CSCI242: Data Structures & Algorithms I

Assignment 2

Name: Zhuoyu (Benjamin) Yang

Original Timing Function

|  |  |  |  |
| --- | --- | --- | --- |
|  | One Million (almost sorted) | One Million (random) | Ten Million (random) |
| Merge Sort | Start: 16:39:40.921  End: 16:39:41.137  Duration: 0.216s | Start: 16:46:57.274  End: 16:46:57.479  Duration: 0.205s | Start: 18:29:13.960  End: 18:29:16.332  Duration: 2.327s |
| Quick Sort | Start: 16:51:39.369  End: 16:51:39.551  Duration: 0.182s | Start: 16:56:02.710  End: 16:56:02.889  Duration: 0.179s | Start: 18:10:29.524  End: 18:10:32.381  Duration: 2.857s |

Experimental Timing Function

|  |  |  |  |
| --- | --- | --- | --- |
|  | One Million (almost sorted) | One Million (random) | Ten Million (random) |
| Merge Sort | 0.21127s | 0.212959s | 2.36681s |
| Quick Sort | 0.173623s | 0.183974s | 2.75589s |

*These tables reflect the runtime I got from the timing functions.*

*In the first table, I used the timing function originally provided in Assignment 1, where I got the timestamp at the start and end of the algorithm and did the calculation for the duration manually.*

*In the second table, I used a Timer class that I wrote where it does the calculation of the duration in the destructor method. When the algorithm finishes running, it prints the duration it took to run. You told me to include a note in my Assignment 2 submission, but I thought I should probably do a little comparison to be sure that it reliably matches up with the time generated by the original timing function.*

Merge Sort

**Data: “1000000almostsorted.bin”**

Lower bound: 500000

Upper bound: 500200

Sample List:

16383 16383 16383 16383 16383 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16390 16390 16390

**Data: “1000000numbers.bin”**

Lower Bound: 250000

Upper Bound: 250200

Sample List:

8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204

**Data: “10000000numbers.bin”**

Lower Bound: 750000

Upper Bound: 750200

Sample List:

747890 747892 747893 747895 747895 747896 747896 747897 747897 747898 747899 747899 747900 747900 747903 747903 747903 747905 747906 747906 747906 747907 747908 747909 747910 747912 747912 747913 747914 747916 747918 747918 747919 747919 747920 747920 747922 747922 747924 747924 747925 747926 747927 747930 747932 747933 747933 747934 747934 747935 747935 747936 747937 747937 747937 747938 747939 747939 747940 747941 747941 747941 747942 747942 747942 747942 747946 747947 747948 747948 747948 747949 747951 747952 747952 747957 747958 747961 747961 747961 747964 747966 747967 747968 747969 747970 747971 747973 747974 747974 747974 747974 747975 747976 747977 747977 747979 747982 747983 747984 747984 747984 747985 747986 747988 747988 747989 747990 747993 747993 747994 747994 747995 747995 747996 747996 747996 747997 747998 747998 748001 748001 748002 748004 748004 748005 748006 748006 748006 748006 748007 748007 748008 748011 748013 748013 748014 748014 748015 748019 748020 748023 748024 748024 748025 748029 748030 748031 748031 748032 748033 748033 748036 748036 748037 748037 748038 748038 748039 748040 748041 748045 748045 748045 748046 748047 748049 748049 748050 748050 748051 748053 748054 748054 748054 748055 748057 748059 748059 748060 748060 748060 748060 748061 748062 748062 748063 748064 748064 748064 748065 748067 748068 748068 748068 748069 748069 748069 748072 748072

Merge Sort Description:

The best, average, and worst time complexities of merge sort is O(n lg(n)). This time complexity is derived from the fact that there will be *n* operations per set of recursive calls, and the “height” of the resulting recursive tree is *lg(n)*. The latter occurs because each recursive call basically splits the array into two sub arrays, so there will be 2n of these calls. Hence, the time complexity of O(n lg(n)).

Merge sort is considered a divide and conquer algorithm where it breaks an array of unsorted elements recursively into sorted single element sub arrays. Then it combines them by placing the elements in the desired order. That is, given an array of *n* elements, it recursively breaks it down into *n* arrays of 1 element each. While combining, the algorithm compares elements from the two sub arrays individually and, in our case, places the smaller elements first. This happens until either of the sub arrays is exhausted and the remainder of the other sub array is appended to give the combined array.

While merge sort has a relatively fast runtime compared to the other sorts in Assignment 1, it comes at a memory cost. This is because it is not an in-place sorting algorithm, and thus requires the use of another array. In most instances of merge sort, the array is created with each recursive call, but in our case, the array is copied once at the beginning of the algorithm and passed into the initial call to merge sort, subsequent recursive calls to merge sort, and the combine function that sorts and merges the sorted sub arrays.

Quick Sort

**Data: “1000000almostsorted.bin”**

Lower bound: 500000

Upper bound: 500200

Sample List:

16383 16383 16383 16383 16383 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16384 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16385 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16386 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16387 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16388 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16389 16390 16390 16390

**Data: “1000000numbers.bin”**

Lower Bound: 250000

Upper Bound: 250200

Sample List:

8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8198 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8199 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8200 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8201 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8202 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8203 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204 8204

**Data: “10000000numbers.bin”**

Lower Bound: 750000

Upper Bound: 750200

Sample List:

747890 747892 747893 747895 747895 747896 747896 747897 747897 747898 747899 747899 747900 747900 747903 747903 747903 747905 747906 747906 747906 747907 747908 747909 747910 747912 747912 747913 747914 747916 747918 747918 747919 747919 747920 747920 747922 747922 747924 747924 747925 747926 747927 747930 747932 747933 747933 747934 747934 747935 747935 747936 747937 747937 747937 747938 747939 747939 747940 747941 747941 747941 747942 747942 747942 747942 747946 747947 747948 747948 747948 747949 747951 747952 747952 747957 747958 747961 747961 747961 747964 747966 747967 747968 747969 747970 747971 747973 747974 747974 747974 747974 747975 747976 747977 747977 747979 747982 747983 747984 747984 747984 747985 747986 747988 747988 747989 747990 747993 747993 747994 747994 747995 747995 747996 747996 747996 747997 747998 747998 748001 748001 748002 748004 748004 748005 748006 748006 748006 748006 748007 748007 748008 748011 748013 748013 748014 748014 748015 748019 748020 748023 748024 748024 748025 748029 748030 748031 748031 748032 748033 748033 748036 748036 748037 748037 748038 748038 748039 748040 748041 748045 748045 748045 748046 748047 748049 748049 748050 748050 748051 748053 748054 748054 748054 748055 748057 748059 748059 748060 748060 748060 748060 748061 748062 748062 748063 748064 748064 748064 748065 748067 748068 748068 748068 748069 748069 748069 748072 748072

Quick Sort Description:

The best and average time complexities for quick sort is O(n lg(n)), while its worst time complexity is O(n2). The worst-case results from picking repeatedly picking pivots that are either the largest or the smallest of the sub arrays. This will result in all the other elements falling to one side of the pivot (shaped like a North Dakota pine tree), and hence causing the algorithm to make the largest number of comparisons. In the case when the data is random, quick sort does *n* operations with each partition, and each partition divides the array into 2 sub arrays, resulting in *lg(n)* operations. Together, we have a time complexity of O(n lg(n)).

Like merge sort, quick sort is a divide and conquer algorithm where it divides the array based on a pivot value. Also, like merge sort, it relies on making recursive calls to itself to divide and sort the elements. Essentially, quick sort picks a pivot element and places other smaller elements on the left of it and other larger elements on the right of it. Every time it does this, the pivot is considered sorted. The elements on the left of the pivot and the elements to the right of the pivot are now considered to be in two distinct sub arrays, and the recursive call to quick sort is made on each of these. The process repeats until the sub arrays are 1 element long, or when the only element is the pivot. The selection of the pivot significantly affects the runtime of the algorithm as well. The worst case, described above, is caused by the repeated selection of the smallest or largest elements as pivots. The best case happens when the pivots are always the median element of each subarray. One way to mitigate this problem is to randomize an array before running quick sort on it. While some implementations of quick sort select the first or last element as the pivot, the algorithm in this assignment picks the median element because in most real world dataset, the data is more or less near sorted, and if it is not, the median element will just be a random element.

Merge Sort vs. Quick Sort:

While the worst case time complexity of merge sort is O(n lg(n)), the worst case time complexity of quick sort is O(n2). That being said, there are ways to reduce the probability of the worst case of quick sort occurring. Quick sort also features an increasing number of elements (the pivots) that are sorted with each iteration. From the runtimes collected, it appears that merge sort is relatively slower than quick sort when dealing with a smaller set of elements.

|  |  |  |
| --- | --- | --- |
|  | One Million (almost sorted) | One Million (random) |
| Merge Sort | 0.21127s | 0.212959s |
| Quick Sort | 0.173623s | 0.183974s |

*(Quick sort slightly faster with 1000000 elements)*

|  |  |
| --- | --- |
|  | Ten Million (random) |
| Merge Sort | 2.36681s |
| Quick Sort | 2.75589s |

*(merge sort slightly faster with 10000000 elements)*

Besides slight runtime differences, quick sort is an in-place sorting algorithm, which means that unlike merge sort, it does not need to create a copy of the original array. This makes it more memory efficient than merge sort.

Merge Sort & Quick Sort vs. Sorts in Assignment 1

The runtimes of merge sort and quick sort are significantly faster than the runtimes of bubble sort, selection sort, insertion sort, and shell sort mainly because merge sort and quick sort utilizes the divide and conquer technique with recursion. By dealing with half the number of elements each recursive call, they create a “tree” of sorts where the height of that “tree” is what the algorithm iterates over instead of individual elements (in the case of bubble sort, selection sort, insertion sort, and shell sort). Instead of having to iterate through approximately *n* times in the outer loop and approximately *n* times in the inner loop like in the sorts in assignment 1, merge and quick sorts only have to iterate approximately *lg(n)* times in the outer loop (recursion) and approximately *n* times at each “level” of the “tree”.