Merge Sort Analysis

CMSC 451 Project 2

Laudro Pineda

Dec 14, 2020

Merge Sort Analysis

In this paper, we explore further the initial selection of Project 1 which was the *merge-sort* algorithm, which is a process in which the input is divided in two halves, calls itself, and then merges the two-sorted halves in what is known as a 'divide and conquer'(D&C) method. The following pseudocode attempts to explain how *merge-sort* will sort *arr[](*array) by dividing *l* (left) and *r* (right) down *m* (middle), arrange both halves by calling on itself(recursive) and merge them at the end:(N.A., 2020)

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Figure 1: Merge-Sort pseudocode. Image retrieved from: https://www.geeksforgeeks.org/merge-sort/

Another way of understanding this top-down approach is through the following:

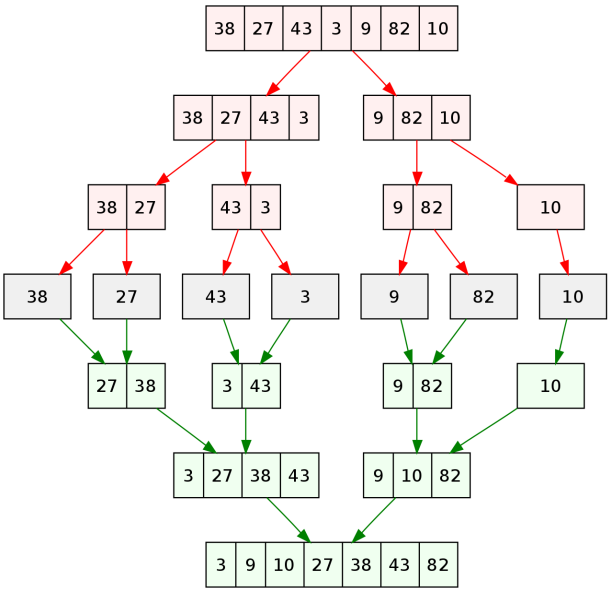


Figure 2: Recursive merge sort algorithm top-down diagram. Image retrieved from:

https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

While the diagram itself looks may look rather large, recursion tends to be pretty simplistic in the nature of calling itself, which begs the question, "Is this efficient in the modern day computer?" We would like to get the best and worst case scenario of the algorithm(known as *Big-O* notation), however we are looking for the 'sweet spot,' known as *Big-Theta(θ)* notation(Bell, 2009).

To find out *Big theta* we need to see how merge-sort is expressed in terms of algorithm and time complexity. Its D&C concept makes it go through 3 steps

- Divide which takes O(1) time.
- Conquer which recursively sort 2 subarrays of n/2 (for even n) elements
- Merge which takes O(n) time.

Taking it together shows us that Merge Sort's Big-theta is (n**log* n).

So, it looks like this might be the more efficient route to take, however one of the expectations of Project 1 was to look for the efficiency of iterative and recursive algorithms in a number of data set sizes. This also meant tackling the problem of JVM warm-up.

So JVM warm-up is when the most important classes of a Java program are used at the time of the process start. In this process, the classes are pushed into the cache making them faster during runtime.(Baeldung, 2018) This could explain why the data sets in Project 1 was very large as creating a large data set would allow a smaller set to run that much more faster.

```
public void recursiveSort(int[] list) throws UnsortedException {
    counter = 0;
    start = System.nanoTime();
    recursive(list, start: 0, list.length);
    duration = System.nanoTime() - start;
    //confirm sort
    if (!isSorted(list)) {
        throw new UnsortedException(Arrays.toString(list));
    }

}
```

Figure 3

Recursive sort starts with an implementation of *int[]* array

Iteration sort starts with an implementation of *int[]* array

```
public void iterativeSort(int[] list) throws UnsortedException {
    counter = 0;
    start = System.nanoTime();
    iterative(list);
    duration = System.nanoTime() - start;
    //confirm sort
    if (!isSorted(list)) {
        throw new UnsortedException(Arrays.toString(list));
    }
}
```

Figure 4

However, the implementation below shows how the difference between both methods:

```
 */
private void recursive(int[] list, int start, int end) {
    counter++; //counter for method execution
    counter++; //counter for comparison below
    if (end - start <= 1) {
        return;
    }
    int middle = start + (end - start) / 2;
    recursive(list, start, middle);
    recursive(list, middle, end);
    merge(list, start, middle, end);
}
```

The *recursive* function starts with providing a method to keep the count while starting its implementation of dividing center, left and then right followed by the merge(see *Figure 1* for *Merge-Sort* pseudocode).

Figure 5

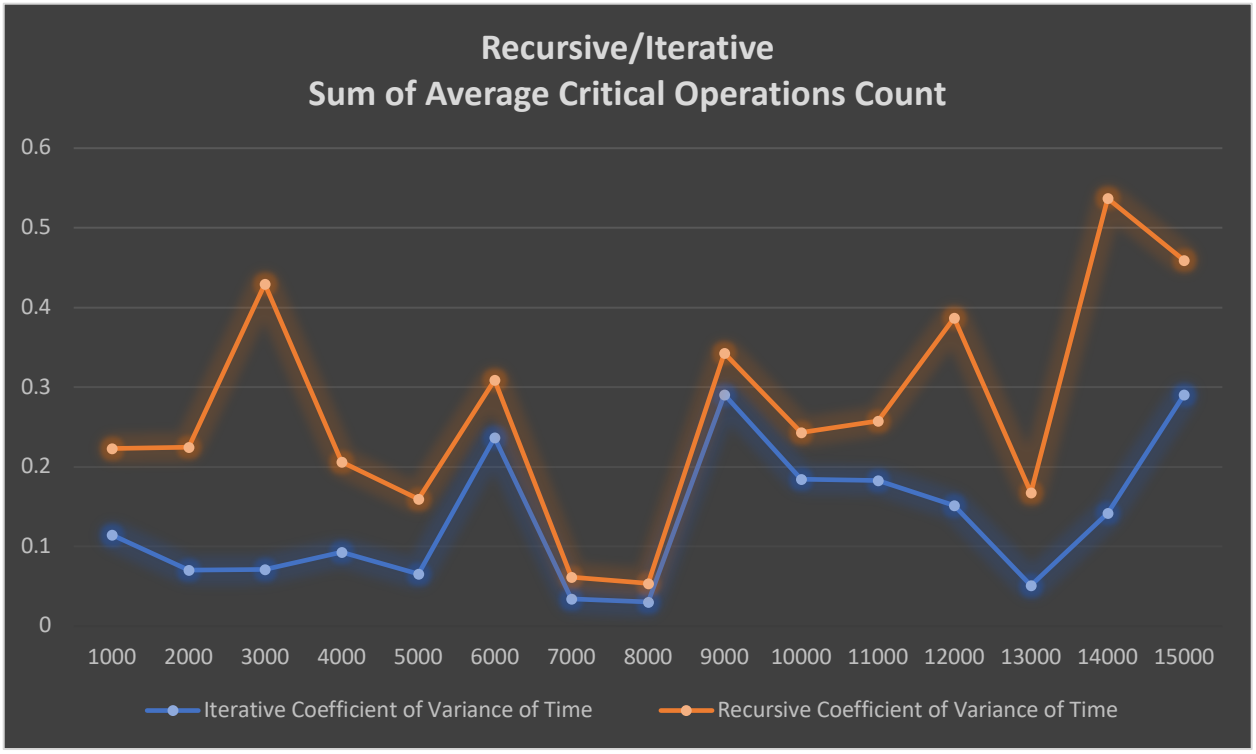
```
private void iterative(int[] list) {
    // The size of the sub-arrays. Constantly changing.
    int subSize = 1;
    // startL: start index for left sub-array
    // startR: start index for the right sub-array
    int startL, startR;

    while (subSize < list.length) {
        counter++; //counter for loop/comparison
        startL = 0;
        startR = subSize;
        while (startR + subSize <= list.length) {
            counter++; //counter for loop/comparison
            merge(list, startL,  middle: startL + subSize,  end: startR + subSize);
            startL = startR + subSize;
            startR = startL + subSize;
        }
        counter++;//counter for comparison below
        if (startR < list.length) {
            merge(list, startL,  middle: startL + subSize, list.length);
        }
        subSize *= 2;
    }
}
```

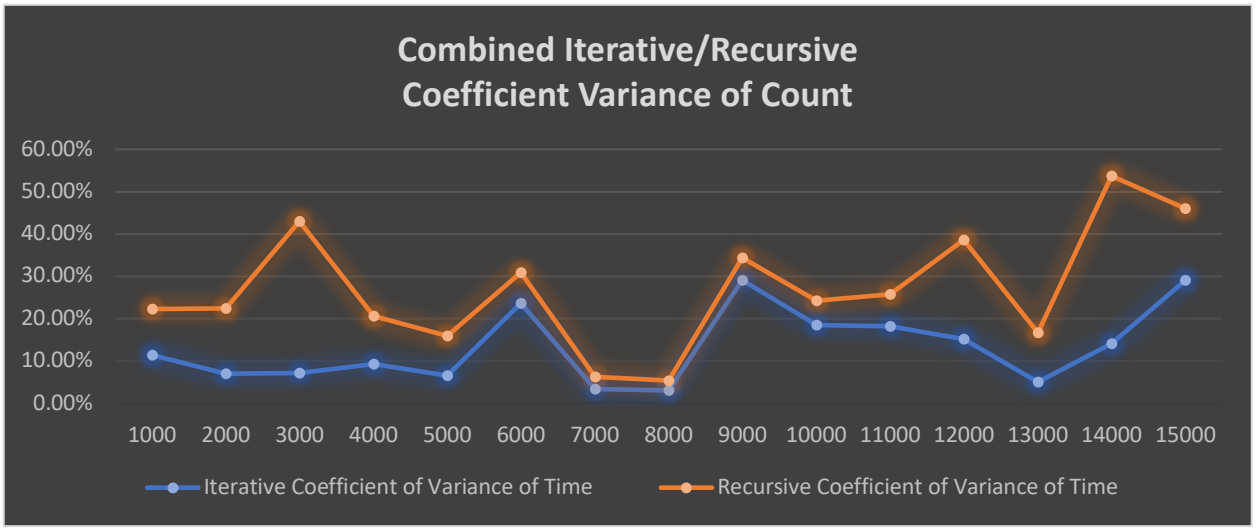
Figure 6

Iterative function goes through conditional statement to decide the termination of the recursion

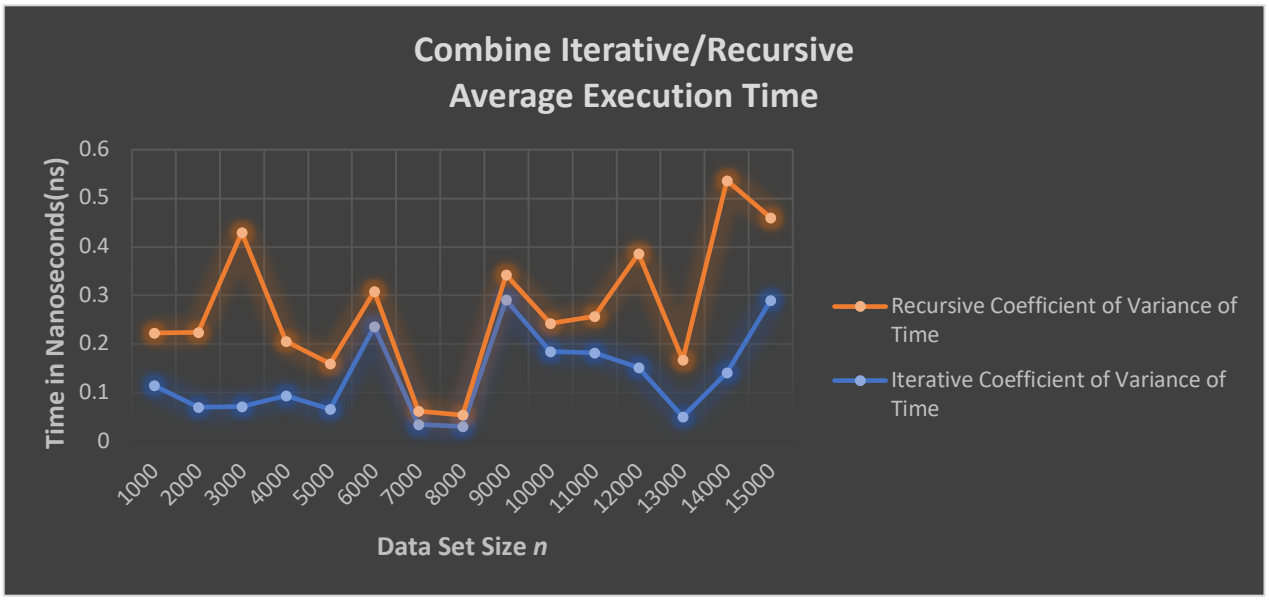
The following two graphs show the performance of the iterative function after JVM warm-up. The first graph shows the combined count average of critical operations for both the recursive and iterative function

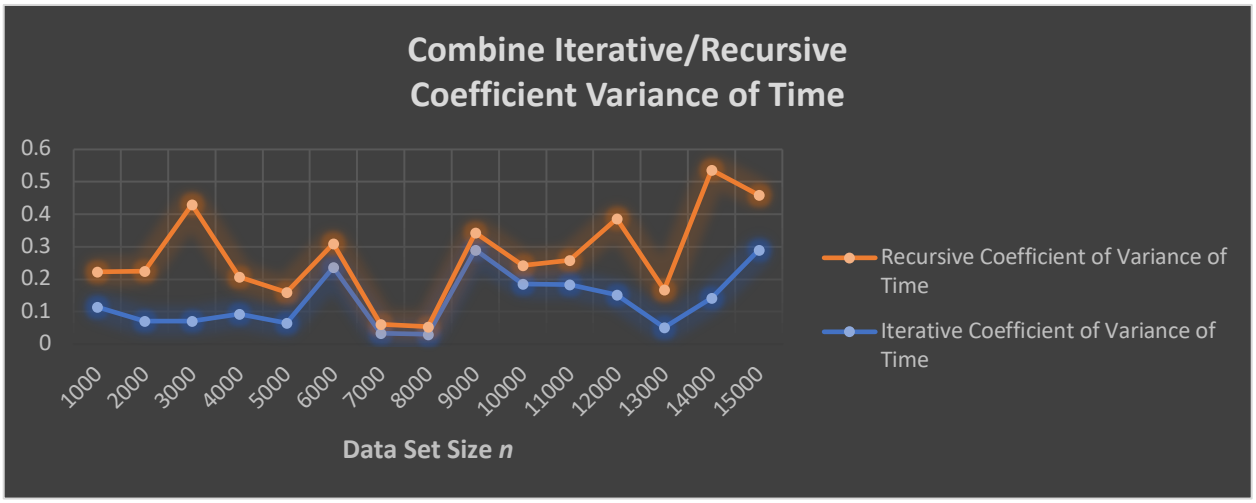


Remember that the coefficient of variation is considered a measure of relative variability; being the ratio of the standard deviation to the mean.(Glen, 2020)



A





Results:

Based on the empirical information provided by both iterative and recursive functions it is safe to say, based on initial analysis of execution time, that the iterative function performed well after JVM warm-up, due to not having to call itself even after being saved to cache. It also shows that in terms of time complexity in all 3 cases (worst, average, and best) it would be considered as $\theta (n * Log\ n)$.

Conclusion:

Prior to extracting the information and placing it in Excel, I was not quite sure that an iterative approach would be better than a recursive approach, especially, after performing Java warm-up would perform better. However, I did see the sense as to how, after continuously increasing data sets how, calling upon itself could prove very resource heavy causing considerable lag.

Works Cited

Baeldung. (2018, April 15). How to Warm Up the JVM. Retrieved December 15, 2020, from https://www.baeldung.com/java-jvm-warmup

Bolaji. (2018, February 27). Iteration vs Recursion. Retrieved December 15, 2020, from https://medium.com/backticks-tildes/iteration-vs-recursion-c2017a483890

Bell, R. (2009, June 23). A beginner's guide to Big O notation. Retrieved December 15, 2020, from https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Glenn, S. (2020, July 06). How to Find a Coefficient of Variation. Retrieved December 15, 2020, from https://www.statisticshowto.com/probability-and-statistics/how-to-find-a-coefficient-of-variation/

N.A. (2020, November 18). Merge Sort. Retrieved December 15, 2020, from https://www.geeksforgeeks.org/merge-sort/