Ashcon Abae

Design & Analysis of Computer Algorithms

CMSC 451 7981

15 December 2019

Project 2

For Project 1, I focused on implementing 2 versions of Bubble Sort: an iterative solution and a recursive solution. When sorting an unsorted array, Bubble Sort starts with comparing the first 2 elements of the array and swaps them if they the first element is greater than the second. If not, Bubble Sort now compares the 2nd and the 3rd elements of the array, again swapping them if the 2nd element is greater than the 3rd element. Bubble Sort continues this process until it reaches the end of the array. This is 1 iteration. Bubble Sort will perform as many iterations as necessary until the array is fully sorted after the last iteration. We can view the pseudocode for Bubble Sort below:

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

      /*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for

end procedure return list
```

Figure 1: Bubble Sort Pseudocode

Specifically, the code I wrote for each version of Bubble Sort can be found below:

```
/* * *
 * 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 2: Iterative Version of Bubble Sort

```
/* * *
 * 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 3: Recursive Version of Bubble Sort

Please keep in mind that the Θ analysis for both versions of Bubble Sort will be the same as they both have the same number of critical operations. In Bubble Sort, n-1 comparisons will be done on the

first iteration, n-2 in the 2nd iteration, n-3 in the 3rd iteration, and so on. This would show that our worst-case would be when Bubble Sort receives a reverse-sorted array, which would take $O(n^2)$. Our best-case would be when Bubble Sort receives a sorted array, which would take $\Omega(n)$. On average, Bubble Sort takes $\Theta(n^2)$ because the number of bad-cases Bubble Sort receives on average outweighs the number of good-cases Bubble Sort receives.

Warming up the JVM is important because the first request made to a Java web application is substantially slower than the average response time during the lifetime of the process. To avoid this issue, we need to cache all classes beforehand so that they're available instantly when accessed at runtime.

For Bubble Sort, there are 2 critical operations we need to keep track of: checking to see if 2 indices need to be swapped, actually making the swap of 2 indices once determined that it's needed. These 2 operations are critical to be counted because everything else will take constant time to complete. These 2 operations are the only operations that will be completed in variable time, so we must keep track of them.

	Iterative				Recursive			
Data Size	Average Operation Count	Coefficient Variance of Count	Average Execution Time	Coefficient Variance of Time	Average Operation Count	Coefficient Variance of Count	Average Execution Time	Coefficient Variance of Time
100	7384.64	169.32	0.14	0.35	7384.64	169.32	0.04	0.20
200	29821.12	551.60	0.16	0.37	29821.12	551.60	0.08	0.34
300	67296.42	751.73	0.12	0.33	67296.42	751.73	0.18	0.39
400	119725.64	1180.13	0.22	0.42	119725.64	1180.13	0.24	0.43
500	187164.88	1942.93	0.34	0.48	187164.88	1942.93	0.40	0.49
600	269758.54	2391.12	0.42	0.50	269758.54	2391.12	0.54	0.50
700	366296.14	3236.77	0.58	0.50	366296.14	3236.77	0.64	0.48
800	479362.84	4201.97	0.72	0.45	479362.84	4201.97	0.80	0.40
900	607186.38	4872.70	0.90	0.30	607186.38	4872.70	1.04	0.20
1000	748995.98	5450.49	1.08	0.27	748995.98	5450.49	1.22	0.42

Figure 4: Iterative vs Recursive Results

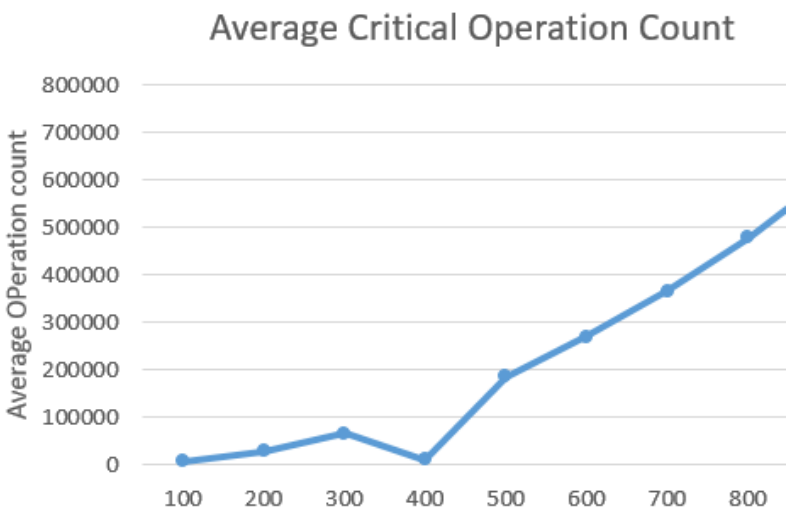

Figure 5: Graph of Critical Operations for both Iterative and Recursive

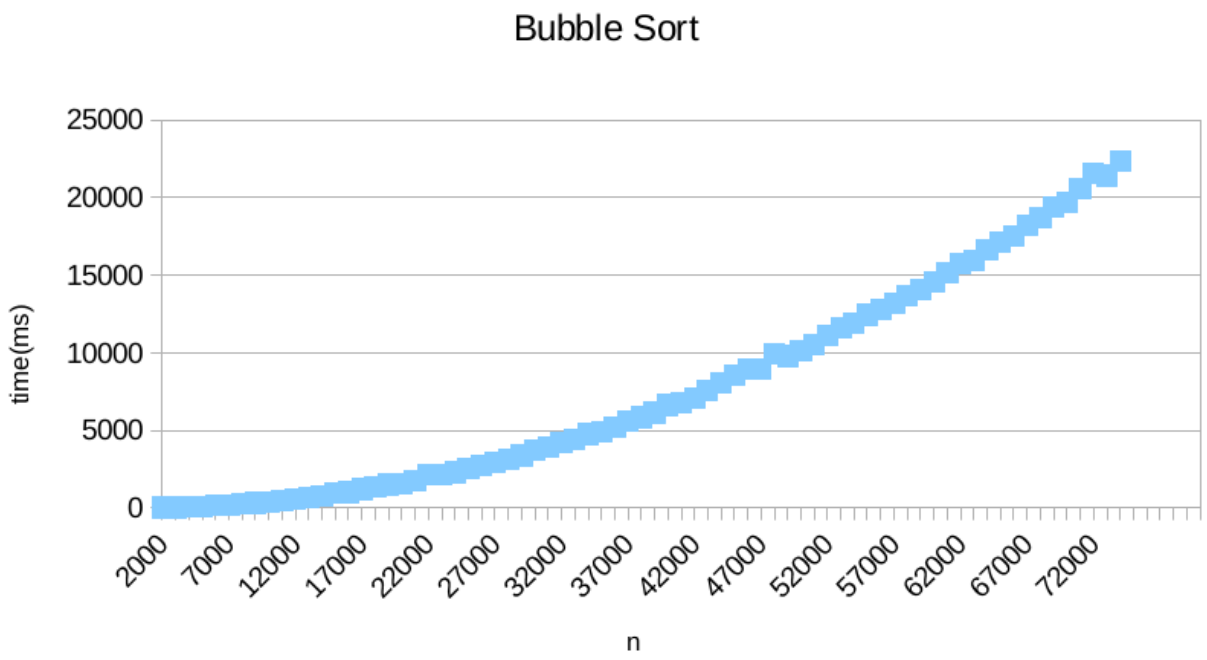

Figure 6: Graph of execution times for both iterative and recursive

As we can see in Figure 4, the Average Operation Count for both the recursive and iterative versions of Bubble Sort are the same, because both versions of the algorithm make the same number of critical operation counts. We can, however, see that the average execution time for relatively-smaller data size arrays is smaller in the recursive version when compared to the iterative. That being said, as data size increases, the iterative version average execution time becomes lower than that of the recursive version. Coefficient Variance of Time is significant when comparing the results because ideally we want low coefficient variances with not a lot of variance when comparing different data sizes. We can see that these results closely align with out calculation of Bubble Sort computing in $\Theta(n^2)$. For example, the average operation count for a data size of 100 for both the iterative and the recursive solution is 7384.64. This is significantly closer to $n^2 = 100^2 = 10{,}000$ than $n = 100$. We can see this trend throughout all our data size samples in the results of Figure 4.

In conclusion, Bubble Sort will tend to take $\Theta(n^2)$ time. There are 2 critical operations to keep track of with Bubble Sort: checking to see if 2 elements need to be swapped, and actually swapping the 2 elements if determined that they need to be swapped. The recursive version of my algorithm tends to perform better with relatively-small data sizes, while the iterative version of my algorithm tends to perform better with relatively-large data sizes.