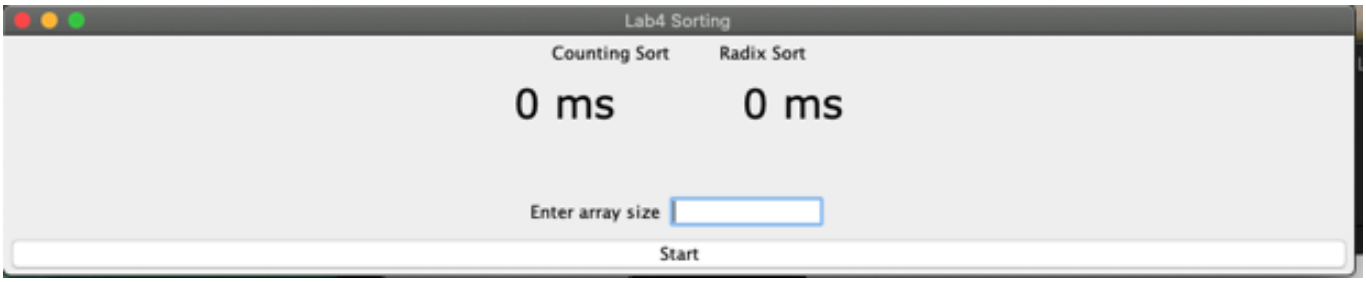
Adrian Kirchner: A20425060, Seat 34
Malcolm Machesky: A20414760, Seat 45

CS430

HW4

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 (RadixSort.java) (2hr) • Worked on analysis (1hr, 50 min) • Total Hours: 4 25 min
Adrian Kirchner	• Worked on sorting algorithms in (CountingSort.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. We used a maximum value of 10000000 so that the differences in performance for counting sort would be apparent. We also ran the same dataset through the previous assignments and included the results.

n	Counting(ms)	Radix(ms)	Quick(ms)	Heap(ms)	Insertion(ms)	Merge(ms)
1000	13	1	1	1	4	1
10000	13	2	1	1	55	1
100000	20	11	8	11	1574	10
1000000	152	103	100	133	162575	107
10000000	1279	988	1088	2420	N/A*	N/A*
100000000	13947	8498	11969	37957	N/A*	N/A*

*Homework 1 took too long to run with the larger values of n, and so these have been excluded.

Radix sort is the fastest sort so far, and counting seems to be somewhere in between heap and quick sort with this range of values.

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

Sorting Algorithms Analysis

CountingSort Analysis:

```java
public static void sort(int[] array, int max) {
    int[] result = new int[array.length]; // initialize result array

    int[] count = new int[max + 1]; // initialize counting array (holds the number of instances of each number)
    for (int i = 0; i <= max; i++) { // clear array to zero
        count[i] = 0;
    }

    for (int j = 0; j < array.length; j++) { // for each element in the initial array
        count[array[j]]++; // increment the corresponding element of the count array
    }

    for (int i = 1; i <= max; i++) { // iterates through the count array and adds all the previous values
        count[i] += count[i - 1]; // the count array contains a record of how many values <= i
    }

    for (int j = array.length - 1; j >= 0; j--) { // for each element in the initial array
        result[count[array[j]] - 1] = array[j]; // insert it in the correct location by checking the count array for
                                                // the number of values below it
        count[array[j]]--; // decrement the value in the count array to handle duplicates
    }

    for (int j = 0; j < array.length; j++) { // copy result array back to input array
        array[j] = result[j];
    }
}
```

Counting sort consists of several loops that iterate through arrays of size n or size max. Combining these gives O(n + m) where n is the number of elements in the array and m is the maximum value of the array.

Line by line breakdown below:

```java
public static void sort(int[] array, int max) { //O(m + n)
    int[] result = new int[array.length]; //O(1)

    int[] count = new int[max + 1]; //O(1)
    for (int i = 0; i <= max; i++) { //O(m)
        count[i] = 0; //O(1)
    }

    for (int j = 0; j < array.length; j++) { //O(n)
        count[array[j]]++; //O(1)
    }

    for (int i = 1; i <= max; i++) { //O(m)
        count[i] += count[i - 1]; //O(1)
    }
```

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

```
    for (int j = array.length - 1; j >= 0; j--) { //O(m)
        result[count[array[j]] - 1] = array[j]; //O(1)
        count[array[j]]--; //O(1)
    }

    for (int j = 0; j < array.length; j++) { //O(n)
        array[j] = result[j]; //O(1)
    }
}
```

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

RadixSort Analysis:

```java
public static void sort(int array[]) {
    int max = getMax(array);
    // Go through each digit of the numbers in the array to sort
    for (int exp = 1; max / exp > 0; exp *= 10) {
        // use a modified version of count sort to sort the array by digit
        countSort(array, exp);
    }
}

// Function to get the maximum value from the array
private static int getMax(int array[]) {
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}

// Same as regular counting sort but uses a exponent value to split it up by a
// individual digit
public static void countSort(int[] array, int exp) {
    int[] result = new int[array.length]; // initialize result array

    int[] count = new int[10]; // initialize counting array (holds the number of instances of each number)
    for (int i = 0; i < 10; i++) { // clear array to zero
        count[i] = 0;
    }

    for (int j = 0; j < array.length; j++) { // for each element in the initial array
        count[(array[j] / exp) % 10]++; // increment the corresponding element of the count array
    }

    for (int i = 1; i < 10; i++) { // iterates through the count array and adds all the previous values
        count[i] += count[i - 1]; // the count array contains a record of how many values <= i
    }

    for (int j = array.length - 1; j >= 0; j--) { // for each element in the initial array
        result[count[(array[j] / exp) % 10] - 1] = array[j]; // insert it in the correct location by checking the
                                                             // count array for the number of values below it
        count[(array[j] / exp) % 10]--; // decrement the value in the count array to handle duplicates
    }

    for (int j = 0; j < array.length; j++) { // copy result array back to input array
        array[j] = result[j];
    }
}
```

The counting sort used here consists of several loops that iterate through arrays of size n or size max. Combining these gives O(n + 10) which simplifies to O(n) where n is the number of elements in the array and 10 is the maximum size of the array as there are only 9 digits.

The counting sort is run once for each digit of the max value in the array, so log(max) times. This gives an overall time complexity of O(n log(m)) where n is the number of elements in the array and m is the maximum value.

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

Line by line breakdown below:

```
public static void sort(int array[]) { //O(n log(m))
    int max = getMax(array); //O(n)
    for (int exp = 1; max / exp > 0; exp *= 10) { //O(n log(m))
         countSort(array, exp); //O(n)
    }
}

    // Function to get the maximum value from the array
private static int getMax(int array[]) { //O(n)
    int max = array[0]; //O(1)
    for (int i = 1; i < array.length; i++) { //O(n)
        if (array[i] > max) { //O(1)
            max = array[i]; //O(1)
        }
    }
    return max; //O(1)
}

public static void countSort(int[] array, int exp) { //O(n)
    int[] result = new int[array.length]; //O(1)
    int[] count = new int[10];  //O(1)
    for (int i = 0; i < 10; i++) { //O(1)
        count[i] = 0; //O(1)
    }

    for (int j = 0; j < array.length; j++) {  //O(n)
        count[(array[j] / exp) % 10]++; //O(1)
    }

    for (int i = 1; i < 10; i++) { //O(1)
        count[i] += count[i - 1]; //O(1)
    }

    for (int j = array.length - 1; j >= 0; j--) { //O(1)
        result[count[(array[j] / exp) % 10] - 1] = array[j]; //O(1)
        count[(array[j] / exp) % 10]--;  //O(1)
    }

    for (int j = 0; j < array.length; j++) { //O(n)
        array[j] = result[j]; //O(1)
    }
}
```