

Storyboard of Program Execution

To get start the process, first run BenchmarkSorts.java. After a successful build it will produce two files, one with the recursive results, the other with the iterative results. Then you just need to run BenchmarkReport.java and choose the file whose data you would like to see displayed.

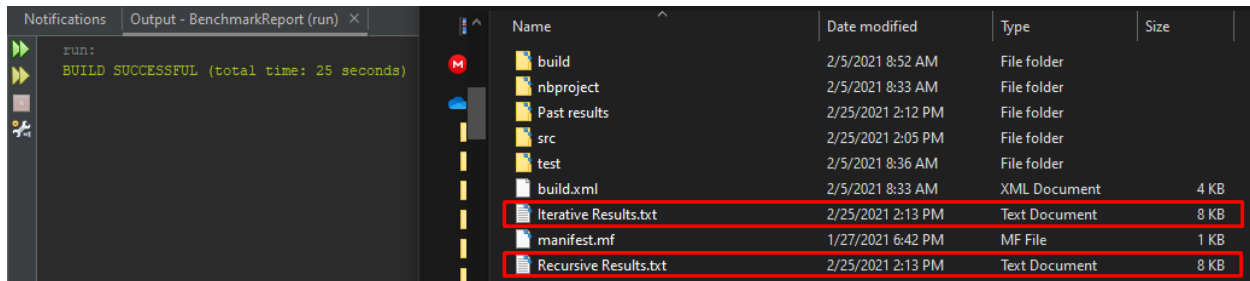


Figure 1. Proof of successful compilation of BenchmarkSorts.java and the files produced.

The screenshot shows a window titled 'Iterative Results.txt's Benchmark Report' containing a table with benchmark data for iterative sorting. The table has five columns: Size, Avg Count, Coef Count, Avg Time, and Coef Time. The data is presented for sizes ranging from 100 to 1000.

Size	Avg Count	Coef Count	Avg Time	Coef Time
100	2456.40	6.90%	1892.02	4.87%
200	9987.16	4.48%	6244.06	6.62%
300	22314.40	3.67%	13170.02	4.74%
400	40038.60	3.33%	23114.06	6.03%
500	62374.78	2.54%	35238.04	2.98%
600	89309.46	2.49%	50093.88	3.99%
700	122456.80	2.19%	68026.06	2.24%
800	159409.96	2.40%	88437.94	2.78%
900	203299.54	1.87%	113085.96	5.20%
1000	249197.52	2.01%	137038.10	2.21%

Figure 2. Proof of BenchmarkReport.java's successful run with Iterative Results.txt and the JTable with the data produced.

The screenshot shows a window titled 'Recursive Results.txt's Benchmark Report' containing a table with benchmark data for recursive sorting. The table has five columns: Size, Avg Count, Coef Count, Avg Time, and Coef Time. The data is presented for sizes ranging from 100 to 1000.

Size	Avg Count	Coef Count	Avg Time	Coef Time
100	2456.40	6.90%	4104.02	5.39%
200	9987.16	4.48%	14763.92	5.18%
300	22314.40	3.67%	32121.96	5.12%
400	40038.60	3.33%	56327.96	3.86%
500	62374.78	2.54%	90117.92	3.19%
600	89309.46	2.49%	143226.12	3.91%
700	122456.80	2.19%	220628.00	3.64%
800	159409.96	2.40%	318780.00	4.60%
900	203299.54	1.87%	446384.04	4.19%
1000	249197.52	2.01%	580079.90	3.06%

Figure 3. Proof of BenchmarkReport.java's successful run with Recursive Results.txt and the JTable with the data produced.

Introduction to Insertion Sort

The algorithm I chose for this project was insertion sort. Insertion sort works by running through an entire array starting with index 1. The value of the current index is compared to the value of the index before it, if the current value is less, then the two values will swap positions in the array. The current value will continue to be compared to the values before it until it reaches a point where it is greater than the value it is being compared to. Insertion sort works well with smaller data sets, with similarly efficiency to bubble and selection sort. But with larger data sets, insertion sort is much less efficient than other similar sorting algorithms like heap or quick sort (“Insertion Sort?”, n.d.).

Iterative Pseudocode:

```
for i = 1 to array.length:
    value = array[i]
    j = i
    while j > 0 and array[j - 1] > value:
        array[j + 1] = array[j]
        j--
    end while
    array[j] = value
end for
```

Recursive Pseudocode:

```
insert (int[] array, int i):
if i < array.length:
    value = array[i]
    j = shift(array, value, i)
    array[j] = value
    insert(array, i + 1)
end if
```

```
shift(int[] array, int value, int i):
insert = i
if i > 0 and array [i - 1] > value:
    array [i] = array [i - 1]
    insert = shift(array, value, i - 1)
end if
return insert
```

Analysis of Big-O

Looking at the pseudocode for the iterative version of the algorithm, there are two factors that determine time complexity, the for loop and the while loop. The for loop will run for n (length of array) $- 1$. The nested while loop will also run $n-1$ times (until j reaches 0). This leaves us with $(n-1)^2/2 = O(n^2)$ for the worst and average cases of the iterative version.

As for the recursive version, we see that the first recursive method, `insert()`, will run for n (length of array) $- 1$. For every call of `insert()`, it makes a call to `shift()`. If the conditions are met (a swap needs to occur), then `shift()` will make a recursive call to itself until the conditions are not met.

This means in the worst and average cases, `shift` will be called as many times as the current index, which means its time complexity for the recursive version is also $O(n^2)$.

JVM Warmup and Avoiding Associated Problems

My approach to avoid problems associated with JVM warm-up was to create a loop that ran 200 times. For each iteration of the loop, a class of type `BenchmarkSorts` is created and then the function that creates all of the arrays with varying data sizes was run. I tried different numbers (100, 200, 500, and 1000) to see how many loops the JVM Warmup required to produce the best results. The most notable difference was going from no warmup to the 100 warmups. I found a small increase going from 100 to 200, but the difference between 200 and anything above it was marginal so I decided to keep it at 200.

Iterative Results.txt's Benchmark Report					Recursive Results.txt's Benchmark Report				
Size	Avg Count	Coef Count	Avg Time	Coef Time	Size	Avg Count	Coef Count	Avg Time	Coef Time
100	2507.98	7.72%	71242.10	80.47%	100	2507.98	7.72%	14064.04	177.83%
200	9988.78	4.30%	102815.96	35.67%	200	9988.78	4.30%	119800.06	553.61%
300	22501.06	2.88%	58691.94	52.91%	300	22501.06	2.88%	46523.90	37.88%
400	39857.18	3.31%	23065.98	3.74%	400	39857.18	3.31%	58146.00	4.72%
500	62060.22	3.06%	34722.00	3.91%	500	62060.22	3.06%	95165.96	9.01%
600	89639.16	2.92%	49316.12	3.05%	600	89639.16	2.92%	150971.98	7.22%
700	122098.92	2.67%	98465.92	30.58%	700	122098.92	2.67%	337464.04	31.59%
800	159700.46	2.53%	99024.02	15.67%	800	159700.46	2.53%	376342.04	39.96%
900	201193.68	2.08%	120259.96	4.03%	900	201193.68	2.08%	429109.90	6.05%
1000	250773.70	1.92%	151875.80	11.04%	1000	250773.70	1.92%	573481.98	5.23%

Figure 4. Algorithm results without any warmup.

Iterative Results.txt's Benchmark Report					Recursive Results.txt's Benchmark Report				
Size	Avg Count	Coef Count	Avg Time	Coef Time	Size	Avg Count	Coef Count	Avg Time	Coef Time
100	2456.40	6.90%	1892.02	4.87%	100	2456.40	6.90%	4104.02	5.39%
200	9987.16	4.48%	6244.06	6.62%	200	9987.16	4.48%	14763.92	5.18%
300	22314.40	3.67%	13170.02	4.74%	300	22314.40	3.67%	32121.96	5.12%
400	40038.60	3.33%	23114.06	6.03%	400	40038.60	3.33%	56327.96	3.86%
500	62374.78	2.54%	35238.04	2.98%	500	62374.78	2.54%	90117.92	3.19%
600	89309.46	2.49%	50093.88	3.99%	600	89309.46	2.49%	143226.12	3.91%
700	122456.80	2.19%	68026.06	2.24%	700	122456.80	2.19%	220628.00	3.64%
800	159409.96	2.40%	88437.94	2.78%	800	159409.96	2.40%	318780.00	4.60%
900	203299.54	1.87%	113085.96	5.20%	900	203299.54	1.87%	446384.04	4.19%
1000	249197.52	2.01%	137038.10	2.21%	1000	249197.52	2.01%	580079.90	3.06%

Figure 5. Algorithm results with the 200 JVM warmup.

Looking at the figures above, the most notable decrease from the JVM warmup is the average time of the Iterative insertion sort for an array of size 100, it went from 71,242.10 to 1892.02 which is a 97.34 decrease. Across the board, there is a notable decrease in average time and therefore coefficient time when comparing the results of sorts performed with no warmup to the ones that were warmed up.

Critical Count Operation Choice

When deciding where to place the critical count, I knew the best place in the iterative version would be in the while loop that is nested in the for loop. The reason for that was that whenever the while loop is running, it means that swaps are occurring. It took me a little longer to determine the correct place to put the critical count for the recursive sort. At first, I mistakenly had it outside of the if statement `shift()`, but I quickly realized that it should be inside the if statement. When it was outside of the if statement, the count was being incremented even if the value of insert was returned instantly (no swaps occurring). So, putting it in the if statement which is where the swap happens was the best gauge for the critical count. Once I saw that the average counts for both the iterative and recursion sorts were identical, I knew I had the critical counts in their correct places.

Graph of Critical Operations and Average Time

Because the critical operation count for both algorithms was identical, I only felt the need to include one graph for it which can be seen below:

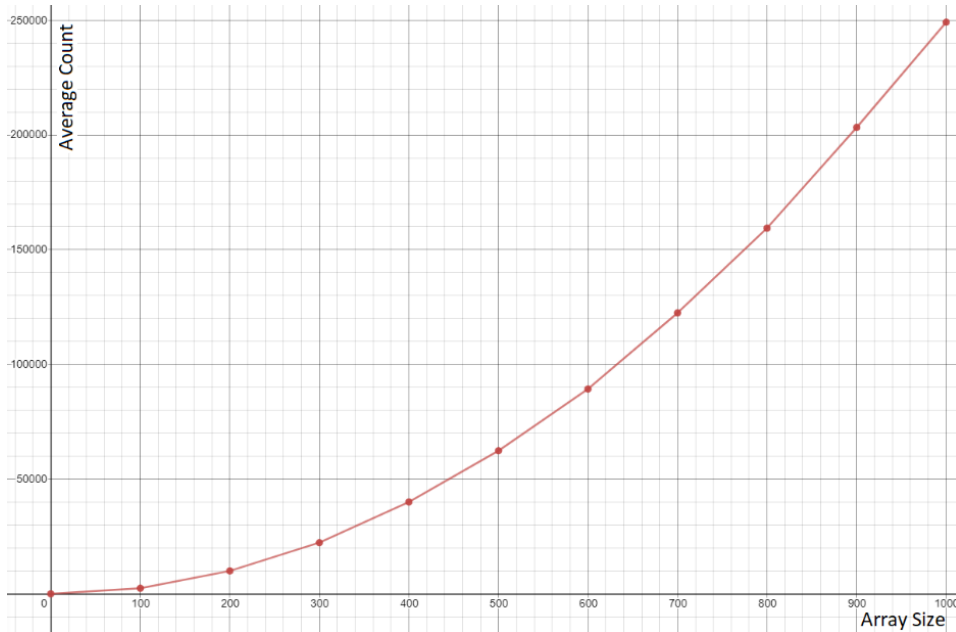


Figure 6. Graph of average critical counts (identical for iterative and recursive).

While the critical count for both algorithms was identical, their average times differ significantly. Although their Big O's are identical, there is a huge disparity in actual execution times:

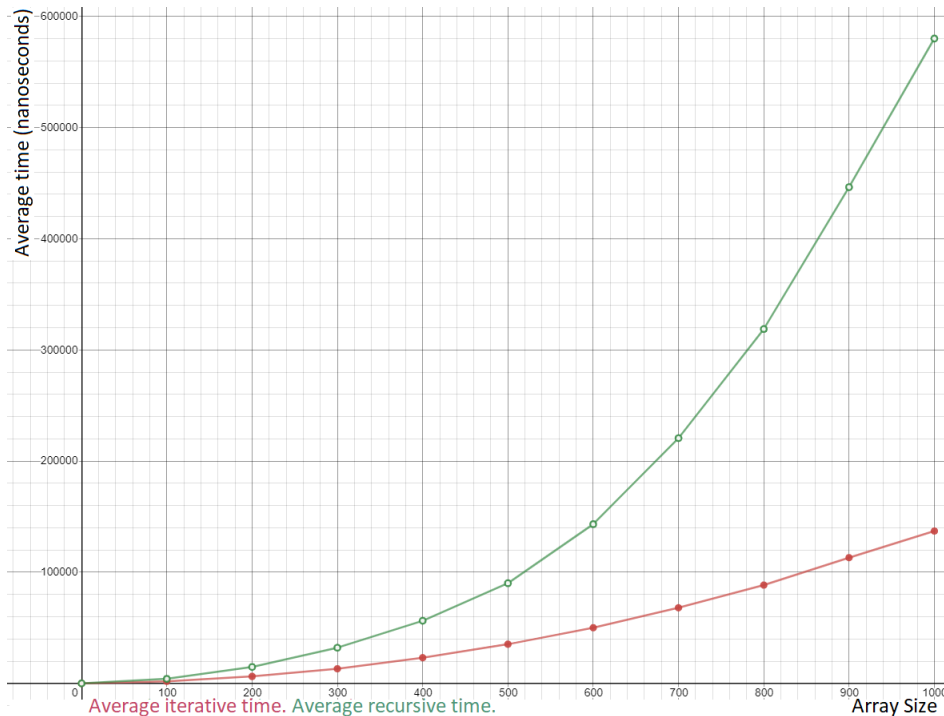


Figure 7. Graph of average time.

Comparison of Critical Operation Results and Actual Execution Time Measurements

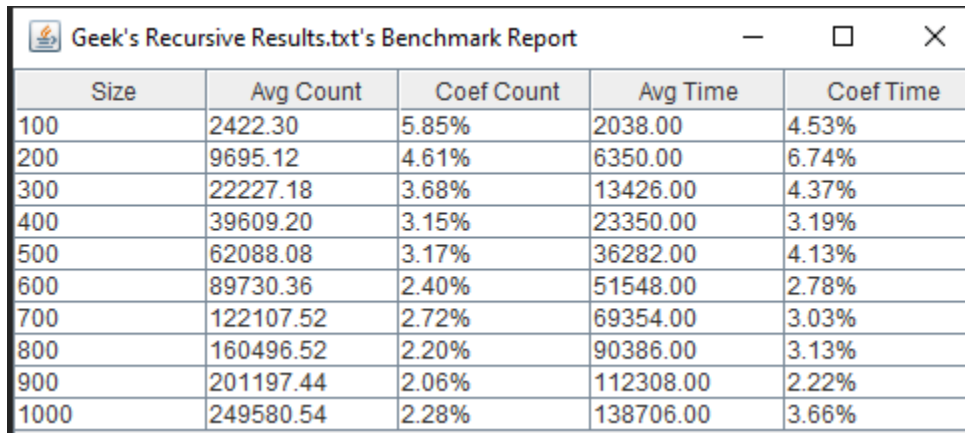
When comparing performance in terms of critical operations, both algorithms are identical so there is no winner in that regard. The Big O's for both algorithms are identical, so in theory they should also have the same theoretical average time. However, when taking actual average time into account, the iterative version clearly beats the recursive version. Based on the average time of multiple runs, the recursive version will take 2-4 times longer (in nanoseconds) than its iterative counterpart. The reason for this is that recursive calls will reserve additional space on the stack until all calls are completed which requires more memory and time taken (Dale, 2018). To make it worse, recursive version also has two recursive methods, `shift()` and `insert()`. Furthermore, the former is nested in the latter which increases the disparity of average time to its iterative counterpart even further.

One conclusion I came to is that the recursive algorithm I used was incredibly inefficient. For the sake of proving this point, I found another recursive insertion sort algorithm online and transplanted it into my code to see if it ran faster. The figure below shows the code:

```
int[] recursive(int arr[], int n) {
    if (n > 1) {
        recursive(arr, n - 1);
        int last = arr[n - 1];
        int j = n - 2;
        while (j >= 0 && arr[j] > last) {
            criticalCount++;
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = last;
    }
    return arr;
}
```

Figure 8. Recursive algorithm taken from: <https://www.geeksforgeeks.org/java-program-for-recursive-insertion-sort/?ref=rp>

After running the algorithm, I found that the results produced were much closer to its iterative counterpart. The recursive algorithm from GeeksForGeeks' average time was found to only be about 1-3% slower than the iterative version, which is much better than the 200-400% increase from the original recursive algorithm used. Proof of the other algorithm's results:



Size	Avg Count	Coef Count	Avg Time	Coef Time
100	2422.30	5.85%	2038.00	4.53%
200	9695.12	4.61%	6350.00	6.74%
300	22227.18	3.68%	13426.00	4.37%
400	39609.20	3.15%	23350.00	3.19%
500	62088.08	3.17%	36282.00	4.13%
600	89730.36	2.40%	51548.00	2.78%
700	122107.52	2.72%	69354.00	3.03%
800	160496.52	2.20%	90386.00	3.13%
900	201197.44	2.06%	112308.00	2.22%
1000	249580.54	2.28%	138706.00	3.66%

Figure 9. Benchmark report of GeeksForGeeks' more efficient recursive algorithm.

Performance Comparison (Inefficient Recursion Algorithm)

Both the recursive and the iterative versions had identical counts when it came to average critical operation results and critical coefficient count. As mentioned in the previous section, because recursion requires additional stack space reservation, the iterative version is more effective when it comes to average time. This also means that the recursive version uses much more memory, making the iterative version the clear winner. Even when taking a more efficient recursive algorithm into account (the one from GeeksForGeeks), the iterative version would still win out because recursive calls reserve additional stack space which requires more memory and a slight increase in average time.

Coefficient of Variance Results and Data Sensitivity

On smaller data sets (size 10-100) I found that the coefficient of variance for both time and critical count was much larger, ranging from 7-40%. However, for the data sizes greater than 100, the coefficient decreased in relation to the increase in size of the data. For example, sizes 100-400 had an average of a 5-7% coefficient. Data sizes of 500-100 mostly ranged from 2-4%, but on occasion certain sizes would spike up to 5-8%. Based on this data and speaking strictly in terms of data sensitivity, insertion sort performs best with larger data sets because that is when it is able to keep the coefficient of variance low.

Big-O Analysis

Because the recursive and iterative version of insertion sort have identical Big-O, it would be assumed that they should have similar real execution times. However, because of the inefficiency of the recursive algorithm chosen, there is the vast difference of real execution times. The iterative versions results follow most closely to the theoretical complexity of $O(n^2)$, while the recursive version's actual execution time appears to be much greater than its theoretical complexity.

Conclusion

Warming up the JVM was an essential factor in ensuring a real benchmark and valid data. Both versions proved that insertion sort works best with larger data sets when it comes to preserving data sensitivity and keeping the coefficient of variances low. That being said, it comes as no surprise that insertion sort works fastest/best with smaller data sets despite the resulting increase in the coefficient of variance. While their theoretical time complexity was identical, the nature of recursion combined with the inefficient recursive algorithm that utilized two recursive functions led to a huge difference in actual execution times.

References

Dale, W. (2018, May 09). Recursion: The Pros and Cons. Retrieved March 9, 2021, from

<https://medium.com/@williambdale/recursion-the-pros-and-cons-76d32d75973a>

Insertion Sort (n.d.). Retrieved March 9, 2021, from

<https://www.techopedia.com/definition/20039/insertion-sort>