CS314 Spring 2014 Midterm 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.)
GCE - Gross Conceptual Error. Did not answer the question asked or showed fundamental misunderstanding
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. Answer as shown or -1 unless question allows partial credit.
No points off for minor differences in spacing, capitalization, commas, and braces OR missing O( )
If """ included in output -1 first occurrence then error carried forward.

A. **O(N)**
B. **O(N)**
C. **O(N^2)**
D. **O(N^2)**
E. **27**
F. **18**
G. **765!31-1**
H. **10**
I. **45 seconds**
J. **42 seconds**
K. **40 seconds**
L. **12**
M. **O(NlogN)base 2 okay**
N. **0 0 1 0 1 2 0**
O. **Don't sort. Just use linear search:**

$10 * 1{,}000{,}000 < 1{,}000{,}000 \log_2 1{,}000{,}000 + 10 * \log_2 1{,}000{,}000$
$10{,}000{,}000 < 1{,}000{,}000 * 20 + 10 * 20$
$10{,}000{,}000 < 20{,}000{,}000$

using 500,000 instead of 1,000,000 on left side (not sorting) ok

P. **Syntax error (can't instantiate interface) OR compile error**
Q. **XQMGVAHQK**
R. **GMQVAXHKQ**
S. **GMAVQKQHX**
T. **14 [5, -1, 4, 6]**

2. Comments. A very simple problem. We essentially did this in class when we covered stacks.

Common problems:

- st = someVariable; // does not restore stack. Recall, value parameters
- using > instead >=

- trying to return something. Method was void

Suggested Solution:

```
public static <E extends Comparable<E>> void remove(Stack<E> st,
                                                    E cutoff) {

    Stack<E> temp = new Stack<E>();
    while(!st.isEmpty()) {
        E val = st.pop();
        if(val.compareTo(cutoff) >= 0)
            temp.push(val);
    }
     while(!temp.isEmpty())
        st.push(temp.pop());
}
```

15 points , Criteria:

- Create temp stack, 3
- loop through given stack, 2
- remove from given stack, 2
- use compareTo correctly: 3
- push on temp stack, 1 point
- restore items in temp to st, 4 points (lose this if try to assign st a value)

Other:

disallowed classes or methods: -3
Worse than O(N): -3
> instead of >=: -1
returning a stack instead of restoring stack: -3
not using E correctly: -1
putting in items backwards: -3

3. Comments: A typical LinkedList question. Possible to write it so no special cases existed. Never have to alter first or last. A neat problem because there were lots of good solutions.

Common Problems:

- getting a Null Pointer Exception when list empty
- checking nodes equal target instead of data in nodes equal target
- not stopping after removing first range. There could be others but those shall not be removed based on the precondition
- altering list if only one occurrence of target found
- not actually iterating through list, infinite loop
- logic errors with OR || instead of AND &&
- using == on tgt and data instead of equals
- if(!tempVar.equals(null)) still causes a NullPointerException if tempVar stores null

Suggested Solutions:

```
public void removeBetween(E tgt) {
    int num = 0;
    Node<E> temp = first;
    Node<E> firstOccurrence = null;
    while(temp != null && num < 2) {
        if(temp.getData().equals(tgt)) {
            num++;
            if(num == 1)
                firstOccurrence = temp;
            else
                firstOccurrence.setNext(temp);
        }
        temp = temp.getNext();
    }
}
```

```
public void removeBetween(E tgt) {
    Node<E> start = first;
    // search until we find tgt or hit last node
    // handle empty case (start == first == last == null)
     while(start != last && !start.getData().equals(tgt))
        start = start.getNext();

    // if start != last, found a value
    if(start != last) {
        // search for the second value
        Node<E> stop = start.getNext();
        while(stop != null && !stop.getData().equals(tgt)) {
            stop = stop.getNext();
            count++;
        }
        if(stop != null) {
            start.setNext(stop);
        }
        // no cases when we have to update first or last
    }
```

15 points , Criteria:

- temp node starting at first, 1
- search for first occurrence of target, 3
- correctly move temp, 3
- if found first occurrence look for second, 1
- second temp, 1
- correctly search for second occurrence of tgt, 3
- if second occurrence found, update correctly to remove values, 3
- not necessary to null out data or next, although this would be good practice

Other:
DISALLOWED METHODS -3 per
NPE on empty list: -2
not calling getData() -2
== instead of .equals on data: -2
destroying parts of list or removing parts of list that should not be removed: -5

4A. Comments: Dealing with two maps. A lot of folks chose to iterate through the keyset of borders instead of coloredAreas. This led to possible problems.

Common problems:

- By far the most common problem was assuming all keys in coloredAreas were in borders as well. The precondition stated this may not be the case. (number of keys in coloredAreas may be less than number of keys in borders.) In that case get returns null and then a Null Pointer Exception occurs when calling equals.
- Also problems calling compareTo correctly.

Suggested Solution:

```
public static boolean colorsOkay(Map<String, List<String>> borders,
                                 Map<String, Color> coloredAreas) {

     Iterator<String> areasIt = coloredAreas.keySet().iterator();
    while(areasIt.hasNext()) {
        String area = areasIt.next();
        // check if any of the areas that border area have same color
        Iterator<String> borderAreas = borders.get(area).iterator();
        while(borderAreas.hasNext()) {
            String borderArea = borderAreas.next();
            if(coloredAreas.containsKey(borderArea)) {
                Color c1 = coloredAreas.get(area);
                Color c2 = coloredAreas.get(borderArea);
                if(c1.equals(c2))
                    return false;
            }
        }
    }
    return true;
}
```

15 points, Criteria:

- iterator for coloredAreas or foreach loop, 1
- get bordering areas for current colored areas, 3 (iterator or foreach loop)
- check if current border area is in the coloredAreas map, 3
- if so, get colors for both areas, 3
- check if equal 2, (== okay here, not a great idea, but okay)
- stop as soon as two bordering areas with same color found, 2
- return correct value, 1

Other deductions:
disallowed methods, -3 per
null pointer exceptions: -3

4B. Comments: A classic recursive backtracking problem. The algorithm follows the recursive backtracking pseudo-code pattern almost exactly. (Note ignoring 0 length borders a map can be colored with 4 colors, but you still have to do the recursion to find a valid coloring.)

Common problems:

- returning early
- not checking colors okay before proceeding. Very inefficient and I decided that was worth a point.
- not getting base case correct
- some form of nested loop. This could undo past changes and is even less efficient
- not removing the last mapping if none worked for a current area. This could lead to false negatives when backing up to previous methods because an area remains colored when it should not be.

Suggested Solution:

```
private static boolean helper(Map<String, List<String>> borders,
        Color[] colors, List<String> areas, int currentArea,
        Map<String, Color> result) {

    // base case, all areas assigned a color with no conflicts
    if(currentArea == areas.size())
        return true;
    else {
        // try the choices for the current area
        String area = areas.get(currentArea);
        int nextArea = currentArea + 1;
        for(int i = 0; i < colors.length; i++) {
            result.put(area, colors[i]);
            if(colorsOkay(borders, result)) {
                boolean solved = helper(borders, colors, areas,
                                                        nextArea, result);
                if(solved)
                    return true;
            }
            // loop will try the next color
        }
        // no colors worked, remove last color from map
        result.remove(area);
    }
    return false;
}
```

15 points, Criteria:

- base case correct with return, 3
- loop through color choices, 3
- place color and area in map, 1
- check if colors okay before proceeding, 1
- make recursive call correctly with next area, 2
- if true, return true, 3
- remove from area and color from map after loop (inside okay, but not before return true), 1
- return false after loop, 1

Common Deductions:

Early return -5