CS314 Spring 2015 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.)
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.

A. **15**

B. **GIU_A!**

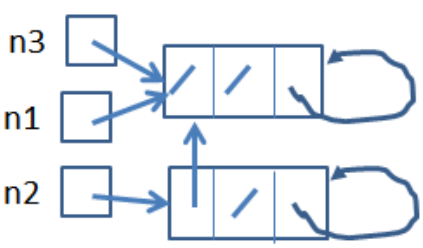
C. **190**

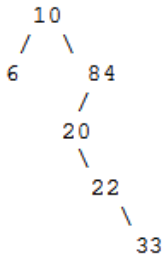
D. **Compiler error OR Syntax error**

E. **O(N^2)**

F. **O(N)** // linked list remove with iterator O(1)

G. **O(N^2)** // remove for ArrayList O(N) even with iterator

H. 



BST for S.

I. 52.5 or 105 / 2 seconds (no logs in answer)

J. **Insertion sort**

K. **320 seconds**

L. **-17**

M. **(3 + 2) * (17 - 4)**

N. **5 10 5 3 8 5 OR Compile Error**

O. **r s j**

P. **A B D G H C E I F**

Q. **B G D H A E I C F**

R. **G H D B I E F C A**

S. **See above next to node drawing on H**

T. **2000 seconds**

2. Comments. A simple toy problem using Stacks and Queues

Common problems:

- not dealing with first element. Topping an empty stack generally causes an exception.
- not handling the case of an empty queue
- using == instead of .equals

Suggested Solution:

```
public static <E> void removeConsecutiveDuplicates(Queue<E> q) {
    Stack<E> st = new Stack<E>();
    while(!q.isEmpty()) {
        E element = q.dequeue();
        if(st.isEmpty() || !element.equals(st.top())) {
            st.push(element);
        }
    }
    while(!st.isEmpty())
        q.enqueue(st.pop());
}
```

20 points , Criteria:

- create Stack, 2 points
- while loop for queue, 5 points
- push element from queue only if Stack empty or top element does not match, 5 points
- while loop for stack, 5 points
- enqueue and pop correctly, 3 points

3. Comments: A decent LinkedList problem. Not too easy, not too hard. Dealing with consecutive elements was the real trick.

Common problems:

- comparing nodes (which are not Comparable) instead of the data in the nodes
- Null Pointer Exception on the last node.
- not advancing through the list
- not dealing with empty list correctly
- Using O(N) space instead of O(1) space
- destroying the list
- $O(N^2)$ solution instead of O(N)

Suggested Solution:

```
public boolean isSorted() {
    if(first == null)
        return true; // trivial case
    // 1 or more elements
    E previousData = first.getData();
    Node<E> temp = first.getNext();
    boolean sorted = true;
    while(sorted && temp != null) {
        E currentData = temp.getData();
        sorted = previousData.compareTo(currentData) <= 0;
        previousData = currentData;
        temp = temp.getNext();
    }
    return sorted;
}
```

20 points , Criteria:

- handle case when list empty (okay for 1 element as well), 3 points
- temp node variable assigned value in first, 1 point
- loop until end of list correctly, 2 points
- correctly compare consecutive values, 4 points
- stop as soon as answer known, 3 points
- move through linked structure of nodes correctly, 6 points
- return correct result, 1 point

4. Comments: A lot of code to write for this. A lot of abstractions to deal with. Determining the number of problems solved was just like the map example we did in class. A good problem because there were many different, viable solutions.

Common problems:

- assuming map is Iterable
- assuming sets have a get based on position
- adding frequency to result instead of problem number
- $O(N^2)$ instead of O(N) where N is the total number of problems solved
- calling contains on map instead of containsKey
- accessing maps and sets like arrays

```
public static TreeSet<Integer> getMostSolverProblems(Map<String,
                                                    Set<Integer>> solved) {
        HashMap<Integer, Integer> freqs = new HashMap<Integer, Integer> ();
        // determine frequency of problems solved
        for(String name : solved.keySet()) {
            for(int problem : solved.get(name)) {
                if(freqs.containsKey(problem)) {
                    int prev = freqs.get(problem);
                    freqs.put(problem, prev + 1);
                }
                else
                    freqs.put(problem, 1);
            }
        }
        // find the problem solved the maximum number of times
       // (could track max in previous part as well)
        int max = Integer.MIN_VALUE;
        for(int problem : freqs.keySet()) {
            int numSolved = freqs.get(problem);
            if(numSolved > max)
                max = numSolved;
        }
        // add problems solved max number of times to result
        TreeSet<Integer> result = new TreeSet<Integer>();

        for(int problem : freqs.keySet()) {
            int numSolved = freqs.get(problem);
            if(numSolved == max)
                result.add(problem);
        }
        return result;
    }
```

20 points, Criteria:

- use HashMap<Integer, Integer> to correctly determine number of times each problem solved, 9
  - includes obtaining key, obtaining value, using iterators or for-each loop correctly, getting and putting in HashMap correctly
- determine which problem solved the most, 5
- add all problems solved the max number of times to result, 5
- return result, 1

5. Comments: A nifty recursive backtracking problem. For the most part students did well.

Common problems:

- stopping when path total greater than target (or target less than zero if subtracting node data from target) Negative values lower in the tree may make it possible so find a path equal to the total. There was an example like this in the question. Target of 3 = 5 + -2 = 3
- not handling case when target == 0 and tree is NOT empty (trivially true)
- not adding root to path total
- not checking base case on leaf nodes after adding there data to total
- destroying the tree

Suggested Solution:

```
public boolean hasPath(int tgt) {
    if(tgt == 0)
        return true;
    else if(root == null)
        return false;
    return hasPathHelp(root, tgt, root.data);
}

private boolean hasPathHelp(IntNode n, int tgt, int pathTotal) {
    // base case, DONE! no more node's necessary
    if(pathTotal == tgt)
        return true;
    else {
        // try children in path
        for(IntNode child : n.children) {
            boolean solved = hasPathHelp(child, tgt,
                                         pathTotal + child.data);
            if(solved)
                return true;
        }
        // no good
        return false;
    }

     }
```

20 points, Criteria:

- kickoff method handles special cases if tree empty or target 0, 1
- kickoff calls helper, 1
- helper method created, 1
- helper adds current nodes value to total and correctly uses all nodes in path (or subtracts from goal), 3
- checks base case correctly, 4
- if not at base case, tries children, 4
- returns only if solved, 5 (not an early return on first result)
- if children don't work, returns false, 1