Analysis of Four Sorting Algorithms

Project Report

Written and prepared by:
Roxanne Lutz
University of New Mexico
May 9, 2025

Introduction

All code, executables, and implementation details can be found not only in the zip file included in the project submission, but also at the following public GitHub repository:

https://github.com/rlutz1/CS361_Project

Preliminaries and Methods

The following section will be introducing any key ideas, characteristics, or implementation details that are relevant to note before diving into the complete analysis and discussion of the sorting algorithms.

General Notes

The process to benchmark these algorithms is as follows:

(1) Randomly generate an array of *n* unsorted integers or doubles for a test case. Each test case is then unique and diverse.
(2) Begin a timer.
(3) Sort the array with the needed algorithm.
(4) End the timer.
(5) Log the time it took to sort this test case with this algorithm to a unique file identified by the algorithm, integer/double type number, and *n*.
(6) Repeat (1) - (5) 200 times for all benchmark cases as to get a diverse set of data.

All cases were run on a 16GB RAM machine under the same conditions (plugged in battery, nothing else running on the machine meanwhile).

The following builtin Java classes have been used as needed as per the allowance of the original project notes:

+ *FileWriter*: basic IO operations; used specifically for writing benchmark times to files within the Log class for ease of data collection.
+ *System*: basic console printing and timing.
+ *Random*: basic random number generation for test cases.
+ *Math*: solely for calculating powers of 2.

All sort algorithms are implemented in their own classes: ThreeWayMergeSort, RandomizedQuickSort, QuadHeapSort, and TimSort. All of them have a global array for the numbers to array that is allocated memory upon call to initialize the array (unsorted). From there, a simple .sort() call from any one of the classes engages their specific algorithm to sort the global array.

No algorithms have been implemented with any specific time complexity optimizations. This choice was made to keep the testing and methods consistent along the “vanilla” implementations of each algorithm.

Quad-HeapSort

Quad-HeapSort (QHS) is implemented in the same fashion as a typical heapsort with a binary max-heap. Below is the code pertinent to this specific algorithm.

We first start by assuming the array is already a quad heap and fix any violation from the first parent. In a binary heap, this would be calculated by $Floor((n - 2) / 2)$, where *n* is the size of the array and assuming zero indexing. Instead, with a quad heap, we calculate the last parent similarly as $Floor((n - 2) / 4)$, the 4 accounting for the ability of 4 children per parent.

Then, we perform the typical heapsort action: swap the top of the heap with the last leaf, and then fix any heap violations from the root down. Repeat this action until we have an array in ascending order.

In the maxHeapify logic, we calculate all children similarly to that of a binary heap, but instead account for the 4 children accordingly: $(4 * i) + \{1, 2, 3, 4\}$. The set *{1, 2, 3, 4}* will yield child 1, child 2, child 3, and child 4 in that order from a given parent index *i* (as seen in maxHeapify below).

A safeguard worth noting on the implementation of this algorithm is, when calculating the 4 children, we do have to do a check on the parent index. We cannot calculate the children if the parent index is equal or larger to 2^29. This will cause an overflow since we are using a 32-bit integer for start and children. For example:

$$int\ child1 = 4 * 2^{29} + 1 = 2^{2} * 2^{29} + 1 = 2^{31} + 1 = -2^{31}\ //\ overflow$$

This would technically catch on the *childX* < *end* clause when finding the maximum and then attempt to access a negative index on the array, throwing an error. Simplest fix is to just not allow any numbers that would cause the problem, such as >= 2^29 for this project. If we were to benchmark larger cases, then we would need to consider how to address this (making it a 64-bit number, sub-problems, etc.), but since we are only going to 2^30, this is out of the scope of the project, however worth noting.

```
public void sort(int size) {

    // assume array is a heap and heapify starting from last parent
    for (int i = (size - 2) / 4; i >= 0; i--) {
        maxHeapify(i, size);
    } // end loop
```

```java
    // move max to the end, fix heap violations from root down to last leaf
    for (int i = size - 1; i >= 0; i--) {
        swap(0, i);
        maxHeapify(0, i);
    } // end loop

} // end method


public void maxHeapify(int start, int end) {
    if (start < (int) Math.pow(2,29)) { // limit to note
        // grab all 4 children
        int child1 = 4 * start + 1;
        int child2 = 4 * start + 2;
        int child3 = 4 * start + 3;
        int child4 = 4 * start + 4;
        int max = start;

        // find the max of the parent and children
        if (child1 < end && toSort[child1] > toSort[max]) {
            max = child1;
        } // end if

        if (child2 < end && toSort[child2] > toSort[max]) {
            max = child2;
        } // end if

        if (child3 < end && toSort[child3] > toSort[max]) {
            max = child3;
        } // end if

        if (child4 < end && toSort[child4] > toSort[max]) {
            max = child4;
        } // end if

        // if we found a new maximum
        if (max != start) {
            swap(start, max); // ensure max is the parent node
            maxHeapify(max, end); // fix the subtree as needed
        } // end if
    } // end if
} // end method

private void swap(int x, int y) {
    int temp = toSort[x];
    toSort[x] = toSort[y];
    toSort[y] = temp;
} // end method
```

Three-Way MergeSort

Three-Way MergeSort (TWMS) is implemented very closely to how the original algorithm is implemented. The following code is all implementation details.

The sort method is the implementation of TWMS in the classic original mergesort fashion, but instead of sorting the first half, second half, we are sorting first, middle, and last thirds.

Then, we must call merge twice instead of once in order to merge the thirds accordingly (first third with middle, then first third + middle with last third).

Due to memory issues, all benchmark data of mergesort were collected using arrays of 16-bit shorts instead of 32-bit integers. Once benchmarking, the 2^30 integer case continually overflowed the JVM heap. Hence, to be able to run automated testing, the arrays for TWMS were downgraded to short[] to combat the memory problem. This issue is specifically due to the merge operation needing a temporary array, as confirmed with a similar problem with TimSort–having the exact same merge operation. More will be discussed on this in the future section.

```
public void sort(int low, int high) {
   if (low >= high) { return; }
   if (high - low == 1) { swapIfNeeded(low, high); return;}

   int third = (high - low) / 3;

   sort (low, low + third);
   sort (low + third + 1, high - third);
   sort (high - third + 1, high);

   merge (low, low + third, low + third + 1, high - third);
   merge (low, high - third, high - third + 1, high);
} // end method

private void merge(int startA, int endA, int startB, int endB) {
   int i = startA, j = startB, k = 0;
   short[] temp = new short[endB - startA + 1];

   // merge the two sorted lists into the temporary array
   while (i <= endA && j <= endB) {
       if (toSort[i] > toSort[j]) {
           temp[k] = toSort[j]; k++; j++;
       } else {
```

```
            temp[k] = toSort[i]; k++; i++;
        } // end if
    } // end loop

    // dump the contents of array 1 if needed
    while (i <= endA) {
        temp[k] = toSort[i]; k++; i++;
    } // end while

    // dump the contents of array 2 if needed
    while (j <= endB) {
        temp[k] = toSort[j]; k++; j++;
    } // end while

    // copy the temporary array's merged contents into the global
    for (i = startA, k = 0; i <= endB; i++, k++) {
        toSort[i] = temp[k];
    } // end loop
} // end method

private void swapIfNeeded(int low, int high) {
    if (toSort[low] > toSort[high])  {
        swap(low, high);
    } // end if
} // end method

private void swap(int x, int y) {
    short temp = toSort[x];
    toSort[x] = toSort[y];
    toSort[y] = temp;
} // end method
```

Randomized QuickSort

Randomized QuickSort (RQS) is implemented almost the exact same as normal quicksort. However, the main difference is in the random pivot selection. There is a single helper method that will generate a random number using Java's Random within the confines of the low and high indices. We then swap the value at that random index with the end of the subarray. Then, the algorithm is quicksort as usual with the end being the pivot value.

```
public void sort(int low, int high) {
    if (low < high) {
```

```
        int pivot = partition(low, high); // partition and return pivot
        sort(low, pivot - 1); // sort first half
        sort(pivot + 1, high); // sort second half
    } // end if
} // end method

private int partition(int start, int end) {
    swap(getRandomPivot(start, end), end); // get a random pivot choice
    int b = start - 1, t = start; // set up bookmark and traveller

    // place all nums smaller than pivot left of b, all things bigger right of b
    while (t < end) {
        if (toSort[t] < toSort[end]) {
            b++;
            swap(t, b);
        } //end if
        t++;
    } // end loop

    // if the bookmark is not at the end of this subarray
    if (b < end) {
        swap(b + 1, end);
    } // end if

    return b + 1; // return the pivot
} // end method

private int getRandomPivot(int low, int high) {
    return random.nextInt(low, high + 1);
} // end method

private void swap(int x, int y) {
    int temp = toSort[x];
    toSort[x] = toSort[y];
    toSort[y] = temp;
} // end method
```

TimSort

TimSort (TS) is implemented the following way:

(1) Use insertion sort on a specified *MIN_RUN* size subarray, sorting *MIN_RUN* sized pieces of the original unsorted array.
(2) Merge the now sorted subarrays with the same merge function as mergesort. Repeat until the entire array has been properly merged.

All baseline benchmarking and test cases were run with a default *MIN_RUN* size of 32, as seen below.

There are no optimizations implemented on TS, such as binary insertion sort, galloping, etc, just due to the desire to be able to more directly compare it with the other algorithms without adding more noise, so to speak.

```
private final static byte MIN_RUN = 32;

public void sort() {

   // use insertion sort on smaller runs
   for (int i = 0; i < toSort.length; i += MIN_RUN) {
       insertionSort(i, Math.min(toSort.length - 1, i + MIN_RUN - 1));
   } // end loop

   // merge the sub runs, increasing the sub run length with each iteration
   for (int j = MIN_RUN; j < toSort.length; j *= 2) {
       for (int i = 0; i < toSort.length; i += 2 * j) {
           merge(i, Math.min(toSort.length - 1, i + j - 1), i + j,
Math.min(toSort.length - 1, i + (2 * j) - 1));
       } // end loop
   } // end loop

} // end method

private void insertionSort(int low, int high) {
   int j; short temp;
   for (int i = low; i < high; i++) { // for every element of the list

       if (toSort[i + 1] < toSort[i]) { // if we find a smaller element ahead
           j = i; // bookmark
           temp = toSort[i + 1]; // grab the element to "insert"
           while (j >= low && toSort[j] > temp) { // until we hit end or
something smaller
               toSort[j + 1] = toSort[j]; // make space
               j--;
           } // end loop
           toSort[j + 1] = temp; // insert element in correct spot
       } // end if
   } // end loop

} // end method


private void merge(int startA, int endA, int startB, int endB) {
   int i = startA, j = startB, k = 0;
```

```java
    short[] temp = new short[endB - startA + 1];


    while (i <= endA && j <= endB) {
        if (toSort[i] > toSort[j]) {
            temp[k] = toSort[j]; k++; j++;
        } else {
            temp[k] = toSort[i]; k++; i++;
        } // end if
    } // end loop

    while (i <= endA) {
        temp[k] = toSort[i]; k++; i++;
    } // end while

    while (j <= endB) {
        temp[k] = toSort[j]; k++; j++;
    } // end while

    for (i = startA, k = 0; i <= endB; i++, k++) {
        toSort[i] = temp[k];
    } // end loop

} // end method
```

Performance Analysis and Discussion

The following section will include an in-depth analysis and discussion of each of the 4 sorting algorithms implemented: Quad-HeapSort (QHS), Three-Way MergeSort (TWMS), Randomized QuickSort (RQS), and TimSort (TS). Following the analysis of the individual algorithms will be an analysis of how the algorithms compare in terms of which is, in practice, the fastest of four.

Quad-Heapsort

Implementation and Coding Complexity

Asymptotic Analysis

Experimental Runtime Analysis and Benchmarking

3-Way Mergesort

Implementation and Coding Complexity

Asymptotic Analysis

Experimental Runtime Analysis and Benchmarking

Randomized Quicksort

Implementation and Coding Complexity

Asymptotic Analysis

Experimental Runtime Analysis and Benchmarking

Timsort

Implementation and Coding Complexity

Asymptotic Analysis

Experimental Runtime Analysis and Benchmarking

Comparisons of All

Comparing Average Experimental Runtimes

Conclusions

Conclusion

Bibliography

Contributions

Acknowledgments