CS314 Spring 2019 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. (Statements in parenthesis not required, only for explanation of answer to students.)
No points off for minor differences in spacing, capitalization, commas, and braces.

A. 13

B. -5

C. L2O*3G1   (-1 if any quotes)

D. 58

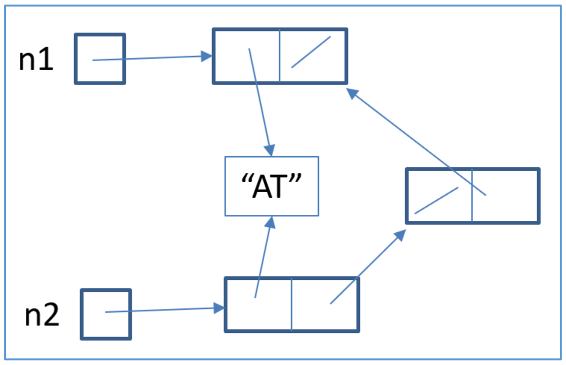
E. Either end may be used as the front. OWTTE (Statements saying the beginning of the list should be used as the front to keep the implementation intuitive, must still state either end could be used.)

F. 9 seconds (O(N^2), this is worst case for insertion sort)

G. selection, insertion, quicksort

H. 0.2 seconds (O(N), granted gacky, but no runtime error. We don't use the iterator again after using the list remove. Question based on something Anthony saw when grading linked list assignment.)

I. ------------------------------------->



J. 80 seconds (code is O(N^3))

K. Revised Question. Call t1.get(i) once and store the result in a local variable before the inner loop.

L.
```
                37
               /  \
             12    73
            /  \     \
           0    20    101
```

M. 0 20 12 101 73 37

N. 9

O. 20 seconds

2.
```
public boolean rangeEqualsTarget(int tgt, int start, int stop) {
     if (start == stop) {
        // no elements in range
        return tgt == 0;
     }
     IntNode temp = first;
     int index = 0;
     // move to the start node
     while (temp != null && index < start) {
         index++;
         temp = temp.next;
     }
     // now sum up the values in the range
     int sum = 0;
     while (temp != null && index < stop) {
         index++;
         sum += temp.data;
         temp = temp.next;
     }
     return sum == tgt;
}
```

17 points, Criteria:

- return immediately if start == stop (efficiency, 1 point)
- get to node at index start in the list correctly, max 6 points off
  - not checking null, -3
  - not moving temp, -3
  - use while loop, but don't increment index (go all the way through), -3
- variable to track sum of values in range, 1 point
- move through nodes in range if they exist, max 8 points off
  - not checking null, -3
  - not checking index < stop, -2 points
  - accessing data incorrectly, -2 points
  - not moving temp, -3 points
  - only checking part of range, early return, -3
  - not summing up whole range correctly, -3
  - got past stop, -2 (eff.))
- return correct result, 1 point

Other deductions:
- off by one errors, -2
- infinite loops (other than not moving temp), -4
- compare ints to null, -2
- go past stop index -2

3.
```java
public LinkedIntList combineIgnoreValue(LinkedIntList other, int tgt) {
    LinkedIntList result = new LinkedIntList();
    IntNode thisTemp = first;
    IntNode otherTemp = other.first;
    IntNode lastResult = result.first;
    while (thisTemp != null && otherTemp != null) {
        if (thisTemp.data != tgt && otherTemp.data != tgt) {
            lastReuslt = add(result, lastResult, thisTemp.data + otherTemp.data);
        }
        thisTemp = thisTemp.next;
        otherTemp = otherTemp.next;
    }
    IntNode temp = (thisTemp != null) ? thisTemp : otherTemp;
    while(temp != null) {
        if (temp.data != tgt) {
            add(result, lastResult, temp.data);
        }
        temp = temp.next;
    }
    return result;
}

private void add(LinkedIntList l, IntNode last, int val) {
    IntNode n = new IntNode();
    n.data = val;
    if (last == null) {
        l.first = n;
    } else {
        last.next = n;
    }
    return n;
}
```
17 points, Criteria:

- handle nodes that exist in both:
    - stop this case when one or both temps are null, 2 points
    - skip nodes that contain target value, 3 points
    - move temp variables / traverse list correctly, 3 points
- add nodes to resulting list (-5 if no new nodes created)
    - handle case for first, 2 points
    - general case, 3 points
- handle remaining nodes in larger list, 3 points (partial credit possible)
- return result, 1 point

Others:

- NPE, -3
- worse than O(1) space, -5
- worse than O(N) time -2 to -5
- calling add method without implementing, -5
- size method -3
- destroys list, -5
- try to compare int to null
- add one extra node, off by one error, -2

4.

```
public static <E> boolean sameBottomN(Stack<E> s1, Stack<E> s2, int n) {
    Stack<E> t1 = new Stack<>();
    Stack<E> t2 = new Stack<>();
    dump(s1, t1);
    dump(s2, t2);
    int count = 0;
    boolean same = true;
    while (same && !t1.isEmpty() && !t2.isEmpty() && count < n) {
        same = t1.top().equals(t2.top());
        if (same) {
            count++;
        }
        s1.push(t1.pop());
        s2.push(t2.pop());
    }
    dump(t1, s1);
    dump(t2, s2);
    return count == n;
}
// empty the contents of org into dest
private static <E> void dump(Stack<E> org, Stack<E> dest) {
    while (!org.isEmpty()) {
        dest.push(org.pop());
    }
}
```

17 points, Criteria:

- correctly create temp stacks and empty originals into temporary stacks, 3 points
- correctly check from bottom if elements the same (count and / or boolean), 4 points (partial credit possible)
- use .equals correctly, 2 points
- handle case when not enough elements correctly, 2 points
- restore stacks, 5 points (lose if early return, -4 if partial, -2 if don't restore smaller correctly)
- return result, 1 point

Other:

- pop or top empty stack, -3
- checking tops equal instead of bottom, -7
- top == null for empty check, -3
- infinite loop, -5
- no attempt to restore stacks, -5
- incorrect calls to stack methods, -1

5.

```
    public int makeFull() {
        if (root == null) {
            return 0;
        } else {
            return help(root);
        }
    }

    private int help(BNode n) {
        // With check in public method, we should
        // never reach a case where n == null.
        // are we a leaf node?
        if (n.left == null && n.right == null) {
            return 0;
        } else if (n.left == null) {
            // not a leaf node, at least one child
            // no child to left, must add it
            n.left = new BNode(n.data);
            // but we know we have a right child
            return 1 + help(n.right);
        } else if (n.right == null) {
            // no child to right, must add it
            n.right = new BNode(n.data);
            // but we know we have a left child
            return 1 + help(n.left);
        } else {
            // 2 children, don't add any nodes here
            // but what do my children do?
            return help(n.left) + help(n.right);
        }
    }
```

17 points, Criteria:

- handle case when tree is empty, 1 point
- base case, leaf node or null, do nothing, return 0, 4 points
- if no child to left or right, add node correctly, 4 points (partial credit possible)
- if added child return 1 + result of recursive call(s), 4 points
- if internal node with 2 children, return 0 + result of recursive calls, 4 points

Other:

- No helper method -14
- null pointer exception possible, -3
- not creating new nodes, -3s

6.

Suggested Solution:
```
// I use -1 to indicate no path found, but it might stil exist
// and -2 to indicate I am sure no path can exist (because we
// found dest before we found start) so stop trying.
private int help(int start, int dest, TNode n, int distanceFromStart) {
    if (n != null ) {
        if (n.data == dest) {
            // We found the destination node.
            // Have we seen start already?
            if (distanceFromStart == -1) {
                return -2; // IMPOSSIBLE
            } else {
                return distanceFromStart; // pop, pop, pop
            }
        } else if (n.children != null){
            // recursive case
            // Is this the startNode?
            if (n.data == start || distanceFromStart > 0) {
                distanceFromStart++;
            }
            for (int i = 0; i < n.children.size(); i++) {
                int result = help(start, dest, n.children.get(i), newDis);
                if (result == -2) {
                    // IMPOSSIBLE! Don't waste any more time;
                    return -2;
                } else if (result > 0) {
                    // success!!!
                    return result;
                } // else we got -1, try next child
            } //end of for
        } // end of else if (n.children != null)
    } // end of if (n != null)
    // n is null, no children, or never found it
    return -1;
}
```
17 points, Criteria:

- handle empty tree case (root == null), 2 points
- destination found base case, 3 points
- checks start found before destination and remember that / transmit it to recursive calls, 3 points
- check children null before looping, 1 point
- loop children, 2 points
- correctly make recursive call, evaluate result, and return if path found, return correct result as soon as possible, 6 points

Other:

- assuming binary tree, -8
- adding helper methods (question disallowed), -2 to - 5
- early return -6