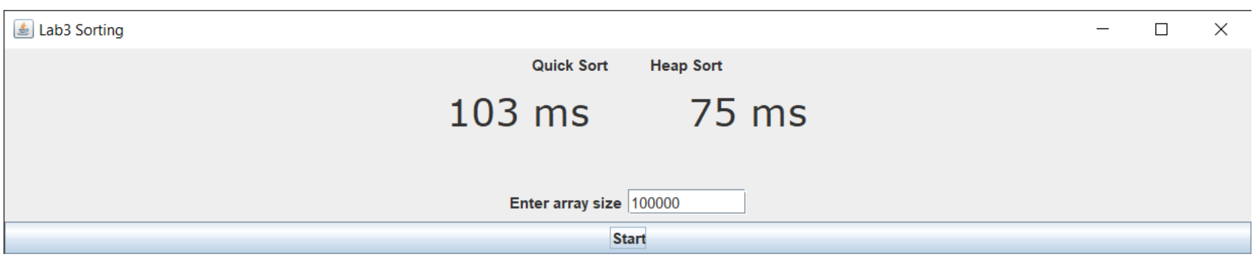
Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

CS430

HW3

Team “Yamaha Piano”

Malcolm Machesky and Adrian Kirchner



Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

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.

Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

Sorting Algorithms Analysis

QuickSort Analysis:

```java
public static void sort(int[] array, int start, int end) {
    if (start < end - 1) { // base case: zero elements in array, no pivot
        int mid = partition(array, start, end); // partition array into two and get pivot
        sort(array, start, mid); // recursively sort on either side of the pivot
        sort(array, mid + 1, end);
    }
}

static int partition(int[] array, int start, int end) {
    int pivot = array[start]; // create pivot
    int i = start + 1; // keeps track of the location the pivot will swap into when done
    for (int j = i; j < end; j++) { // iterates through all of array except the pivot
        if (array[j] <= pivot) { // if element is smaller than pivot
            array = swap(array, i, j); // swap with the currently tracked pivot position
            i++; // increment pivot position
        }
    }

    array = swap(array, start, i - 1); // swap the pivot into place
    return i - 1; // return pivot location to sort
}

static int[] swap(int[] array, int i, int j) { // basic java swap function
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    return array;
}
```

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:

$$T(n) = 2T(n/2) + n$$
$$a = 2, b = 2, f(n) = n$$
$$n^{log_2 2} = n$$

Since $f(n) = \Theta(n), T(n) = nlog(n)$ 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 $n - 1$ times. Therefore, the worst-case time complexity of `sort` is O(n^2)

Line by line breakdown below:

Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

```java
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)
}
```

Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

HeapSort Analysis:

```
public static void sort(int[] array) {
    // build the heap by going through the heap and heapifing the different
    // subtree's
    for (int i = array.length / 2 - 1; i >= 0; i--) {
        heapify(array, array.length, i);
    }
    // go through the array and move the root node to the top of the heap and
    // replace it with the value at the end of the array
    for (int i = array.length - 1; i >= 0; i--) {
        array = swap(array, 0, i);
        // re heapify
        heapify(array, i, 0);
    }
}

// int i is a index in the array int s is the size of the heap
static void heapify(int[] array, int s, int i) {
    // the root of the tree or of the subtree that is getting heapified
    int root = i;
    // find its left and right node
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    // if the left node is greater then the root node set the root node to the left
    // node
    if (l < s && array[l] > array[root]) {
        root = l;
    }

    // if the right node is greater then the root node set the root node to the
    // right node
    if (r < s && array[r] > array[root]) {
        root = r;
    }
    // Swap the nodes then reheapify with the new root node
    if (root != i) {
        array = swap(array, i, root);
        // re heapify with the new root node
        heapify(array, s, root);
    }
}

public static int[] swap(int[] array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    return array;
}
```

Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

Heapify mathematical analysis:

$$T(n) = T\left(\frac{n}{2}\right) + C$$
$$F(n) = C$$
$$A = 1, B = 2$$
$$n^{\log_2 1} = 1$$

Since $f(n) = \Theta(1)$, $T(n) = \Theta(1 * \log(n)) = \Theta(\log(n))$.

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 Θ(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))
    }
}
```

Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

```
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)
}
```