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

Sorting Algorithms Analysis

Insertion Sort

```java
public static void sort(int[] array) { //just takes an array, not recursive
    for (int i = 0; i < array.length; i++) { //iterate through array
        for (int j = i; j > 0; j--) { //iterate backwards to find insertion point
            if (array[j] < array[j - 1]) { //check against previous element to see it should go further
                int temp = array[j]; //swap array[j] and array[j-1]
                array[j] = array[j - 1];
                array[j - 1] = temp;
            } else { //if we don't need to go back further,
                break; //we're done inserting this element and can exit the inner loop
            }
        }
    }
}
```

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 $O(n^2)$.

Line by line breakdown below:

```java
public static void sort(int[] array) { // O(n^2)
    for (int i = 0; i < array.length; i++) { // O(n^2)
        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)
            }
        }
    }
}
```

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

Merge Sort

```java
static void merge(int[] array, int start, int split, int end) { // start index, index beginning the second array,
                                                                // and index after the end of the second array
    int[] a = new int[split - start]; //new arrays for each half of the merge
    int[] b = new int[end - split];

    for (int i = 0; i < a.length; i++) { //copying from main array into new arrays a and b
        a[i] = array[i + start];
    }
    for (int i = 0; i < b.length; i++) {
        b[i] = array[i + split];
    }

    int ai = 0; //increment variables for each of the temporary arrays
    int bi = 0;
    for (int i = start; i < end; i++) { //increment through the relevant section of the array
        if (ai == a.length || a[ai] > b[bi]) { //if the next lowest element is in b, it gets transferred.
            array[i] = b[bi];                  //we also check if we're done iterating through a here.
            bi++;
        } else if (bi == b.length || a[ai] <= b[bi]) { //otherwise, the element in a gets transferred
            array[i] = a[ai];
            ai++;
        }
    }
}

public static void sort(int[] array, int start, int end) { //we take the array, and a start and end index for recursion
    if (start < end - 1) { //empty base case because the entire sorting operation takes place in the merge
        int split = (start + end) / 2; //integer division gives the floor, so this gives an adequate split location
        sort(array, start, split); //recursion on each half of the array
        sort(array, split, end);
        merge(array, start, split, end); //merge operation on each sorted half
    }
}
```

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 $\log_2 n$ times before all recursive calls reach the base case. Therefore, the loop runs $\log_2 n$ 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.

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

```
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)
    }
}
```