MT2 Official Solutions

Question 2: Runtime

Question 2.1

The outer for loop is doubling the `i` value every time (the update is `i+i`) and the middle for loop runs from 0 to `i`, meaning these two for loops in combination run n times. This is from the summation 1 + 2 + 4 + 8 + 16 + …. = n.
The inner for loop is running exactly n times. The function f0(n) runs in constant time, because the for loop executes 200 times no matter the input. Thus, we have our inner loop running n times a total of n times (from the outer two for loops).
The final answer is Theta(n^2), meaning that any omega bound up to and including n^2 is valid, and any big O bound equal to or greater than n^2 is valid.

Question 2.2

Factorial ends with two methods calls:

1. sum(n) this runs in n time due to creating n-1 recursive calls.
2. A recursive fall to factorial on n-1

This means we will be performing n operations, then n-1 operations, then n-2 operations… then 2 operations, then 1. This sum is theta n^2.

Question 2.3

The outer for loop runs in n time.
We then have two things to deal with in sequence:

1. The call to check. This runs from 0 to n. Thus, this specific call will always be bounded by n.
2. Our for loop happens when `flag` is false. The method check will return false for every single element except the maximum. This means we will be executing the inner for loop n-1 times. It runs in n time.

Thus, overall we have n*(n + n) = n^2 (a theta bound).
Any omega up to and including n^2 is valid, and any big O greater than or equal to n^2 is valid.

Question 3: Mechanical Trees

Question 3.1

22 14 57 2 21 66 -5 10 60 72 -3

Question 3.2

W H B C Y A U T J X R Z

Question 3.3

1. The node promoted is always the middle element. The root node is always black. Because it is fully stuffed, both its children will be red.
2. The left child of the root is the leftmost element in the root node of the 234 Tree, which is four. Again, this is red because it was initially in the root node of the corresponding tree.
3. Because we give priority to left leaning red edges and because the child of four must be black (why? We can't have two red nodes in a row) the child of four must be 3, which is black.
4. The node 32 is the same idea conceptually as the node four. It is red, so its child must be black. The only option is that its child is 72.

Question 3.4

We rotate left on the root to promote a new root. We then rotate left again. The tree is now balanced, and we are done.

Question 4

For each part below, you must fully explain the benefits of two distinct features of the data structure to get full credit.
Solutions must directly tie benefit to ADT and clarify why it is important to the algorithm. Benefits cannot be negations (e.g. "this ADT is superior to maps because maps…." were not counted). Most partial credit solutions were simply too vague.

Question 4.1

To get full credit on this part, you had to fully explain at least one distinct feature that was a benefit.
Accepted answers: List, Stack, Queue

- For list (accepted benefits):
    - Maintaining an ordering is important. (Has to specify when/how you sort)
        - The same as saying sortable, Collections.sort, etc. Not different
        - Can't just say sorted list, order collection. Has to say they will "sort the list."
    - Runtime once sorted is O(logN) for list. Saying .get() runtime is constant is NOT a valid reason unless justified (for example by creating an additional array to store what index each child is at)
- For queue and stack (accepted benefits):
    - Maintain ordering. (Has to be sorted before adding. Can't do Collections.sort)

    - PQ
- Map (accepted benefit)
    - Look up a student is fast.
    - Getting sorted stuff is a case that needs to be taken care of separately.
- Set (accepted benefits)
    - LLRB
        - Maintains ordering
        - Super efficient. All operations are O(logN)
    - Hashsets/HashMaps have more cases:
        - Look up a student is fast.
        - To get back sorted alphabetically, you need to sort on keySet or values.
    - Trieset (Absolute megamind, Jimmy Neutron)
        - Trie set that uses DFS preorder. O(NM) or O(E) where N is number of strings, M is the length of string and E is the number of edges. *Have to figure out a way to fix children's ordering. Like an Array or something.
        - Maintains ordering.
        - Getting information takes O(M) where M is the length of the string.

Question 4.2

Example answer: To implement the catalogue, I would rely on the Map ADT. I would use a key value pair of Bug Object -> count. I can store all the required information about a new bug when I create the initial Bug object, and I can store them in a map for fast lookup. Speed is of the essence when your sample can hop away! This is much more memory efficient than storing count Bug Objects (as a Bug object is assumed to hold more data than a single integer).

Question 4.3

Example answer: I would use a Map to store this information. The key value pairing we would use is String (delimiter) -> count. When iterating over a string, when encountering a special character I would increment the count associated in the map. At the end, the string is valid if all counts are even (everything had a matching closing delimiter). The benefits are speed (contains searches and incrementing a value in a Map is fast) and ease of implementation of algorithms. Other data structure choices would lead to a more complex algorithm.
Solutions that did not permit interleaving of special characters (e.g. _some*more_words*) were given full credit but denoted “stack based solution” in rubric.

Question 5

Question 5.1

Acceptable answers included:

```
public static int sumItUpStream(List<Integer> lst) {
    return lst.stream().map(x -> { if (x % 2 == 0) {
                                       return x;
                                   } else {
                                       return x * -1;
                                   }
                                }).reduce(0,(x,y) -> x + y);
}
```

OR

```
public static int sumItUpStream(List<Integer> lst) {
    return lst.stream()
              .map(x ->  x % 2 == 0 ? x : x * -1)
              .reduce(0,(x,y) -> x + y);
}
```

OR

```
public static int sumItUpStream(List<Integer> lst) {
    return lst.stream()
              .map(x -> x * (-2 * (x % 2) * (x % 2) + 1))
              .reduce(0,(x,y) -> x + y);
}
```

Note could have also done: `.reduce(0, Integer::sum)` instead of `.reduce(0,(x,y) -> x + y)` for all of the above answers.

Question 5.2

```
public static List<Double> findScoresByTA(List<Student> lst, String ta) {
    return helper(lst, Student::getTa, ta, Student::getTotalScore);
}

public static <R,S,T> List<S> helper(List<T> lst, Function<T,R> propGetter,
                                     R propToMatch, Function<T,S> mapper) {
    return lst.stream()
              .filter(x -> propGetter.apply(x).equals(propToMatch))
              .map(mapper)
              .collect(Collectors.toList());
}
```