CS314 Fall 2018 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.)

LE - Logic error in code.

MCE - Major conceptual error. Answer is way off base, question not understood.

NAP - No answer provided. No answer given on test.

NN - Not necessary. Code is unneeded. Generally, no points off.

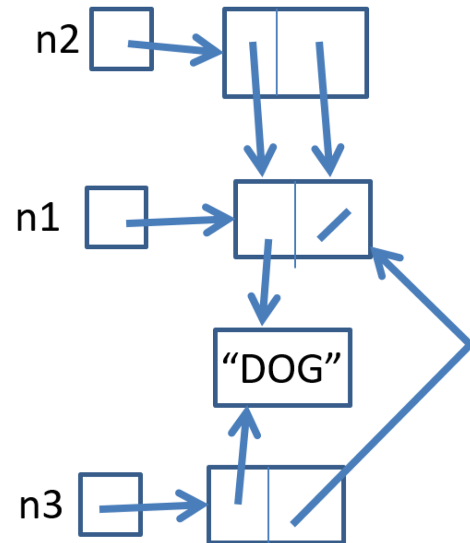
NPE - Null Pointer Exception may occur.

OBOE - Off by one error. Calculation is off by one.

RTQ - Read the question. Violated restrictions or made incorrect assumption.

1. Answer as shown or -1 unless question allows partial credit.
No points off for minor differences in spacing, capitalization, commas, and braces.

```
A.  16
B.  1 4 13 40 40 40 40
C.  130
D.  Runtime error (due to stack overflow error)
E.  -10
F.  210 seconds
G.  insertion sort, radix sort, mergesort
H.  -------------------------------------->
```




```
I.  54 seconds (method is O(N³))
J.  8 2 10 2 12
K.  A linked list so we have a method of
   accessing all values in the list one
   after the other in O(N) time.
   (or words to that effect)
L.  16
M.  UTCS!DF
N.  TCU!SFD
O.  CT!FDSU
```

2. Comments. Meant to be a simple tree problem. Some solutions didn't use the return value from the recursive call correctly. Some solutions referenced root instead of the parameter n. Several different ways of solving this problem.

```
private boolean helper(BNode n) {
    if (n == null)
        return true;
    else if (n.left == null && n.right != null)
        return false;
    else if (n.left != null && n.right == null)
        return false;
    else
        return helper(n.left) && helper(n.right);
}
```

9 points, Criteria:

- base case of empty tree correct, 2 points
- okay to check leaf and return true if do so AFTER checking null
- check one child cases correctly, 3 points
- make recursive calls correctly, 2 points
- return conjunction (and, &&) of results of 2 sub trees, 2 points (logical equivalents acceptable)

Other deductions:

other helper added, -3
not using return value, -3
while loop, -4
referencing root instead of n, -2

3. Comments: This was a very simple map problem. Efficiency was emphasized. If the maps are different sizes it is much faster to determine which map is smaller and loop through that one. No need to create a new map or loop through both maps.

```
public static int dotProduct(Map<String, Integer> m1,
        Map<String, Integer> m2) {

    int result = 0;
    Map<String, Integer> smaller, larger;
    // loop through the smaller map
    smaller = m1.size() <= m2.size() ? m1 : m2;
    larger = smaller == m1 ? m2 : m1;
    for (String key : smaller.keySet()) {
        Integer freqLargerMap = larger.get(key);
        if (freqLargerMap != null) {
            result += smaller.get(key) * freqLargerMap;
        }
    }
    return result;
}
```

19 points, Criteria:

- correctly selects the smaller map to loop through, 3 points
- only loop through one map, 4 points
- loop through keys of map correctly, 3 points
- access elements from other map with get correctly, 3 points
- correctly check return value not null before performing computation, 2 points
- correctly calculate and track cumulative some of product of frequencies, 3 points
- return correct result, 1 point

Other:

- nested loop, O(NM) N = m1.size(), M = m2.size(), -7
- creating a new map, -4

4. Comments:

```java
public boolean removeFirstOccurrenceStartIndex(E tgt, int start) {
    Node<E> temp = header;
    // move to the node at index (start - 1)
    int index = 0;
    while (index < start && temp.next != null) {
        index++;
        temp = temp.next;
    }
    // now search for the first occurrence of target
    while (temp.next != null) {
        // is the next node the one with the data?
        if (temp.next.data.equals(tgt)) {
            temp.next = temp.next.next;
            return true;
        }
        temp = temp.next;
    }
    // never found target
    return false;
}
```

19 points, Criteria:

- while loop to get to node before start index correct:
  - check index < start (or logical equivalent), 2 points
  - check temp.next !- null, 2 points (temp == null can work)
- move temp correctly in loop, 3 points
- second while loop to search for first occurrence of target correct, 2 points (temp.next != null)
- in loop correctly check if next node's data equals target, 2 points (lose if ==)
- correctly removes node, 3 points
- correctly moves temp, 2 points
- stops as soon as first occurrence found, 2 points
- return correct result, 1 point

Other deductions:

- NPE possible, -5
- worse than O(1) space, -5
- remove more than 1 element, -5
- checking temp.data == null to determine if we have go off end of list, -3
- destroys list by not using temp Node reference, -6
- infinite loop due to not moving through list correctly, -6
- treating the header node as the first node with data. NPE due to calling method on header.data, -3, trying to change what node header refers to (it is declared final), -2
- assuming a size instance variable exists, -4
- assuming null data means node doesn't exist, -4
- not attempting to move past start location, -8
- loop through list twice instead of once, -2
- recursive solution that works, - 4 (O(N) space vs. O(1) space)

5.

```java
public boolean containsAll(LinkedList314<E> other) {
    Node<E> tempOther = other.first;
    while(tempOther != null) {
        E tgt = tempOther.data;
        if (!this.present(tgt)) {
            return false;
        }
        tempOther = tempOther.next;
    }
    return true; // found a match for each element in other
}

private boolean present(E tgt) {
    Node<E> temp = first;
    while (temp != null) {
        if (temp.data.equals(tgt)) {
            // found tgt! We are done.
            return true;
        }
        temp = temp.next;
    }
    // never found tgt in this list
    return false;
}
```

19 points, Criteria:

- outer temp variable starting at first correctly, 1 point
- outer loop correctly goes through all elements of other, 4 points (partial credit possible)
- inner loop (or helper method) correctly searches traverses calling object from the start, 4 points (partial credit possible)
- correctly compares data correctly. Meaning compare data from nodes not nodes, 3 points
- correctly uses equals, 1 point (lose if ==)
- inner loop stops (or method returns true) as soon as match found, 2 points
- outer loop stops (returns false) on first element without match, 2 points
- return true if all elements in other have match, 2 points

Other:

- OBOE, -3 (temp.next == null without look ahead code)
- alter either or both lists, -6
- NPE, -5
- Missing inner loop, -6
- recursive solution that works, - 4 (O(N) space vs. O(1) space)
- using nodeVar.data == null to first.data == null to check empty instead of nodeVar == null or first == null, -3
- checking if other contains all elements of this list instead of checking if this list contains all elements of other, -5
- early return, -8
- No inner loop, -11
- not moving references, -3
- not resetting inner loop, -3
- new data structures, -5

6. Comments: A classic recursive backtracking problem. The problem statement required the method to deal with a single rectangle. Many solutions had a loop to go through the array of rectangles. This is both inefficient and contradicts the instructions. Many solutions didn't consider the base case when there are no Rectangles left to consider and the goal is not met. Important to check the goal met first; perhaps the last Rectangle led to the solution. Many solutions failed to consider the choice or option of not using the current Rectangle. We don't have to use every Rectangle, only a subset, so not using one is a valid option.

```java
    private static boolean helper(int i, Rectangle[] rects,
            boolean[][] mat, Rectangle goal) {

        if (allMatch(0, 0, goal, true, mat))
            return true; // pop, pop, pop. Covered all elements!
        else if (i == rects.length)
            return false; // no more options, too bad :(

        // deal with the current Rectangle.
        // Options are upper left corner to place it in.
        Rectangle currentRect = rects[i];
        int rowLimit = mat.length - currentRect.height;
        int colLimit = mat[0].length - currentRect.width;
        for (int r = 0; r <= rowLimit; r++) {
            for (int c = 0; c <= colLimit; c++) {
                if (allMatch(r, c, currentRect, false, mat)) {
                    // we can place current rectangle at r, c
                    setVals(r, c, currentRect, true, mat);
                    if (helper(i + 1, rects, mat, goal)) {
                        return true; // It worked!!!!
                    }
                    // didn't work, pick up the rectangle
                    setVals(r, c, currentRect, false, mat);
                }
            }
        }
        // One last chance! Don't use this Rectangle at all!!
        return helper(i + 1, rects, mat, goal);
    }
```

19 points, Criteria:

- base case for all cells covered, 2 points
- base case for no Rectangles left, 2 points
- nested loop for options, upper left corner to place Rectangle, 4 points
- correctly check that current option works (calling all match correctly), 2 points
- if works, place rectangle, set boolean, 1 point
- correctly make recursive call and return true if result is true, 4 points
- undo rectangle if didn't work, 2 points
- after all options, try the last option of not using the rectangle, recursive call at end, not simply return false, 2 points

Other:

- early return -6
- not using return value correctly, -5
- other helper, -5
- trying to handle more than Rectangle per call, -5
- i++ on argument (logic error), -2
- new data structures, -6