2-k-2k+2k+1+1-2}{2}$$

$$T(k) = \frac{k^2-k}{2}$$

Keeping only the larges term and removing any coefficients we are left with:

$$T(n) = \Theta(n^2)$$

JVM Warm-up

At the beginning of a program, the JVM loads certain classes into the cache to make them more quickly accessible during the program runtime. This caching process usually results in slower runtime speed at the beginning of a program. One method of avoiding this problem, especially when trying to obtain accurate benchmarking values, is to create a "dummy" class with enough processing work to initialize (or warm-up) the JVM to its optimum runtime environment (Baeldung, 2018).

The program in this project includes a "dummy" class that simply cycles through an empty for loop thousands of times in order to warm-up the JVM. There is a section of code before any of the benchmarking that calls the warm-up class in a for loop. This for loop was included to help find an adequate amount of activity to include before the benchmarking. So, when a single iteration of the warm-up class instantiation was not sufficient, another iteration of the warm-up could easily be called until the warm-up was sufficient.

The gauge for a sufficient warm-up was either that the durations of the warm-up iterations were consistent (not substantially decreasing from the last iteration) or the duration was so small it was inconsequential for the benchmarking results (hundreds of nanoseconds).

Critical Operation

Bubble sort, and other sorting algorithms, are used to manipulate a sequence of values such that the sequence becomes descending or ascending. The method by which this is accomplished comes in two main stages. First is to compare a pair of values and second is to swap those values in the sequence if they are not in the desired order. The critical operation in this project (and in general) for the bubble sort is when two values are swapped.

Using the first stage of comparison as the critical operation could be useful in reflecting how many iterations of the sort occurred. However, a sequence of thousands or millions of values would have the same critical operation count whether there were no swaps or a swap at every pair, even though these two situations would likely have drastically different time durations.

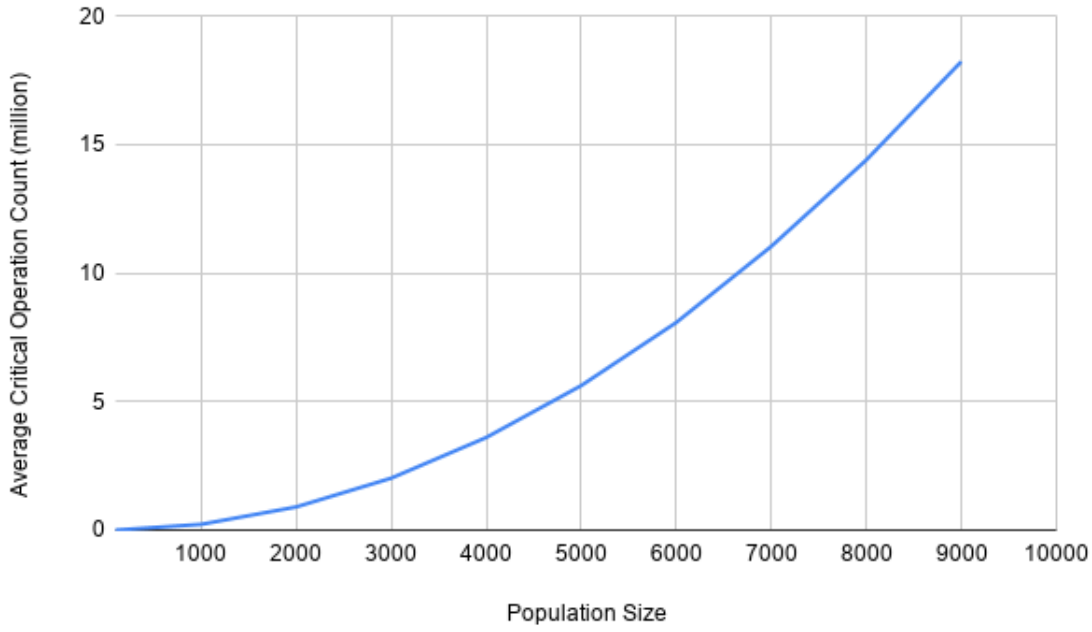
Using the swap as the critical operation, on the other hand, makes more sense to more accurately reflect the time duration used in the benchmarking. The number of swaps a single run of the program makes reflects the duration of the program runtime. If no swaps are made in a large population size, the exception time should be much less than if a swap is made at every pair.

Analysis

What follows are the results of the analysis and discussions on key components of that analysis. This section includes graphs of the population size's effect on the average critical operation count and average execution times for the recursive and iterative algorithms and discussions on the performance of the two algorithms, the critical operation counts compared to the execution times, the coefficient of variance calculation results, and finally the benchmark results compared to the Big-Θ analysis above.

Average Critical Operation Graph

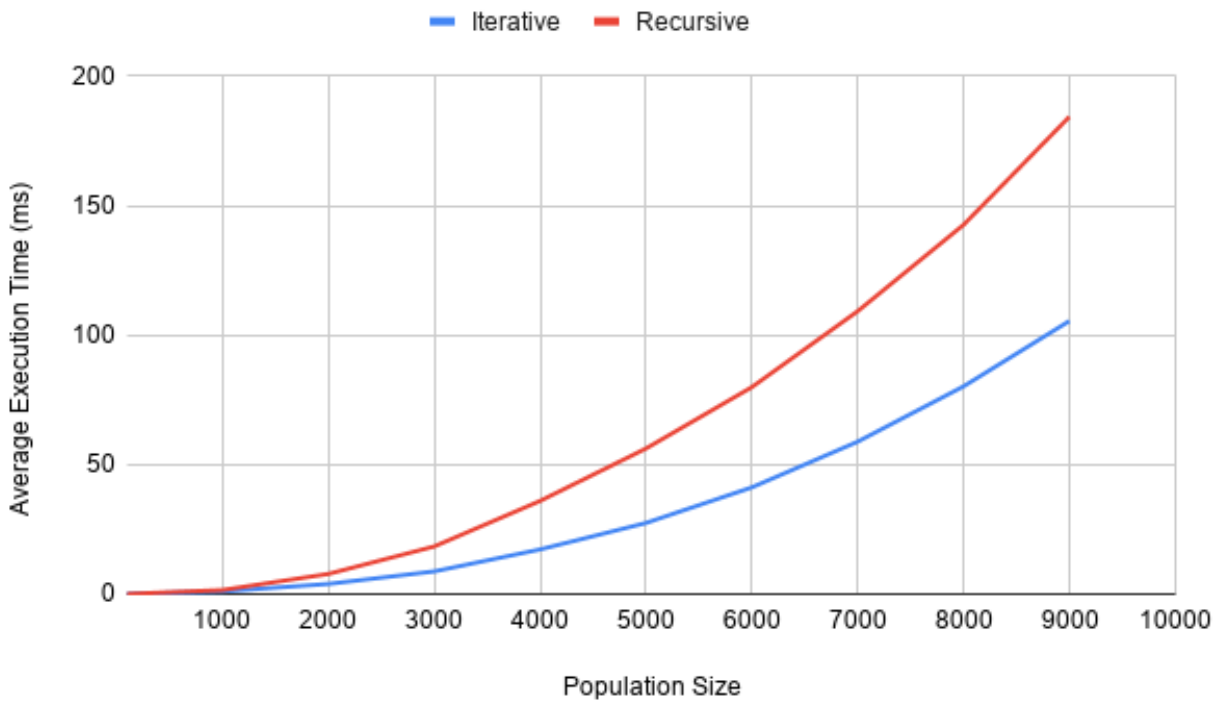
Fig. 1: Effects of population size on the average critical operation count (million) of both the recursive and iterative algorithms.



Note: Because both algorithms were using arrays with the same values, the critical operation count for both algorithms for every repetition of each population size is the same. This one graph accurately represents both algorithms.

Average Execution Time Graph

Fig. 2: Effects of population size on the average execution time (ms) of the recursive algorithm (red line) and the iterative algorithm (blue line).



Performance Comparison

Though the critical count is a reflection of the work performed by each algorithm, each repetition of the benchmarking used the same array values for both algorithm types. This makes the comparison of the average execution times to be more appropriate.

As can be seen in Fig 2 above, the average execution time for the recursive algorithm grows faster than the iterative algorithm by a constant factor. Initially the two algorithms have similar average execution times, but the recursive algorithm branches off to grow at a faster rate. This shows that the iterative algorithm is more time efficient at the values used in this project.

However, asymptotically speaking, the terms and coefficients causing this faster growth rate will become trivial. Both graphs show a polynomial trend with n^2 as the term that effects the growth rate the most, therefore they are both in the same order of growth rate.

Critical Operation vs. Execution Time

As can be seen in Fig. 1, the critical operation count results are the same for both algorithm types. This is because the same array values were used for each algorithm during each repetition of the benchmarking. Since each algorithm effectively accomplishes the same task (bubble sorting the array) for the same array values, they will have the same critical operation values.

What is worth noting is that the graph for the critical operation counts has a direct relation with the graph of the execution times, they are both growing at a polynomial rate (specifically $\Theta(n^2)$). This relationship makes sense because the critical operation, like the execution time, is a reflection of the work the algorithm is performing. The more critical operations each algorithm performs, the longer the algorithm will run.

Coefficient of Variance vs. Data Sensitivity

The coefficient of variance (CoVar) was used to measure the precision of the execution time and critical operation counts that were measured at the different population sizes. A higher CoVar means that there is a wide range of values in the data collected, while a lower CoVar means there is a narrow range of values collected.

As can be seen in the table 1 in the appendix, the CoVar (as represented by a percentage) for the average execution times for the iterative and recursive algorithms begin relatively high (88.12% and 200.98% respectively), but drastically tapers off as the population size grows. The final population size of 9000 tested results in a minuscule CoVar of 0.65% and 0.60% respectively, almost identical. The results for the average critical operation count are similar but with less drastic numbers (8.70% and 0.89% for the upper and lower bounds).

What this reveals is that the algorithms are much more sensitive to the data at lower population sizes. There is a much wider range of results, especially for average execution time, when the population size is lower. In stark contrast is when the population size is higher. In this case the algorithm is much less sensitive to the differences in the data and results in a precise and consistent average execution time and critical operation count.

Results vs. Big-Θ Analysis

The results match well with the Big-Θ analysis in the previous section. Even with relatively low array sizes, the graph representations of the data clearly show a polynomial growth rate (n^2) for both algorithm types which is expected for the bubble sort algorithm for the average (and worst) cases. The graphical comparison in fig 2 of the average execution times for each algorithm type reveals that the recursive algorithm is growing at a faster rate than the iterative algorithm, but when the term that effects the growth rate the most is isolated and any coefficients are removed, we are left with $\mathbf{n^2}$ for both algorithms which is the asymptotic growth rate we would expect, $\mathbf{\Theta(n^2)}$.

Conclusion

For this benchmarking of bubble sort algorithms, the results reflect what we would expect. Both graphs (fig. 1 and 2) show the measured growth rate for both algorithms to be consistent with the expected asymptotic growth rate of $\Theta(n^2)$.

$\Theta(n^2)$ is not a particularly efficient growth rate, so bubble sort seems best for relatively small sizes of data. Its simplicity allows it to be easily understood, programmed and manipulated depending on the data being sorted. However, 9,000 data points might seem like a large population size in certain situations. Even in the less efficient recursive algorithm, the average execution time was less than 0.2 seconds, as can be seen in the results table 1 in the appendix. Considering it's relative simplicity, small scale uses of this algorithm could be appropriate.

Another consideration not discussed yet is the space efficiency of the algorithm. The reason the population size cap was 9,000 was because anything above that resulted in a stack-overflow for the recursive algorithm. The iterative algorithm, the one that was slightly more time efficient, has a space complexity of $\Theta(1)$, so it would be able to process larger population sizes. Again, this might be an ideal algorithm to use if the population size was relatively small (but still greater than 10,000) since this will still result in a total runtime of less than half a second.

Asymptotically speaking, this algorithm has a relatively inefficient growth rate, but there could be very reasonable situations that the bubble sort, as well as the selection and insertion sort algorithms, could be very useful.

Appendix:

Table 1: Table showing the results of the data collected from project 1

	Iterative				Recursive			
Data Set Size	Average Critical Operation Count	Coefficient of Variance of Count (%)	Average Execution Time (ms)	Coefficient of Variance of Time (%)	Average Critical Operation Count	Coefficient of Variance of Count (%)	Average Execution Time (ms)	Coefficient of Variance of Time (%)
100	2185	8.70	0.09	88.12	2185	8.70	0.06	200.98
1000	224512	2.46	1.04	22.87	224512	2.46	1.57	17.68
2000	900786	1.68	3.93	7.09	900786	1.68	7.79	4.12
3000	2020113	1.51	8.77	0.82	2020113	1.51	18.40	0.50
4000	3601240	1.01	17.25	12.07	3601240	1.01	35.98	9.27
5000	5620611	1.08	27.48	4.72	5620611	1.08	56.22	7.41
6000	8073319	0.95	41.21	3.22	8073319	0.95	79.92	3.20
7000	11033424	0.85	58.87	2.35	11033424	0.85	109.26	3.13
8000	14392147	0.85	80.23	1.24	14392147	0.85	142.63	0.88
9000	18220812	0.89	105.49	0.65	18220812	0.89	184.37	0.60

References:

Baeldung. (2018, April 15). How to Warm Up the JVM. Retrieved October 9, 2019, from https://www.baeldung.com/java-jvm-warmup.