Jonathan Tsai

Programming Methodology II

Midterm A

Problem 1

1. I chose to implement merge sort because it is a stable sorting algorithm that runs O(NlogN) in the best and worst case. A stable sorting algorithm sorts identical elements in the same order that they appear in the input: merge sort is stable because if two elements are identical, we insert from the left sub-array, and thus order is preserved. This is seen in the merge() function (code highlighted):

**public** **static** **int**[] merge(**int**[] a, **int**[] b) {

**int**[] c = **new** **int**[a.length + b.length];

**int** i = 0;

**int** j = 0;

**for** (**int** k = 0; k < c.length; k++) {

**if** (i >= a.length) {

c[k] = b[j++];

} **else** **if** (j >= b.length) {

c[k] = a[i++];

} **else** **if** (a[i] <= b[j]) {

c[k] = a[i++];

} **else** {

c[k] = b[j++];

}

}

**return** c;

}

2. The best-case time complexity of merge sort is O(NlogN), where N is the size of the input array. This is because merge sort is a divide and conquer algorithm: we are continually dividing the array in half, which contributes logN to our run time, and because of our merge step (where we combine our arrays back together), we multiply that contribution by N. Therefore, out overall best-case time complexity is O(NlogN).

/\* a is the array to be sorted. size is the size of the array. \*/

**public** **static** **int**[] ExamSort(**int**[] a, **int** size) {

// array of size 1 is already sorted

**if** (size <= 1) {

**return** a;

}

// split the input array in half

**int**[] left = **new** **int**[size / 2];

**int**[] right = **new** **int**[size - (size / 2)];

// populate the arrays

**for** (**int** i = 0; i < left.length; i++) {

left[i] = a[i];

}

**for** (**int** i = 0; i < right.length; i++) {

right[i] = a[i + left.length];

}

**return** *merge*(*ExamSort*(left, left.length), *ExamSort*(right, right.length));

}

**public** **static** **int**[] merge(**int**[] a, **int**[] b) {

**int**[] c = **new** **int**[a.length + b.length];

**int** i = 0;

**int** j = 0;

**for** (**int** k = 0; k < c.length; k++) {

**if** (i >= a.length) {

c[k] = b[j++];

} **else** **if** (j >= b.length) {

c[k] = a[i++];

} **else** **if** (a[i] <= b[j]) {

c[k] = a[i++];

} **else** {

c[k] = b[j++];

}

}

**return** c;

}

Note that the highlighted code recursively calls the ExamSort() function, and this is where our logN contribution comes from. In the merge() function, we can see that our highlighted for loop depends on the size of the input array N.

3. The worst-case time complexity of merge sort is the same as the best-case time complexity: O(NlogN). This is because no matter what the order of the elements in our input array is, we are still going to continually divide our array in half before merging it back together. Thus, our analysis is the same as in the previous question. The following highlighted code supports our analysis:

**return** *merge*(*ExamSort*(left, left.length), *ExamSort*(right, right.length));

**for** (**int** k = 0; k < c.length; k++) {