CS314 fall 2016 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.

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. 36

B. 6 5 =4= 3 2 (differences in spaces okay)

C. 23

D. -516-8-4-2-1 (ignore dashes)

E. O(N^2)

F. O(N^2)

G. 15 10 6 (on different or same line)

H. 6

I. 40 minutes

J. 42 seconds

K. 15 7 3

L. .88 seconds

M. two of:

- has one or more abstract methods
- implements an interface but doesn't implement all methods of interface
- inherits from an abstract class with abstract methods and doesn't implement those methods

N. beginning of list should be top of stack

O. .01 seconds (makeTree is O(N) in this case, adding N duplicates)

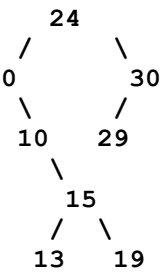
P. 3

Q. 5 0 7 4 8 9 2

R. 0 4 7 8 5 9 2

S. 4 8 7 0 2 9 5

T.



2. Comments. A simple linked list question. Students did quite well on this question

Common problems:

- using disallowed methods. equals was the only method that could be used
- not using compareTo correctly
- assuming compareTo returns only -1, 0, or 1

```
public void replaceSmaller(LinkedList314<E> otherList) {
     Node<E> t1 = first;
     Node<E> t2 = otherList.first;
     while( t1 != null && t2 != null) {
          if (t1.data.compareTo(t2.data) < 0) {
               t1.data = t2.data;
          }
          t1 = t1.next;
          t2 = t2.next;
     }
}
```

16 points , Criteria:

- temp nodes for each list, 1 point
- while loop correct, 5 points
- check and replace smaller element in this correctly, 4 points
- traverse through nodes in list correctly, 6 points

Usage errors:
using disallowed methods, -6
adds public method to get first, -5
off by one errors, - 3

3. Comments: A more difficult LinkedList question. Lots of special cases to worry about.

Common problems:

- if number to remove is 0, there is nothing to do. Don't waste time moving through the linked list if number is 0
- not handling the case when start is 0 and first must be updated
- off by one errors when removing nodes
- not guarding against number being larger than the number of nodes we can actually remove
- using disallowed methods
- moving to node at position start instead of start - 1. The node at position start is to be removed.

```
public int removeNum(int start, int number) {
      int count = 0;
      if (number > 0) {
            // actually have to remove some nodes
            if (start == 0) {
                  // handle case when must update first
                  while (first != null && count < number) {
                        first = first.next;
                        count++;
                  }
            } else {
                  // general case, must get to start node
                  Node<E> temp = first;
                  for (int i = 1; i < start; i++)
                        temp = temp.next;
                  // temp now on node at position start - 1;
                  while (temp.next != null && count < number) {
                        temp.next = temp.next.next;
                        count++;
                  }
            }
            return count;
```

16 points, Criteria:

- efficiency, do nothing if number == 0, 2 points
- handle case when start = 0 and first must be updated, 3 points
- got to correct node in list (the one before start), 4 points
- remove correct nodes, 5 points
- calculate number of node removed, 1 point
- return correct result, 1 point

Other deductions:

4. Comments: An interesting Stack problem. Allowing use of Stack or Queue made it harder. In order to solve the problem correctly it is necessary to use a Stack, not a Queue. Part of the question was realizing this. There is also a special case when the Stack is empty and we cannot remove any elements.

Common problems:

- popping or topping an empty stack
- using a queue
- comparing to the original top element only instead of the element above. So for example some people simply stored the top element and compared to that. So top [12, 5, 8] bottom. The 8 must be removed because it is not less than or equal to 5. Comparing it to 12, the original top, is a logic error.

Suggested Solution:

```
public void makeDescending(Stack314<Integer> st) {
      if (!st.isEmpty()) {
            Stack314<Integer> temp = new Stack314<Integer>();
            temp.push(st.pop()); // we know not empty

            while (!st.isEmpty()) {
                  int x = st.pop();
                  if (x <= temp.top()) {
                        temp.push(x);
                  }
            }

            while (!temp.isEmpty()) {
                  st.push(temp.pop());
            }
      }
}
```

16 points, Criteria:

- handle empty case correctly, 4 points (lose if possible to top or pop empty stack)
- process elements in st with while loop, 2 points
- check if element to be added to temp correctly and push, 3 points
- restore elements from temp to st, 3 points
- element in correct order, uses stack, 4 points

Others:

5. Comments: A simple tree problem

Common problems:

- when a node met the criteria NOT exploring its left child
- not handling empty tree case
- using while loops instead of recursion. (not possible without aux data structures)
- not using equals method to check data
- not checking all three conditions for counting node met
- trying to use a parameter to track the count. Recall, value parameters. If we make a copy and increment that it does not alter the original argument.
- Not passing the value to check as a parameter.

Suggested Solution:

```
public int numNodesWithValueAndLeftChildOnly(E value) {
      return help(root, value);
}

private int help(BNode<E> n, E value) {
      if (n == null) {
            // dead end, nothing to see here
            return 0;
      else if (n.left != null && n.right == null && value.equals(n.data) {
            // We have a winner! Count this one and check left child.
            // No need to check right, because it is null
            return 1 + help(n.left, value);
      } else {
            // Whelp, this node didn't meet the criteria, just check children
            return help(n.left, value) + help(n.right, value);
      }
}
```

16 points, Criteria:

- create helper method with proper parameters, 3 points
- handle base case of empty tree / n == null, 4 points
- check if current node meets criteria correctly and return correct result, 4 points
- recursive calls correct in all cases, 4 points
- return correct, int result, 1 point

Other:
not handling empty tree case / NPE possible, -3
not using .equals -3
major logic error with while loops, -12

6. Comments: Not the hardest recursive backtracking question in the world, especially as the magic board class finds all the valid moves for you. It was acceptable to alter the permanently, not undo the correct moves. No helper method was necessary.

Common problems:

- base case of 0 marbles on the board. This is not possible unless the board starts with 0 marbles. A solution is one marble is left on the board. It is not possible to go from a board with 1 marble to one with 0.
- Only using the moves from the first board configuration.
- Trying to track the index to start at for the moves. The moves for each board will be different and so we must start at 0 for each array of moves
- not undoing the move if a solution no reached.
- returning early instead of only returning if result is true
- not returning true as soon as we get that value. If we have solved the puzzle lets not waste any more time. We are done. Pop, pop, pop the champagne corks.

Suggested Solution:

```
public boolean canBeSolved(Board board) {
      if (board.numMarblesOnBoard() == 1) {
            return true; // HOORAY!!!
      }
      Move[] moves = board.getMoves();
      // current choices are the moves for this board
      for (int i = 0; i < moves.length; i++) {
            Move m = moves[i];
            // make current move
            board.removeMarble(m.sourceRow(), m.sourceCol());
            board.removeMarble(m.removedRow(), m.removedCol());
            board.placeMarble(m.destRow(), m.destCol());
            if (canBeSolved(board)) {
                  // done!!!
                  return true;
            }
            // that didn't work out, undo it so we can try the next one
            board.placeMarble(m.sourceRow(), m.sourceCol());
            board.placeMarble(m.removedRow(), m.removedCol());
            board.removeMarble(m.destRow(), m.destCol());
      }
      // never found a solution with these choices.
      return false;
}


}
```

16 points, Criteria:

- correct base case, 4 points
- loop through correct choices, 2 points
- move marbles correctly, 1 point
- make correct recursive call, 4 points
- return true if solved with this move, 3 points
- return false if no moves worked out, 2 points