# CS430

# HW3

# Team “Yamaha Piano”

Malcolm Machesky and Adrian Kirchner

A screenshot of a social media post

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 (1hr, 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 |

## Algorithm comparison and analysis

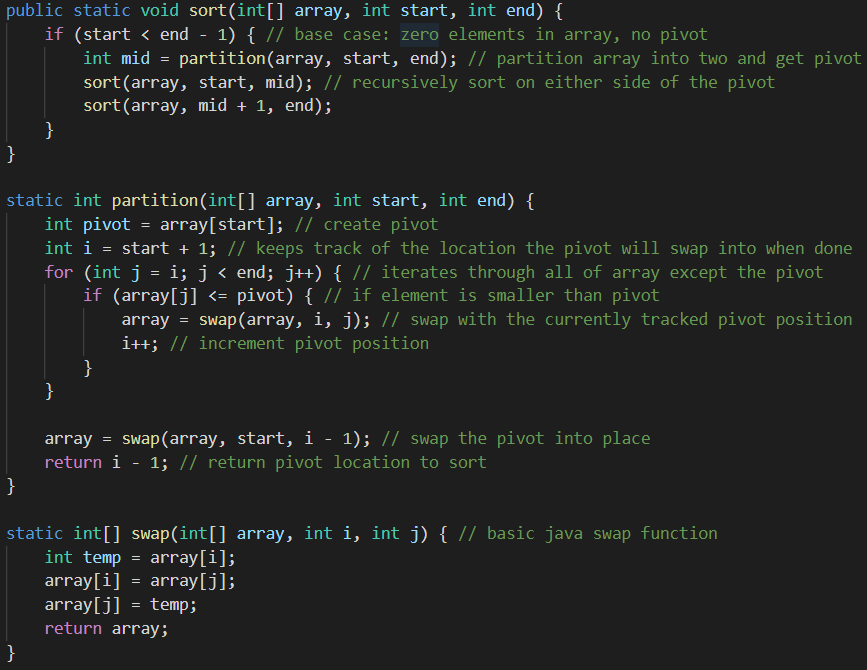
Below is a table of results from running our program on various sizes of arrays.

|  |  |  |
| --- | --- | --- |
| 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.

# Sorting Algorithms Analysis

## QuickSort Analysis:



The partition function runs in O(n) because it has to iterate through the entire length of the array.

In order to get the runtime complexity of sort we have to solve the following recurrence relation:

Since , according to Master’s Theorem.

The worst-case scenario for sort is that partition always returns the first element of the array as the pivot because that forces sort to be called recursively times. Therefore, the worst-case time complexity of sort is O(n2)

Line by line breakdown below:

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

    if (start < end - 1) { // O(n2)

        int mid = partition(array, start, end); // O(n)

        sort(array, start, mid); // O(n2)

        sort(array, mid + 1, end); // O(n2)

    }

}

static int partition(int[] array, int start, int end) {

    int pivot = array[start]; // O(1)

    int i = start + 1; // O(1)

    for (int j = i; j < end; j++) { // O(1)

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

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

            i++; // O(1)

        }

    }

     array = swap(array, start, i - 1); // O(1)

    return i - 1; // O(1)

}

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)

}

## HeapSort Analysis:

## 

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).

Line by like breakdown below:

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)

}