# CS430

# HW1

# Team “Yamaha Piano”

Malcolm Machesky and Adrian Kirchner

A screenshot of a cell phone

Description automatically generated

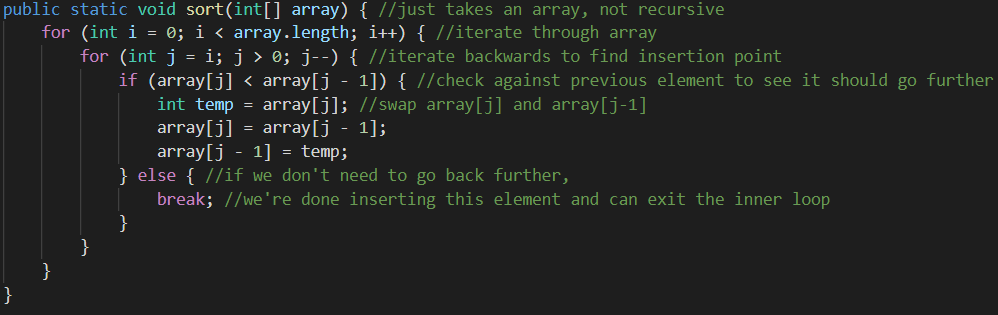
# Project Management

Table presented by name of participant and by day

|  |  |
| --- | --- |
|  | Wednesday |
| Malcolm Machesky | * Modified (Gui.java) (5 min) * Worked on instruction ppt and Project management(10 min) * Helped combine GUI and sorting algorithms (20 min) * Worked on (HeapSort.java) (2hr) * Worked on analysis (1 50 min) * Total Hours: 4 25 min |
| Adrian Kirchner | * Worked on sorting algorithms in (QuickSort.java) (2 Hr) * Helped combine GUI and sorting algorithms (20 min) * Modified (Gui.java) (5 min) * Worked on analysis (2 hr) * Total Hours: 4 25 min |

# 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)

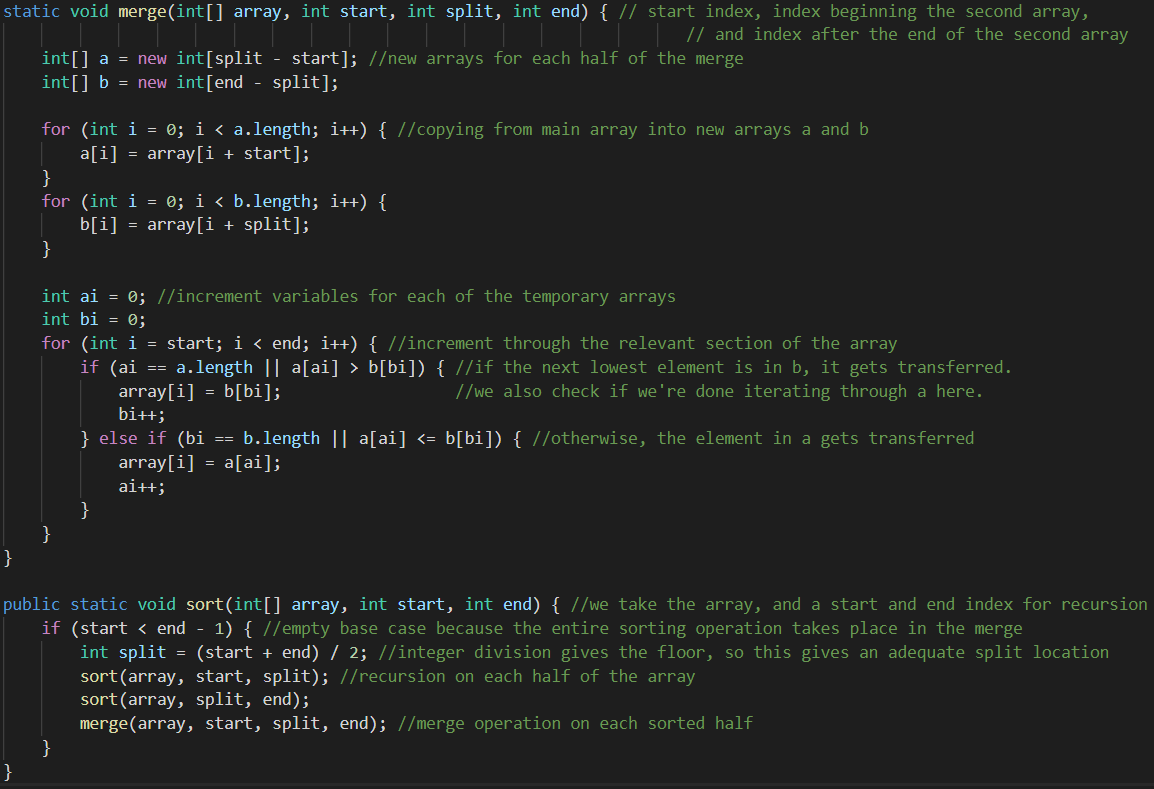
            }

        }

    }

}

## 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.

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

for (int i = array.length / 2 - 1; i >= 0; i--) { // O(n log(n))

heapify(array, array.length, i); // O(log(n))

}

for (int i = array.length - 1; i >= 0; i--) { // O(n log(n))

array = swap(array, 0, i); // O(1)

// re heapify

heapify(array, i, 0); // O(log(n))

}

}

static void heapify(int[] array, int s, int i) {

int root = i; // O(1)

int l = 2 \* i + 1; // O(1)

int r = 2 \* i + 2; // O(1)

if (l < s && array[l] > array[root]) { // O(1)

root = l; // O(1)

}

if (r < s && array[r] > array[root]) { // O(1)

root = r; // O(1)

}

if (root != i) { // O(log(n))

array = swap(array, i, root); // O(1)

heapify(array, s, root); // O(log(n))

}

}

public static int[] swap(int[] array, int i, int j) {// O(1)

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

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

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

return array; // O(1)

}

Heapify mathematical analysis

Since ,

In the sort function there are 2 for loops which are each O(n log(n)) because in each of the for loops there is a call to the recursive function heapify which has a runtime complexity of (log(n)) as shown above. Making the entire function O(n logn).

|  |  |  |
| --- | --- | --- |
| n | quick(ms) | heap(ms) |
| 1000 | 1 | 1 |
| 10000 | 1 | 2 |
| 100000 | 11 | 17 |
| 1000000 | 95 | 169 |
| 10000000 | 1112 | 2954 |
| 100000000 | 12847 | 42354 |

Both Quick sort and heap sort both have the run time complexity of O(n log(n)). As you can see they have similar growth as you make the array larger.