# CS430

# HW1

# Team “Yamaha Piano”

Malcolm Machesky and Adrian Kirchner

A screenshot of a cell phone

Description automatically generated

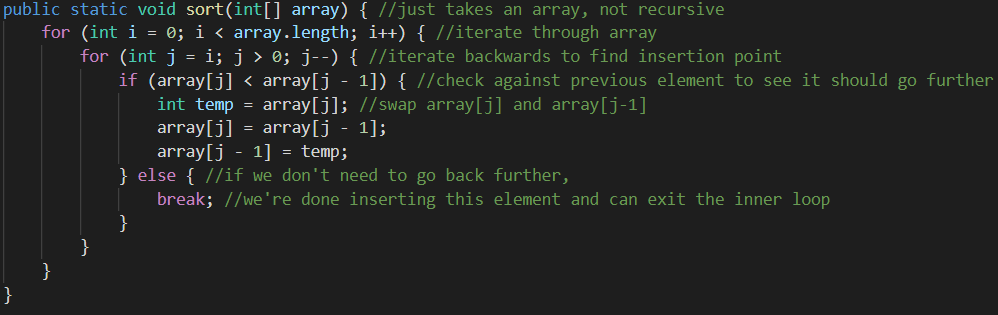
# Project Management

Table presented by name of participant and by day

|  |  |  |
| --- | --- | --- |
|  | Wednesday | Thursday |
| Malcolm Machesky | * Worked on GUI in (Gui.java) (2 Hr) * Worked on instruction ppt (30 min) * Helped combine GUI and sorting algorithms (1.5 Hr) * Total Hours: 4 | * Finished GUI (30 min) * Wrote this Project management document (30 min) * Total hours: 1 |
| Adrian Kirchner | * Worked on sorting algorithms in (Lab1.java, MergeSort.java and insertionSort.java) (2.5 Hr) * Helped combine GUI and sorting algorithms (1.5 Hr) * Total Hours: 4 | * Finished sorting algorithms (30 min) * Did algorithm analysis (30 min) * Total hours: 1 |

# Sorting Algorithms Analysis

## Insertion Sort



Most operations in this algorithm are constant, with the two main exceptions being the two for loops.

The inner loop has worse performance as the size of the array (n) gets larger, and iterates once over the entire array except one element when i is largest. Therefore, it has a runtime complexity of O(n) (in isolation).

The outer loop iterates over the entire array once and runs the inner loop (which has a runtime complexity O(n)) each time, resulting in a performance of O(n2).

Line by line breakdown below:

public static void sort(int[] array) { // O(n­2)

    for (int i = 0; i < array.length; i++) { // O(n2)

        for (int j = i; j > 0; j--) { // O(n)

            if (array[j] < array[j - 1]) { // O(1)

                int temp = array[j]; // O(1)

                array[j] = array[j - 1]; // O(1)

                array[j - 1] = temp; // O(1)

            } else { // O(1)

                break; // O(1)

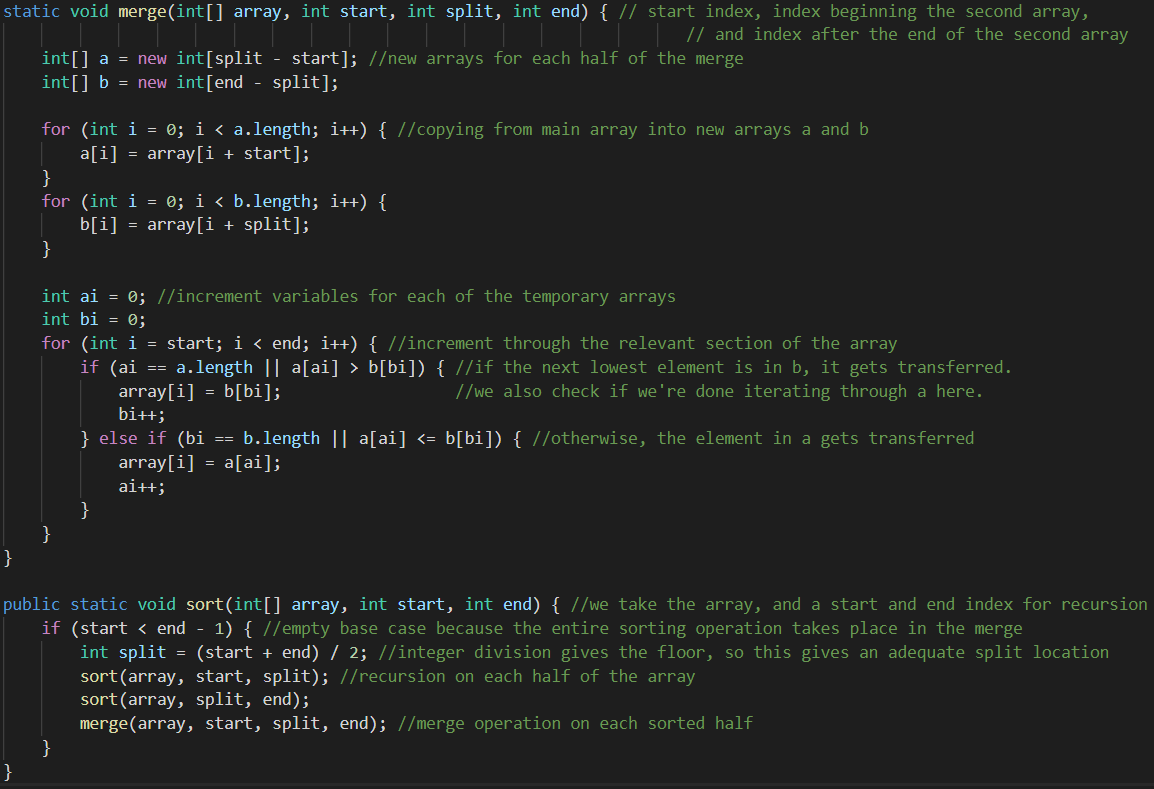
            }

        }

    }

}

## Merge Sort



Most operations in merge() are constant, with the exception of the loop near the end, which iterates through the array once with variable bounds, with a runtime complexity of O(n).

The sort() function is a little more complicated, since it is recursive. It contains a call to merge, which has O(n) complexity, but it also has two calls to itself, so we have to figure out how many times it can loop in order to determine the overall complexity. Since the array splits in half each time, and the base case is an array size of 1 or less, it can only split in half log2n times before all recursive calls reach the base case. Therefore, the loop runs log2n times, running a O(n) complexity function each time (merge()), so the overall sort() function must have a runtime complexity of O(n log n).

Line by line breakdown on following page.

static void merge(int[] array, int start, int split, int end) { // O(n)

    int[] a = new int[split - start]; // O(1)

    int[] b = new int[end - split]; // O(1)

    for (int i = 0; i < a.length; i++) { // O(n)

        a[i] = array[i + start]; // O(1)

    }

    for (int i = 0; i < b.length; i++) { // O(n)

        b[i] = array[i + split]; // O(1)

    }

    int ai = 0; // O(1)

    int bi = 0; // O(1)

    for (int i = start; i < end; i++) { // O(n)

        if (ai == a.length || a[ai] > b[bi]) { // O(1)

            array[i] = b[bi]; // O(1)

            bi++; // O(1)

      } else if (bi == b.length || a[ai] <= b[bi]) { // O(1)

            array[i] = a[ai]; // O(1)

            ai++; // O(1)

        }

    }

}

public static void sort(int[] array, int start, int end) { // O(n log n)

    if (start < end - 1) { // O(1)

        int split = (start + end) / 2; // O(1)

        sort(array, start, split); // O(n log n)

        sort(array, split, end); // O(n log n)

        merge(array, start, split, end); // O(n)

    }

}