CS314 Fall 2023 Exam 2 Solution and Grading Criteria.

Grading acronyms:
AIOBE - Array Index out of Bounds Exception may occur.
BOD - Benefit Of the Doubt. Not certain code works, but, can't prove otherwise.
Gacky or Gack - Code very hard to understand even though it works. (Solution is not elegant. Lack of Zen.)
LE - Logic Error in code.
MCE - Major Conceptual Error. Answer is way off base, question not understood based on answer provided.
NAP - No Answer Provided. No answer given on test.
NN - Not Necessary. Code is unneeded. Generally, no points off.
NPE - Null Pointer Exception may occur.
OBOE - Off By One error. Calculation is off by one.
RTQ - Read The question. Violated restrictions or made incorrect assumption.
EFF - Efficiency. Order is worse than expected or unnecessary computations done.

1. Answer as shown or -2 unless question allows partial credit.
First use of quotes in output is wrong, then error carried forward.
No points off for minor differences in spacing, capitalization, commas, and braces.
Text in parenthesis not required. It is simply grading guidance and / or a brief explanation for answer.

A. 124!842
B. Runtime error (StackOverflow error occurs)
C. 66
D. 16 seconds (Method is O(2^N))
E. 0.002 seconds (Method is O(N))
F. {1=T, 3=Z, 5=C}
G. {0=3, 2=1, 4=0}
H. 0.2 seconds (HashMap put is O(1))
I. 0.44 seconds (TreeMap put is O(logN))
J. 4 (No partial credit)
K. 10
L. O(N)
M. O(N^2)
N. Quicksort (Average case O(NlogN) worst case O(N^2))
O. 1 9 7 3 5
P. 2000 (full credit if answer in range [1800, 2200])
Q. 100,000,000
R. 8 seconds (based on the implementation dequeue is O(N))
S. It would not be reasonable because there may be duplicate values in the map but sets do not contains duplicates. (The client may WANT all the values.)
T. O(N)
U. O(N^2)
V. 2
W. 8
X. Y T P O H N
Y. T Y O N H P

```
public static Object[] getMap(String[] results) {
        TreeMap<String, int[]> map = new TreeMap<>();
        for (int i = 0; i < results.length; i += 2) {
            String name = results[i];
            String gameResult = results[i + 1];
            int[] record = map.get(name);
            if (record == null) {
                // first time seen this team.
                record = new int[2];
                map.put(name, record);
            }
            // Win or Loss?
            if (gameResult.equals("W")) {
                record[0]++;
            } else { // must have been a loss
                record[1]++;
            }
        }
        // Now find the best team.
        double bestWinPercentage = -1;
        String best = "";
        for (String teamName : map.keySet()) {
            int[] record = map.get(teamName);
            double winPercentage = 1.0 * record[0]
                / (record[0] + record[1]);
            if (winPercentage > bestWinPercentage) {
                bestWinPercentage = winPercentage;
                best = teamName;
            }
        }
        return new Object[] {map, best};
    }
```

18 points, Criteria:

- create resulting map, 1 point
- loop correctly (+2 to index) through data array, 2 points
- if current team not present add key to map, 2 points
- correctly create and put array of int[] for new team / key, 2 points (lose if create unnecessary arrays)
- correctly access array with get for team / key already present, 2 points
- determine if win or loss, 1 point
- updated number of wins or losses in value correctly, 1 point
- variables for best win % and team name, 1 point
- correctly iterate through keyset of map, 2 points
- access values (int[]) for current key, 1 point
- calculate win % correctly, 1 point (lose if int /)
- check if best and update, 1 point
- return correct new object[], 1 point

Other deductions:

- using disallowed methods unless implemented, varies   -try to find best in main loop -5
- Alter data array, -4   -new, unnecessary data structures, -2   - get more than necessary, -2

3. Comments:

```
public int longestRun(E tgt) {
    // single loop approach
    int maxRun = 0;
    int currentRun = 0;
    Node<E> temp = first;
    while (temp != null) {
        if (temp.data.equals(tgt)) {
            // run still going!
            currentRun++;
            if (currentRun > maxRun)
                maxRun = currentRun;
        } else {
            currentRun = 0;
        }
        temp = temp.next;
    }
    return maxRun;
}
```

```
    // Double loop approach. O(N)
    int maxRun = 0;
    Node<E> temp = first;
    while (temp != null) {
        if (temp.data.equals(tgt)) {
            // Start of a run! .
            int currentRun = 0;
          while (temp != null
           && temp.data.equals(tgt)) {
                currentRun++;
                temp = temp.next;
            }
            // Longest?
            if (currentRun > maxRun) {
                maxRun = currentRun;
            }
        } else {
            // Not our target
            temp =temp.next;
        }
    }
    return maxRun;
}
```

```
public int longestRun2(E tgt) {
```

16 points, Criteria:

- variable to track maxRun init to value <= 0, 1 point
- temp Node variable initialized to first,1 point
- loop until ALL nodes in list checked / visited, 3 points (lose if OBOE)
- check if current node contains target, must use equals, 2 points
- when a run exists, correctly determine its length, 3 points
- determine if current run new max run, 2 point (lose if check necessary after loop and not done)
- make temp refer to next node in structure, 3 points
- return result, 1 point

Other:

- Disallowed methods, varies
- Destroys or alters list (typically do to altering first) -5
- NPE not covered by other crit -3
- O(N^2) -5

4. Comments:

```
public void update(int row, int col) {
    // Thinking ahead row and col may be out of bounds
    // nothing to do in that case. Likewise, if this
    // cell is already revealed nothing to do.
    // So, make sure inbounds and not already revealed.
    if (0 <= row && row < numMines.length
            && 0 <= col && col < numMines[0].length
            && !revealed[row][col]) {
        // reveal it
        revealed[row][col] = true;
        if (numMines[row][col] == MINE) {
            gameOver == true;
        } else if (numMines[row][col] == 0) {
            // Oh my, need to reveal my neighbors.
            for (int r = row - 1; r <= row + 1; r++) {
                for (int c = col - 1; c <= col + 1; c++) {
                    update(r, c); // update handles out of
                                  // bounds or already revealed.
                }
            }
        }
    }
}
```

16 points, Criteria:

- base case for out of bounds (maybe buried in loops), 2 points (Okay if helper method)
- base case for cell already revealed (maybe buried in loops), 3 points
- if cell not already revealed, reveal it, 1 point
- if cell is a mine, set game over boolean 1 point
- Check if revealed cell is a 0, 1 point
- For 0 cells loop through all neighbors, ATTEMPTED, 1 point (okay if hard code)
- For 0 cells loop through all neighbors, CORRECT, 3 points (lose if only check 4 dirs)
- Correct recursive calls on neighbors, 4 points (Lose if recurse on values > 0)

Other:

- Trying to return something, -4
- Adding a helper method besides out of bounds, -3
- Not accessing instance variables correctly, -3
- infinite loop for some other reason besides revealed, -4
- undo revealed cells, -3 (nothing to undo)
- helper added, -4