

CMSC 451

Project 2

Bubble Sort

1. Project Summary

1.1. Purpose

Using Java programming language, we benchmark the recursive and iteration function using the bubble sort algorithm. While identifying the number of critical operations and the time it took in nanoseconds. Then analyze the data our interface generates.

1.2 Operations

There are two programs to run. One is labeled 'CMSC451-Project1' and 'CMSC451-Project1_GUI'. 'CMSC451-Project1' performs the benchmarking and generates two data files labeled 'Iterative.txt' and 'Recursive.txt'. You can delete these files after each run, or the data will just keep adding to the text file. Either way, 'CMSC451-Project1_GUI' will produce the report. This program allows you to choose the text file you want to generate a report on. The report will calculate the average and coefficient of variance for count and time.

1.3 Display

The design is very straight forward for both programs. 'CMSC451-Project1' output text produces 10 lines that correspond to the 10 data set sizes. The first value of each line is the data set size, followed by element count, and time in nanoseconds; than 49 more pairs of count and time (Figure 1). The only thing separating each value is a space.

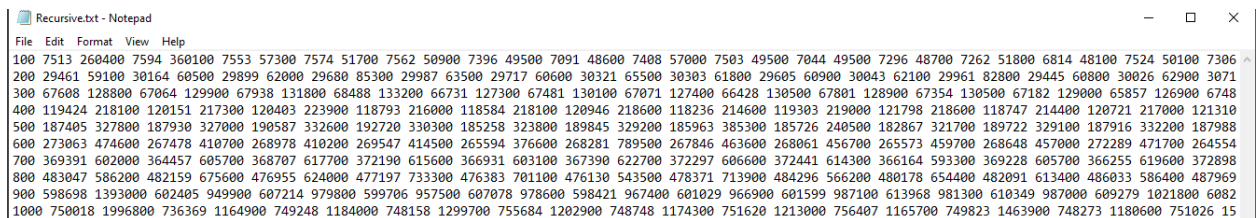


Figure 1: Generated Value

'CMSC451-Project1_GUI' allows you to select the text file you want to run the benchmark report on. The report is taking the information from figure 1 and puts it into a table (Figure 2).

Benchmark Report - Iterative.txt						Benchmark Report - Recursive.txt					
Size	Avg Count	Coef Count	Avg Time	Coef Time		Size	Avg Count	Coef Count	Avg Time	Coef Time	
100	7403.54	2.42%	117312.0	98.35%		100	7408.92	2.53%	59094.0	95.43%	
200	29846.9	1.35%	110792.0	74.15%		200	29842.56	1.56%	61188.0	11.72%	
300	67381.46	1.32%	142550.0	6.06%		300	67350.86	1.15%	125330.0	7.13%	
400	119471.86	1.07%	232688.0	9.59%		400	119864.02	1.07%	205760.0	12.23%	
500	187118.88	0.95%	358872.0	3.16%		500	187343.7	1.11%	326852.0	4.83%	
600	269677.6	0.85%	537108.0	23.54%		600	268951.16	0.88%	478630.0	23.28%	
700	367750.42	0.88%	627878.0	13.17%		700	367824.02	0.81%	563354.0	12.91%	
800	479909.36	0.6%	979414.0	35.22%		800	479793.76	0.88%	908970.0	38.53%	
900	605293.56	0.72%	1227718.0	28.24%		900	605528.96	0.79%	1124760.0	28.29%	
1000	747788.6	0.86%	1424772.0	12.12%		1000	749270.68	0.6%	1384772.0	23.99%	

Figure 2: Benchmark Report

2. Bubble Sort

2.1 Definition

“This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order” (Data Structure - Bubble Sort Algorithm).

2.2 Iterative and Recursive Solution

2.2.1 Iterative

The program will take an unsorted array and begin sorting it using the bubble sort method. It starts by comparing the first 2 elements of the array and swaps them if the first element is greater than the second. At the end of each pass the next largest element will “bubble up” to its correct position. This is 1 iteration and will keep iterating $n-1$ times. Since the array is 10 elements there will be 9 iterations. The auxiliary space used is $O(1)$ for the call stack. The pseudocode is as follows:

```
// Iterative version of Bubble Sort
public int[] iterativeSort(int[] a) throws UnsortedException {
    timeStart = System.nanoTime();

    int n = a.length;
    for (int i = 0; i < n - 1; i++) {
        // last i items are already sorted, so inner loop can avoid looking at the last
        // i items
        for (int j = 0; j < n - 1 - i; j++) {
            count++;
            if (a[j] > a[j + 1]) {
                swap(a, j, j + 1);
            }
        }
    }

    timeEnd = System.nanoTime();
    checkSortedArray(a);
    return a;
}
```

Figure 3: Iterative Version of Bubble Sort

```
// Utility function that will swap values at 2 indices in array
public void swap(int[] arr, int i, int j) {
    // increment count to count swap
    count++;
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Figure 4: Swap Function

2.2.2 Recursive

The recursive function calls itself $n-1$ times. The auxiliary space used is $O(n)$ for the call stack. The pseudocode is as follows:

```
// Recursive version of Bubble Sort
private void recursiveSort(int[] a, int n) {

    for (int i = 0; i < n - 1; i++) {
        count++;
        if (a[i] > a[i + 1]) {
            swap(a, i, i + 1);
        }
    }

    if (n - 1 > 1) {
        recursiveSort(a, n - 1);
    }
}

public int[] recursiveSort(int[] a) throws UnsortedException {
    timeStart = System.nanoTime();

    int n = a.length;
    recursiveSort(a, n);

    timeEnd = System.nanoTime();
    return a;
}
```

Figure 5: Recursive Version of Bubble Sort

2.3 Big-Θ Analysis

The algorithm is simple, but impractical because the efficiency decreases dramatically on lists of more than a small number of elements. The advantage to bubble sort is the ability to detect if the list is already sorted; than Big-Θ is $O(n)$ (Bubble sort Algorithm 2019). In iterative and recursive have the same number of critical operations by avoiding looking at the last element because its already in the final place. Since our array has 10 elements there will be 9 iterations as follows: $n-1, n-2, \dots, n-9$. The average and worst-case time complexity is $O(n^2)$, the worst case happens when the array is reverse sorted.

2.4 JVM Warm-up

This method helps load classes to ensure that the objects are initialized before running the main function. The class ManualClassLoader is the first request made to initialize the warm-up. A Dummy class is than declared and the ManualClassLoader class will be executed at least 500,000 times as soon as the application starts and with each execution. This will provide a more accurate runtime of the critical counts versus including the time it takes for everything to initialize.

2.5 Critical Operations

The critical operations that I chose to count are the number of times an element checks to be swapped and the actual swap of the two elements. An example would be with an array `int [] arr = {2,1,3}`. This would be sorted by bubble sort algorithm `arr = [1,2,3]`. The element `2` in `arr` would be compared with `1` and these elements need to be swapped. Then element `2` would compare with `3`, there would be a check to swap, but it is not needed, that concludes the first iteration.

3. Analysis

3.1 Graph of Critical Operations

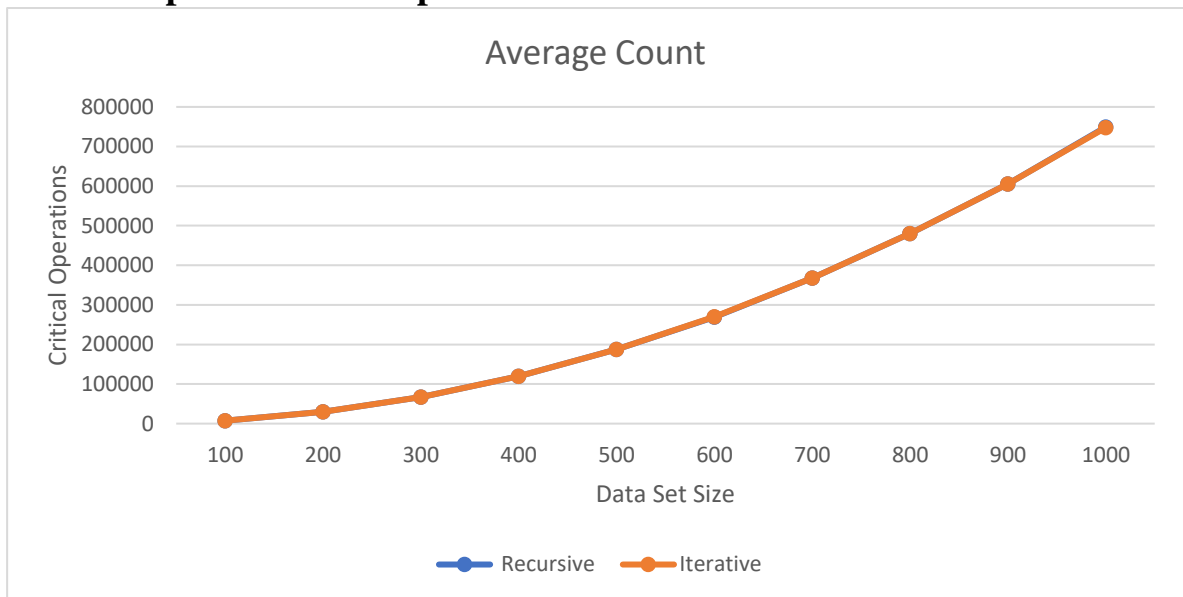


Figure 6: Average Count Graph

3.1.1 Count Evaluation

Figure 6 displays the average critical operations/count for each data set size. The data shows that the two versions of the bubble sort are similar in the average number of operations for each data set size. The Big- Θ analysis for bubble sort is $O(n^2)$ and from the data we can determine that this is true. Figure 6 and 2 show that $n=100$ for iterative and recursive versions the solution is $\approx 7,400$ and $100^2 = 10,000$. The data matches Big O notation $O(n^2)$ more accurately than the other notations.

3.2 Graph of Execution Time

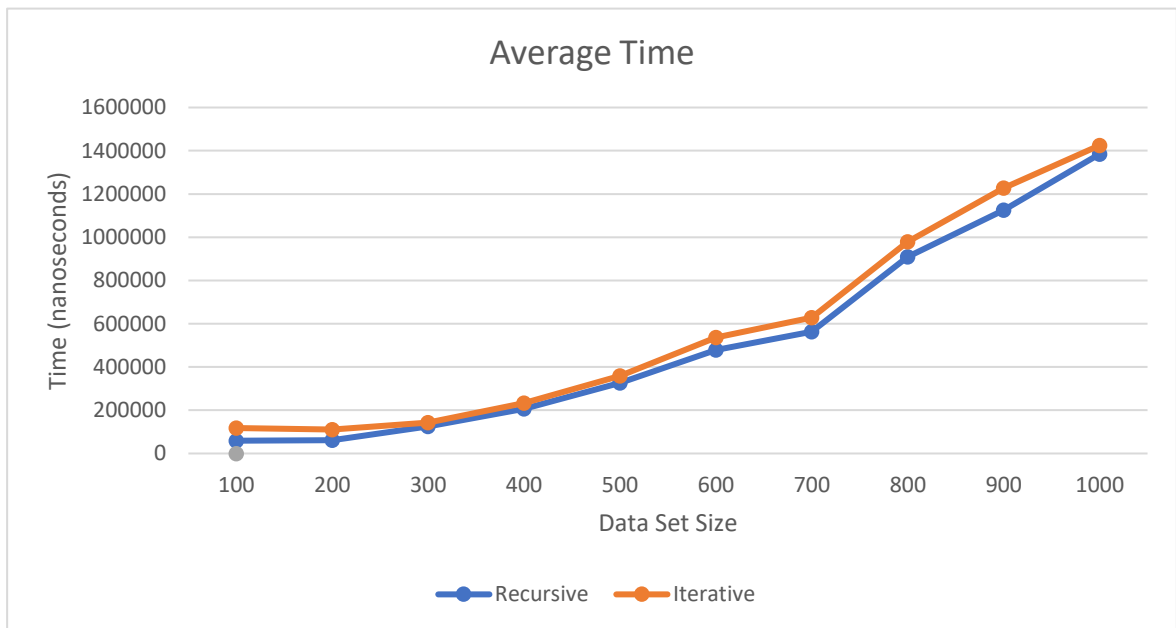


Figure 7: Average Time Graph

3.2.1 Time Evaluation

Figure 7 displays the average time it takes in nanoseconds for each data set size to sort. We can conclude that as the data set sizes increase so does the time for both iterative and recursive versions of the bubble sort.

3.3 Performance

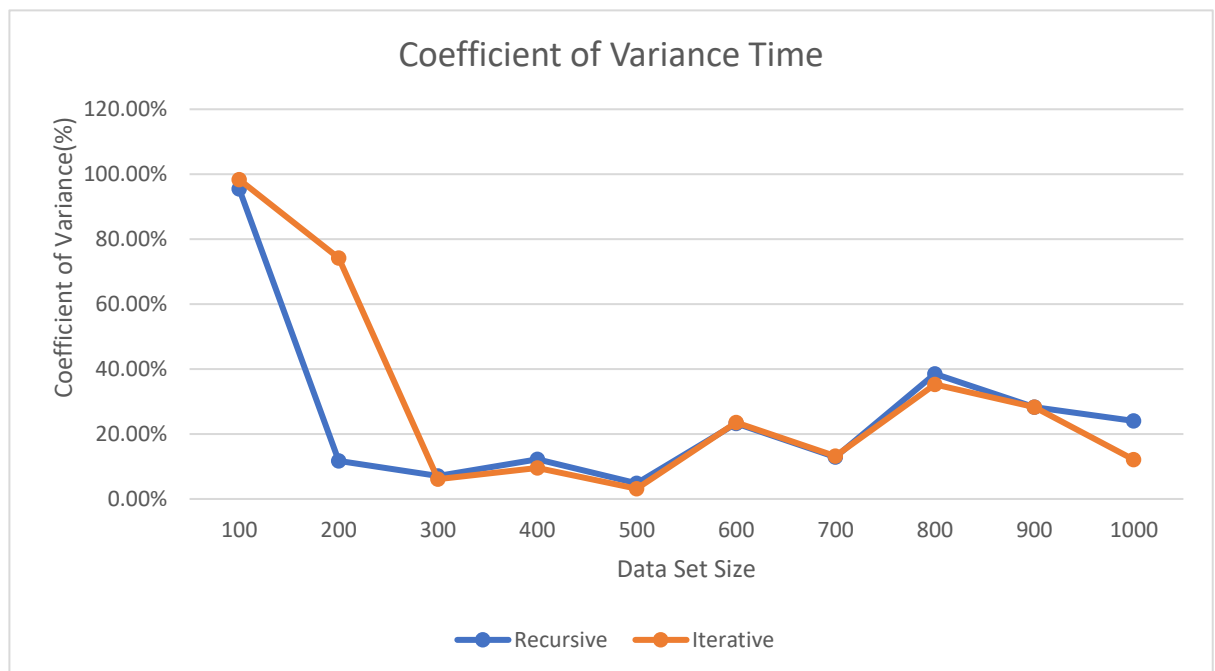


Figure 8: Coefficient of Variance Time Graph

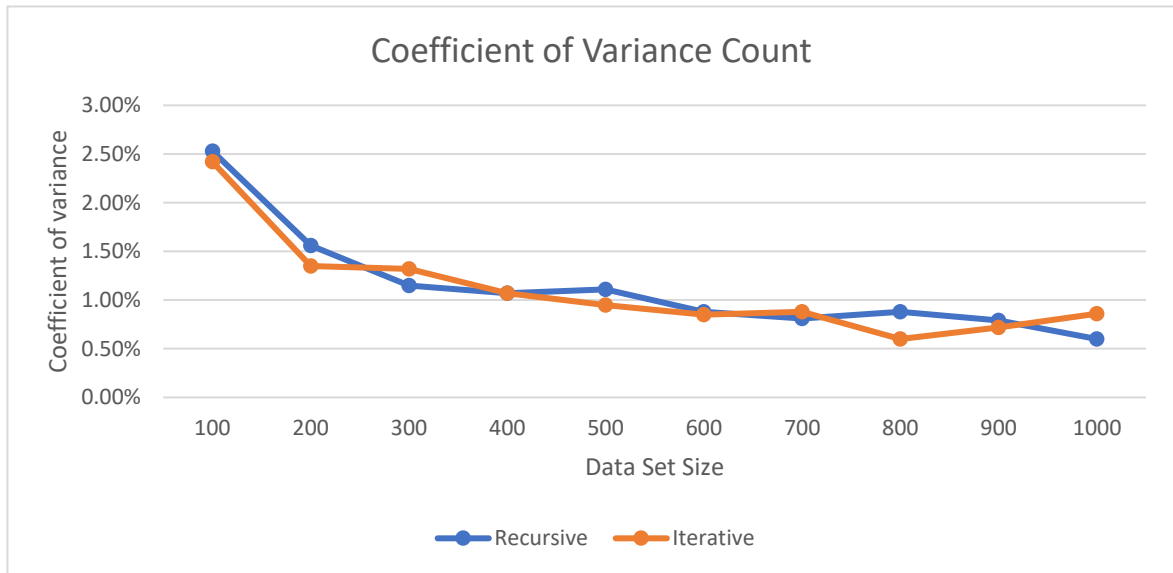


Figure 9: Coefficient of Variance Count Graph

3.3.1 Coefficient of Variance Evaluation

“The coefficient of variation represents the ratio of the standard deviation to the mean, and it is a useful statistic for comparing the degree of variation from one data series to another, even if the means are drastically different from one another” (Hayes, 2020). Figure 9 is comparing the deviation in time for each data set. Ideally, we will want to have a low variation. As you can see from the figure 9 iterative and recursive bubble sort follow a descending order as the data sets increase, meaning the deviations are getting smaller.

3.3.2 Comparing Coefficient of Variance to Big- Θ

On figure 8 and 9 the smaller data set size shows higher deviations in count and time, proving to be very unstable at smaller data sizes. The coefficient of variance for time was upwards of 100% at 100 data set size. While the coefficient of variance for count was around 2.50%. Concluding that the execution time is way more unstable than that of critical operations. As the data set sizes increases so does the stability of critical operations and the time is getting better but it is still unstable with over %20 towards the end of the data sets.

3.4 Critical Operation and Execution Time

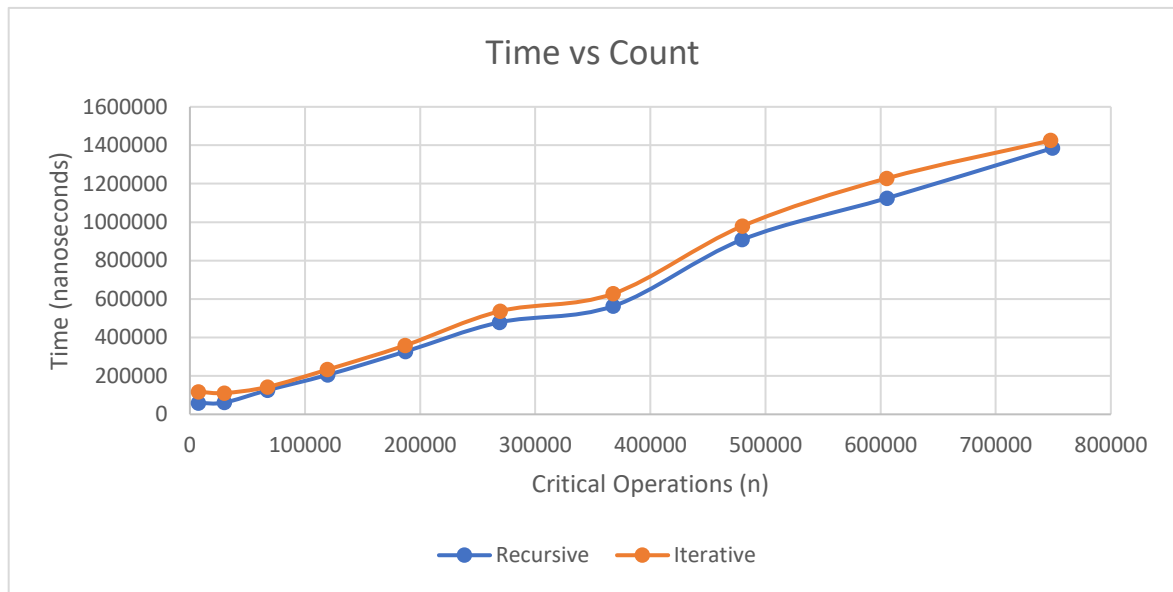


Figure 10: Time versus Count Graph

3.4.1 Time and Count Evaluation

Figure 10 displays the time it takes in nanoseconds versus the average number of critical operations for each data set. We can conclude that both iterative and recursive versions for time and count grow at the same rate.

4. Conclusion

Overall, bubble sort is an easy algorithm to understand but it has poor performance and should be avoided if there are large collections of data. It is a good educational tool but there are algorithms that have better average-case complexity than $O(n^2)$. Bubble sort does have a cool advantage and that is being able to recognize that the list is already sorted with a time complexity of $O(n)$. From the data we can determine that both iterative and recursive versions behave very similar with bubble sorting method and follow a time complexity of $O(n^2)$. Also, the variance showed us that the count became more stable as data increased but the execution time was unstable throughout.

Reference:

Data Structure - Bubble Sort Algorithm. (n.d.). Retrieved April 26, 2020, from https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

Bubble sort Algorithm. (2019, June 10). Retrieved April 30, 2020, from <https://www.techiedelight.com/bubble-sort-iterative-recursive/>

Hayes, A. (2020, February 05). Coefficient of Variation (CV) Definition. Retrieved from <https://www.investopedia.com/terms/c/coefficientofvariation.asp>