CS312 Fall 2019 Final 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.)
LE - Logic error in code.
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.

1. Expressions - 1 point each. -1 for missing or extra " OR .0. differences in capitalization of `true` and `false` okay. No limit on points off.

A. `14`
B. `"LLEN"`
C. `1.5`
D `false`
E. `"7CS212"`
F. `false`
G. `-1`
H. `5.0`
I. `'D'`
J. `5`

2. Code Tracing - 1 point each. Answer as shown or -1. -1 for first four occurrences of "'s. Differences in spacing, commas, and grouping symbols okay for array output.

A. `false`
B. `7 7 5 3`
C. `[1, -2, 15]`
D `[K, OO, X, GG]`
E. `7`
F. `12 -2`
G. `3 [ADA, C#]`
H. `2.0`
I. `[3, 2, 4]`
J. `[7, 0, 5, 0]`
K. `.5 each`
`legal`
`syntax error`
L. `MAIN0`
M. `1st0`
N. `APT 7`
O. `8 OAK0`
P. `SYNTAX ERROR (declared / static type of d4 is Dwelling. no hasGarage method in Dwelling class.)`
Q. `false`
R. `true PARK2`
S. `Won't compile because it tries to extend 2 classes. In Java every class, except Object, extends exactly one other class.`
T. `object`

3. Critters - 15 Points. Sloth class

```
public class Sloth extends Critter {

    private static final int CENTER_LIMIT = 10;

    private int timesCenter;
    private int timesUntilEat;
    private Direction nextMove;

    public Sloth(int timesUntilEat) {
        this.timesUntilEat = timesUntilEat;
        nextMove = Direction.CENTER;
    }

    public Direction move() {
        Direction result = nextMove;
        // get ready for next move
        if (nextMove == Direction.CENTER) {
            timesCenter++;
            if (timesCenter == CENTER_LIMIT) {
                nextMove = Direction.WEST;
                timesCenter = 0;
            }
        } else { // moving west this time
            nextMove = Direction.CENTER;
            timesUntilEat--;
        }
        return result;
    }

    public Attack fight(String oppo) {
        timesUntilEat--;
        Attack result = Attack.SCRATCH;
        if (nextMove == Direction.WEST) {
            result = Attack.ROAR;
        }
        return result;
    }


    public boolean eat() {
        return (timesUntilEat <= 0);
    }
```

} Points:
header correct with extends clause: 1 point
instance variables: 1 point (can be different)
instance vars private: 1 point
constructor that takes in for times until the sloth eats, init values correctly, 1 point
eat overridden correctly (doesn't eat until moved WEST and fought required number of times): 2 points
fight method: check correct next direction and return correct value, 3 points (lose this if call move)
updates number of times fought for eat response, 1 point
return correct value type, 1point

getMove
correctly handles returning CENTER or WEST depending on which move number, 3 points
correctly updates variable for number of times moved WEST for eat behavior, 1 point

Other Deductions:
-6 loop in getMove method / trying to move multiple times in getMove

4. Scanners 15 points.

```
public static String procesFile(Scanner sc) {
    int intSum = 0;
    double doubleSum = 0;
    int tokenSum = 0;
    while (sc.hasNext()) {
        if (sc.hasNextInt()) {
            intSum += sc.nextInt();
        } else if (sc.hasNextDouble()) {
            doubleSum += sc.nextDouble();
        } else {
            tokenSum += sc.next().length();
        }
    }
    String result = "I";
    if (doubleSum > intSum && doubleSum > tokenSum) {
        result = "D";
    } else if (tokenSum > intSum) {
        result = "T";
    }
    return result;
}
```

- variables to track 3 sums (array okay), 1 point
- correctly loop while hasNext true for Scanner, 2 points
- correctly check if hasNextInt first, 2 points
- correctly check hasNextDouble before reading as String, 2 points
- correctly call next methods, 2 points
- correctly update cumulative sum variables, 2 points
- correctly determine largest sum, 3 points
- return correct value, 1 point (only String of length 1)

Other Deductions:

- -3 creating new Scanners
- -2 OBOE
- -3 count any one value wrong (meaning doesn't sum correctly for int, double, or other tokens)
- -4, skip every other token with extra call to next in loop
- -2, scope of counters wrong (declared in loop)
- -4, early return
- -4, inner while loop for reading next tokens after while loops for int and double (this will read all of the rest of the tokens.)
- -3, any output to System.out
- -3, String concatenation
- -1, round off error

5. Arrays and Strings 15 Points.

```
public static String[] getSuffixArray(String s) {
    String[] result = new String[s.length() + 1];
    String suffix = "" + (char) 5;
    result[0] = suffix;
    int stringIndex = s.length() - 1;
    for (int i = 1; i < result.length; i++) {
        suffix = s.charAt(stringIndex) + suffix;
        stringIndex--;
        result[i] = suffix;
    }
    Arrays.sort(result);
    return result;
}
```

Criteria, 15 points:

- creating resulting array of correct size: 2 points
- correctly cast int 5 to char for use as sentinel character, 2 points
- correctly start with String with just sentinel char, 1 point
- loop with correct bounds (can vary from solution as long as it works), 3 points
- correctly call length and charAt methods on the String, 1 point
- correctly build up suffixes, 3 points (Can do with a nested loop. Lose this if call substring)
- add all suffixes to array, no suffixes skipped or left out, 2 points
- Call Arrays.sort once and correctly, 1 point

Other Deductions:

- -1 don't return result
- -3 any output
- -2, off by one errors (OBOE)
- -4, array index out of bounds exceptions (AIOBE)
- -4, using substring (disallowed by question)
- -3, any output to System.out

6. Arrays. 15 Points.

```
    // fast solution using mapping
    public static int[] lastIndexOf(int[] data, int n) {
        int[] result = new int[n + 1];
        for (int i = 0; i < result.length; i++) {
            result[i] = -1;
        }
        for (int i = 0; i < data.length; i++) {
            int x = data[i];
            if (0 <= x && x <= n) {
                result[x] = i;
            }
        }
        return result;
    }

    // slower solution using nested loop, this solution owuld get -1
    public static int[] lastIndexOf2(int[] data, int n) {
        int[] result = new int[n + 1];
        // look for each element
        for (int tgt = 0; tgt <= n; tgt++) {
            int index = data.length - 1;
            while (index >= 0 && data[index] != tgt) {
                index--;
            }
            if (index >= 0) {
                result[tgt] = index;
            } else {
                result[tgt] = -1;
            }
        }
        return result;
    }
```

Criteria, 15 points:

- create resulting array with correct length and type, 1 point
- nested loop solution, -1
- on nested loop solution, stopping inner when last index found, 4 points
- on nested loop solution, not starting from back of given array, 1 point
- correctly determine last index of target value (either mapping one loop, or nested loop) 3 points
- correctly set elements whose index do not appear in given array to -1, 3 points
- correctly access array elements, (no AIOBE possible) 3 points
- return result, 1 point

Other Deductions:

- disallow methods, -2 to -5 depending on severity
- efficiency (doing unnecessary work), -2
- -array index out of bounds exceptions (AIOBE), -4
- returns first index instead of last, -4
- stops too early, -7
- infinite loop, -4

7. ArrayList - 15 points - Question turned out to be much more difficult than intended.

```
public static void consolidateRecords1(ArrayList<BikeRaceRecord> list) {
        int i = 0;
        while (i < list.size()) {
            BikeRaceRecord current = list.get(i);
            int otherIndex = list.indexOf(current);
            if (otherIndex == i) {
                i++;
            } else {
                BikeRaceRecord other = list.get(otherIndex);
                if (other.getTime() < current.getTime()) {
                    list.remove(i);
                } else {
                    list.remove(otherIndex);
          // if there are more duplicates the others will be checked later
                    i++;
                }
            }
        }
    }
// front to back
public static void consolidateRecords2(ArrayList<BikeRaceRecord> list) {
        int i = 0;
        while (i < list.size()) {
            BikeRaceRecord current = list.get(i);
            int minTime = current.getTime();
            boolean removedCurrent = false;
            // find duplicate that is better
            int j = i + 1;
            while (j < list.size() && !removedCurrent) {
                BikeRaceRecord other = list.get(j);
                if (current.equals(other)) {
                    // duplicate, keep the smaller time
                    if (other.getTime() < minTime) {
                        list.remove(i);
                        removedCurrent = true;
                    } else {
                        list.remove(j);
                    }
                } else {
                    j++; // okay to go on
                }
            }
            if (!removedCurrent) {
                i++; // okay to go on
            }
        }
    }
```

```
// back to front
  public static void consolidateRecords3(ArrayList<BikeRaceRecord> list) {
        // go from the back to see if any easier
        int indexLast = list.size() - 1;
        while (indexLast >= 0){
            BikeRaceRecord current = list.get(indexLast);
            int indexFirst = list.indexOf(current);
            boolean removedLast = false;
            while (indexFirst != indexLast && !removedLast) {
                BikeRaceRecord closerToFront = list.get(indexFirst);
                if (closerToFront.getTime() > current.getTime()) {
                    list.remove(indexFirst);
                    indexFirst = list.indexOf(current);
                    indexLast--; // due to shifting
                } else {
                    // remove last
                    list.remove(indexLast);
                    removedLast = true;
                }
            }
            indexLast--;
        }
    }
```

Criteria, 15 points:

- outer loop, 1 point
- search after element for matching (loop) 3 points
- correctly list methods, 1point
- correctly check if records equal (call method or compare Strings with equals), 2 points
- correctly compare times, 1 point
- attempt to remove smaller record, 1 point
- correctly remove and adjust indices correctly, 3 points
- use size method for list correctly, 1 point
- attempt to not skip elements, 1 point
- code clearly does not skip elements

Other deductions:

- disallowed methods -2 to -5 depending on severity
- creating new arrays or ArrayLists, -5
- no inner loop, -10
- -array index out of bounds exceptions (AIOBE), -5
- [] instead of get method, -4
- possible to skip elements based on remove, -4
- triple loop, -6
- remove self, -5
- == instead of .equals(var) -2

8. 2D Arrays - 15 points

```
public static boolean isDistinctColumnSum (int[][] mat, int index) {
    // get the sum of the target column
    int tgtSum = getColumnSum(mat, index);

    // now check that no other columns have the same sum
    for (int c = 0; c < mat[0].length; c++) {
        if (c != index) {
            int columnSum = getColumnSum(mat, c);
            if (columnSum == tgtSum) {
                return false;
            }
        }
    }
    // never found a match, must be distinct
    return true;
}

private static int getColumnSum(int[][] mat, int index) {
    int result = 0;
    for (int r = 0; r < mat.length; r++) {
        result += mat[r][index];
    }
    return result;
}
```

Criteria, 15 points:

- determine sum of target column correctly, 2 points
- determine sum of other columns correctly, 4 points
- avoid false negative, don't compare target column to itself, 2 points
- return false as soon as we find a column that matches the target column sum, 4 points
- correctly access array elements, 2 points
- return true only if no match found, 1 point

Other Deductions:

- disallowed methods, -5 points off
- AIOBE, -4
- OBOE, first -2, second and subsequent, -1
- infinite loop, -6
- uses array, -2 (disallowed by question)