CMIS 451 Project 2
Alex Hong
Professor Potolea
12.13.22

The referenced source code (https://levelup.gitconnected.com/sorting-algorithms-merge-sort-top-down-bottom-up-for-arrays-linked-lists-2426dcc39611.)
for iterative and recursive merge sort algorithms use the same merging function.

Pseudocode for Merge Function:

```
merge (array, low, mid, high) {
        int[] auxiliary array = int[high + 1]
        compare low to mid - the lower value gets added to auxiliary
        while (i <= mid && j <= high) {
                if (arr[i] < arr[j]) {
                        auxiliaryArr[k++] = arr[i++];
                } else {
                        auxiliaryArr[k++] = arr[j++];
                }
        }

        remaining values get added to auxiliary
        while (i <= mid) {
                auxiliaryArr[k++] = arr[i++];
        }
        while (j <= high) {
                auxiliaryArr[k++] = arr[j++];
        }

        rewrite original array with values from auxiliary array
        for (var r = low; r <= high; r++) {
                arr[r] = auxiliaryArr[r];
        }

}
```

Pseudocode for Iterative MergeSort:

```
int low = 0
int high = array.length
int mid = (low + high) / 2

Outer for loop at first merges 2 values, then 4, then 8 and so on until we reach the size
of the original array
iterative merge sort ( {
        for (increment = 2; increment < array length; increment *= 2){
                for (i = 0; i + increment - 1 < array length; i += increment) {
                        merge from i to increment
        }
}
```

Pseudocode for Recursive MergeSort:

```
recursiveMergeSort(int[] array, int low, int mid, int high)
set mid to low + high / 2
var mid = (low + high) / 2;
call recursiveMergeSort on array from low to mid
recursiveMergeSort(arr, low, mid);
call recursiveMergeSort on array from mid+1 to high
recursiveMergeSort(arr, mid+1, high);
merge the two halves of the array
Merge(arr, low, mid, high)
```

Big O Analysis for Iterative and Recursive MergeSort

Iterative and Recursive MergeSort will result in a time complexity of O(n log n). For recursive merge sort we divide the problems by resetting the “middle” value each time and recalling the recursive function resulting in log n time complexity. In iteration we use 2 for loops to set the increments of comparison which also results in log n time. The merge function can potentially compare each value if the list values are alternating (worst case) and gives a time complexity of n. Thus the time complexity is n log n. Recursion uses stack memory and will be slower for larger size data as compared to iteration.

JVM Warmup:

I warmed up the JVM by running the program 100 times using the same array of data sizes I used in the actual benchmark. I noticed that for smaller data sizes (16, 32, 64, 128, 256, 512) the average times for both iterative and recursive merge sorts were much higher if the JVM wasn’t warmed up.

With 100 warmups

data

size	iter counts avg	iter counts coeff	iter times avg	iter times coeff	recur counts avg	recur counts coeff	recur times avg	recur times coeff
16	45.74	4.16247590521...	666.28	19.2653156585...	45.74	4.16247590521...	655.92	16.4120057610...
32	121.36	2.31126556479...	1549.38	15.8781586991...	121.36	2.31126556479...	1458.02	7.44708433263...
64	304.76	1.51249653982...	3681.94	9.78208972429...	304.76	1.51249653982...	3401.42	5.34799623721...
128	736.44	0.72330584689...	8396.46	8.15191597111...	736.44	0.72330584689...	7790.84	2.53623927967...
256	1728.32	0.53240636787...	20085.3	9.32365525524...	1728.32	0.53240636787...	20387.16	12.1585641662...
512	3962.54	0.33066541564...	57382.08	7.47083859024...	3962.54	0.33066541564...	58863.7	7.94258145516...
1024	8943.2	0.21030926856...	201172.42	9.44348271044...	8943.2	0.21030926856...	206476.04	9.13831989604...
2048	19937.54	0.12490633007...	789755.96	18.3715696958...	19937.54	0.12490633007...	781767.08	12.5784580180...
4096	43973.02	0.09023619226...	2813623.1	8.33917999046...	43973.02	0.09023619226...	2816044.08	8.93513278012...
8192	96135.44	0.04788383252...	1.057930968E7	4.02374679194...	96135.44	0.04788383252...	1.068694732E7	3.98017336846...

Without 100 warmups

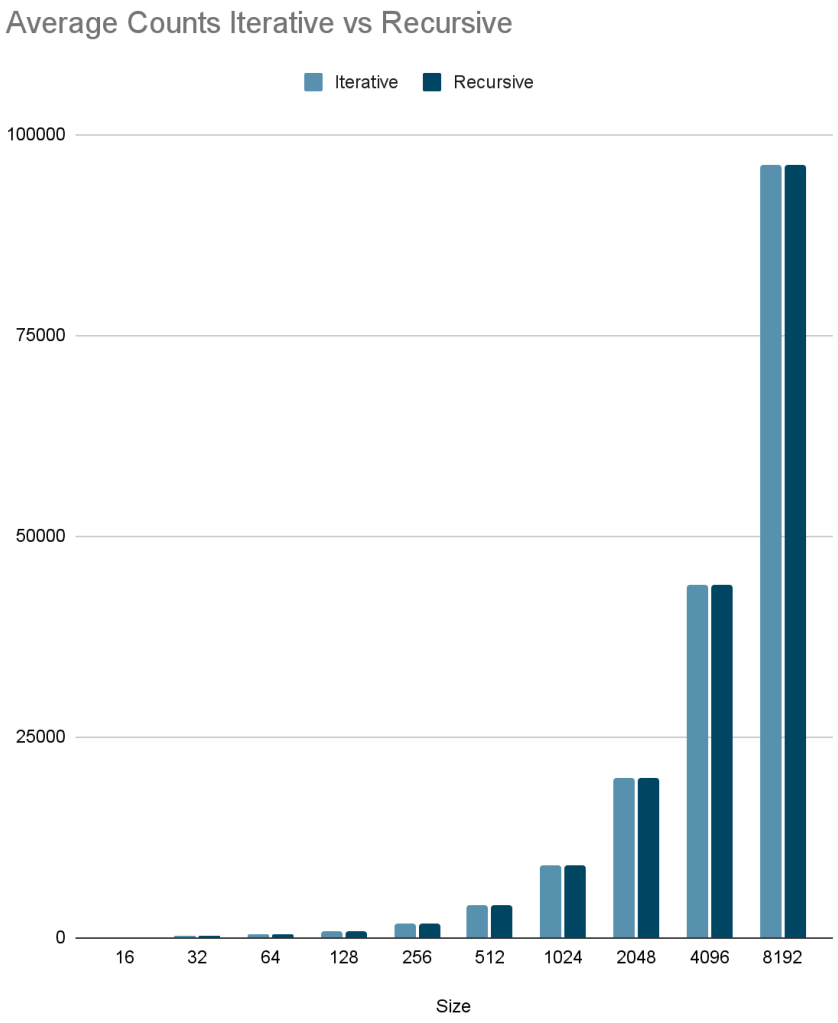
size	iter counts avg	iter counts coeff	iter times avg	iter times coeff	recur counts avg	recur counts coeff	recur times avg	recur times coeff
16	45.36	4.91469115725...	4919.7	54.1442495329...	45.36	4.91469115725...	3779.14	55.2435420845...
32	121.5	2.30554633170...	4110.0	43.2882905509...	121.5	2.30554633170...	4830.7	33.2921744846...
64	304.22	1.21420184495...	10317.4	12.0175891668...	304.22	1.21420184495...	10551.56	16.5381337455...
128	737.16	0.83550140514...	28494.22	5.38654219825...	737.16	0.83550140514...	27296.66	5.86743515140...
256	1725.06	0.47041929311...	69358.22	24.0883984407...	1725.06	0.47041929311...	92726.42	189.836214513...
512	3962.04	0.27063779232...	137685.4	55.7250937330...	3962.04	0.27063779232...	156084.6	96.3196493580...
1024	8945.7	0.23878605591...	509126.16	56.8842548297...	8945.7	0.23878605591...	528067.46	64.4196071836...
2048	19935.82	0.10186450062...	1162987.06	75.2517609028...	19935.82	0.10186450062...	1138565.3	68.1296323776...
4096	43978.44	0.07717695694...	3599599.1	66.2551460143...	43978.44	0.07717695694...	3569371.5	62.3749904322...
8192	96133.12	0.05531987709...	1.057178756E7	3.85967250866...	96133.12	0.05531987709...	1.061565086E7	3.23676197595...

For size 16, the average time for iterative merge sort with warm up was 666.28 ns.
For size 16, the average time for iterative merge sort without warm up was 4919.7 ns.
That's over 7 times the difference between warm up and non warm up for size 16.
For size 16, the average time for recursive merge sort with warm up was 655.92 ns.
For size 16, the average time for recursive merge sort without warm up was 3779.14 ns.
That's over 5 times the difference between warm up and non warm up for size 16.

Critical Operation:

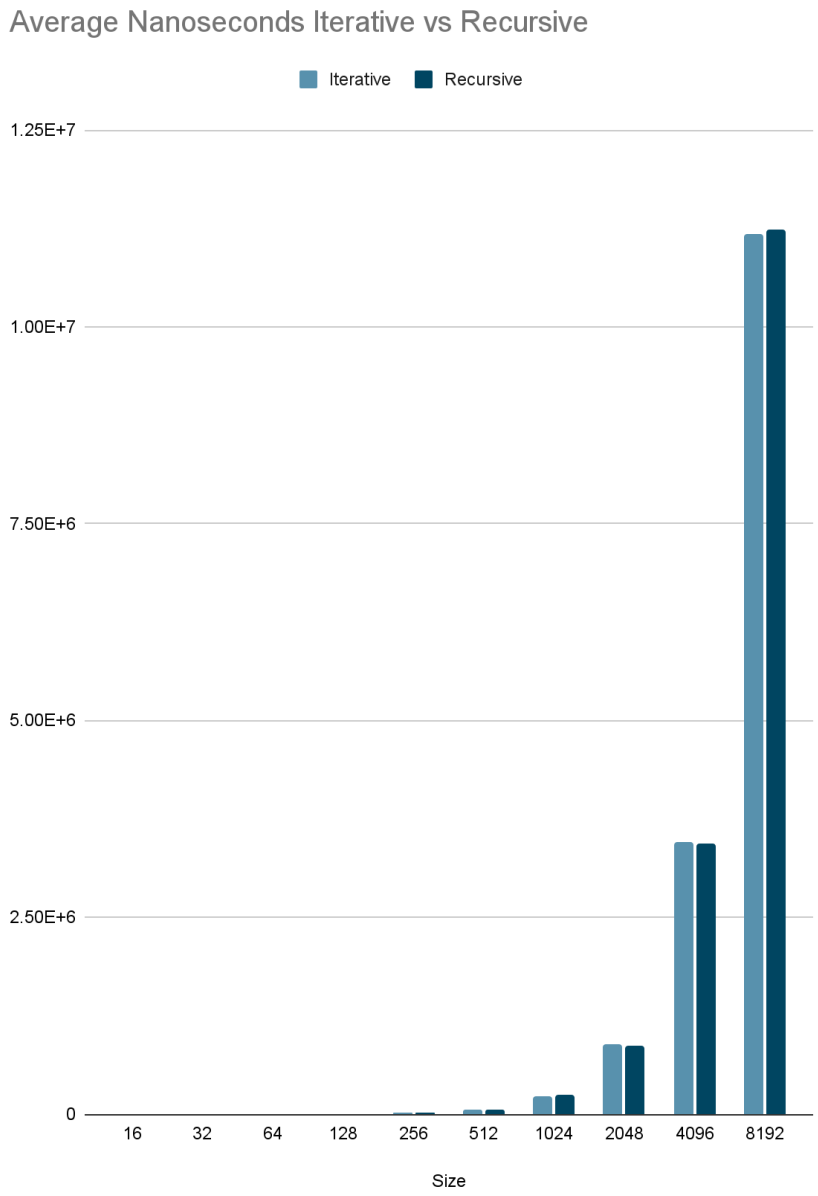
Since the iterative and recursive algorithms will always break the array down into smaller subproblems the number of times that will happen will always be log n. I decided to count both comparisons and assignments of indices of a subarray to the auxiliary sub array in the shared merge method. Additionally, since both the recursive and iterative use the same merge method, their critical operation count will be the same.

Critical Operations Average Graph: Y axis is counts average



This graph shows the average number of critical operations (y axis) as the data size gets larger. Since, in this program, both iterative and recursive merge sort use the same merge function, the average critical operations are the same for both.

Execution Times Average Graph:



This graph shows the average amount of nanoseconds it takes for the recursive and iterative functions to complete on each data set. The average times are very similar - iterative is slightly faster for data sets above 4000.

Performance Comparison:

Table of Average Times Comparison

Size	Iterative	Recursive
16	643.62	611.62
32	1526.2	1410.24
64	3382.76	3368.18
128	8103.4	7757.84
256	19840.02	19913.62
512	62727.7	65724.12
1024	222092.58	240760.44
2048	878033.52	863281.9
4096	3449419.26	3444146.76
8192	1.06E+07	1.07E+07

The iterative and recursive implementations of merge sort ran very similarly on my system (imac) in terms of execution time average. The above screenshot shows the average times of one of my run-throughs. The difference in nanoseconds between iterative and recursive is extremely small even as data gets larger. The only difference that comes to mind is that recursion uses the call stack and would thus use more memory for recursive calls as compared to the iterative version of merge sort. For this reason I would lean towards the iterative version, but the average times are so similar on my system that it doesn't make a huge difference performance wise for the data sizes that I've chosen.

Critical Operations/Execution Times Comparison:

The difference between the iterative and recursive merge sorts is the way that they break the problem down into sub problems. The critical operations occur during the merging of the sub arrays when we have to compare two values within a subarray and then assign them to an auxiliary array. If the list has many alternating values then more comparisons and assignments will have to be made. For example, if we are comparing two sub arrays - right and left, and the first index of the right subarray is greater than all of the indices of the left subarray, then we won't have to reference the other indices in the right sub array and simply pass all the values of the left subarray to the auxiliary array instead of continuously offering indices from the right subarray to compare to the left.

Significance of Coefficient of Variance:

Coefficient of variance differed in terms of average critical operation count and average execution time. Concerning average execution time, the coefficient of variance followed no discernable pattern, with varying coefficient variants across each data set. Concerning average critical operation counts, the coefficient of variance went down as the size of the data set increased. As data set sizes increase, the difference between critical counts across each merge sort lessens - leading to less deviation between the average counts of each execution.

Conclusion:

In my implementation of the referenced source code for iterative and recursive merge sort, I found that both implementations performed similarly in terms of average time of execution for

the data sets of sizes 16, 32, 64, 128, 256, 512, 1024, 2048, and 8192. The coefficient of variance for average time of execution was very high for each data set size with no coefficient of variance lower than 1. Concerning the average counts of critical operations, the coefficient of variance becomes smaller as data sizes grow larger. This means that as data sets grow larger in size, the deviation between average counts of critical operations becomes smaller and smaller.