CS314 Spring 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.

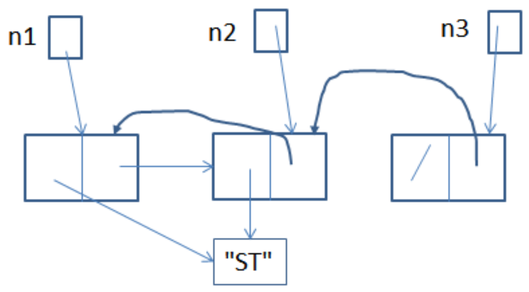
A. 8

B. 46913!13964

C. Runtime error OR exception or Stack overflow

D. 37

E. O(N^3)



F.

G. O(N^2)

H. -100

I. 22 seconds

J. quicksort (average case O(NlogN) worst case (N^2) sorted data

K. 150 seconds

L. 5 10 14 17

M. For child classes to call in order to initialize private instance variables in the abstract class. (We were very particular.)

N. 1000 seconds or 1024 seconds (any value between 1000 and 1024 ok)

O. 42 seconds

P. 160 seconds

Q. X Q G Y D H P T R K

R. X Q Y P D G H T K R

S. P Y D Q K T R H G X

T. ----------------------------→

```
-5
  \
   10
  /  \
 0    15
  \
   5
```

2. Comments. **Very** similar to a question from the quiz! Students did fairly well on this question. A lot of abstractions and classes to keep straight.

Common problems:

- not using a TreeMap to store the result.
- Making the question harder than it was. A lot of students thought they had to do the work to ensure the keys in the result were in sorted order. The TreeMap does that all on its own.
- Calling contains on a map. There is no contains method for maps. containsKey
- Assuming there is an ArrayList constructor that takes in a single value. No such constructor

```
public static Map<String, ArrayList<String>> getVisitorMap(
        Map<String, ArrayList<String>> countriesVisited) {

    TreeMap<String, ArrayList<String>> result
                                = new TreeMap<String, ArrayList<String>>();
    for (String person : countriesVisited.keySet()) {
        for (String country : countriesVisited.get(person)) {
            ArrayList<String> visitors = result.get(country);
            if (visitors == null) {
                 visitors = new ArrayList<String>();
                result.put(country, visitors);
            }
            if (!visitors.contains(person)) {
                visitors.add(person); // no need to put back. References!!
            }
        }
    }
    return result;
}
```

16 points , Criteria:

- create resulting map: 1 point
- resulting map is a TreeMap: 2 points (Lose this if HashMap or don't convert to TreeMap at end)
- loop through keys (people) correctly: 2 points
- get associated value / list for person: 1 point
- loop through list of countries: 1 point
- get people list from resulting map correctly: 1 point
- handle null case (first instance) of country correctly: 4 points (create and put)
- add person to list and only if not present: 3 points
- return correct result: 1 point

3. Comments: A simple traversal of a linked list. A little complication added with no size variable and no references to the last node. Also having to deal with possible null data. Students did fairly well

Common problems:

- not handling nulls at all
- Assuming data is null and using data.equals(other) to see of other is also null. This leads to a null pointer exception. When checking null on calling object must instead use data == null or data == other
- Implementing a recursive solution. That is O(N) space. The iterative solution is O(1) space.
- off by one errors. Starting off at first.next or stopping when temp.next == null

```
public int frequency(E tgt) {
      int result = 0;
      Node<E> temp = first;
      while (temp != null) {
            E data = temp.data;
            if (tgt != null && tgt.equals(data)) // case when tgt it not null
                  result++;
            else if (data == tgt) // handles case when tgt is null
                  result++;
            temp = temp.next; // move to next node
      }
      return result;
}
```

16 points, Criteria:

- create temp node: 1 point
- variable to count frequency: 1 point
- loop with correct condition to move through list until end: 4 points (off by one error -2)
- handle case when data and / or tgt not null: 3 points (-2 if don't use equals)
- handle case when data and / or tgt equal null: 1 point (must ensure no NPE)
- Move temp correctly: 5 points
- return correct result: 1 point

Other deductions:

- destroy some or all of list by altering first, -5
- call next on first, off by one error, -2
- assuming nodes have a hasNext method

4. Comments: Interesting problem involving queues. Similar to the merge algorithm from mergesort and the Set assignment. Overall students did very well on this question.

Common problems:

- Using logical or || in the while loop condition and not having special cases inside the loop. (We saw a number of solutions that kept going while one of the lists wasn't empty, but this required handling special cases in the loop.
- dequeening from an empty queue
- only adding one element and trying to add the other one the next time through the loop

Suggested Solution:

```
public Queue<Integer> interleave(Queue<Integer> q1, Queue<Integer> q2) {
      Queue<Integer> result = new Queue<Integer>();
      while (!q1.isEmpty() && !q2.isEmpty()) {
            int v1 = q1.dequeue();
            int v2 = q2.dequeue();
            if (v1 < v2) {
                  result.enqueue(v1);
                  result.enqueue(v2);
            } else {
                  result.enqueue(v2);
                  result.enqueue(v1);
            }
      }
      Queue<Integer> temp = q1.isEmpty() ? q2 : q1; // pick the non empty queue
      while (!temp.isEmpty()) {
            result.enqueue(temp.dequeue());
      }
      return result;
}
```

16 points, Criteria:

- correctly create resulting queue, 1 point
- correctly loop while elements in both queues, 4 points
- for a pair of elements, dequeue from each queue once, 3 points
- add min of pair then max of pair correctly, 4points
- add rest of elements from "bigger" queue correctly, (2 while loops okay), 3 points
- return correct result, 1 point

Other:

5. Comments: A relatively easy recursive problem. Very similar to exploring all the sub directories in a starting directory.

Common problems:

- not creating and adding to a local variable for the number of nodes that have the desired number of children
- treating leaf nodes (0 children) as a special case. I saw a number of solutions that would always return 0 for leaf nodes. However, `numChildrenTarget` could equal 0 itself meaning we want to count the number of leaf nodes
- Not handling case when tree is empty and n is initially null
- returning early. For example, even if the current node has the target number of children when don't return 1. We need to keep descending down the tree to look for other nodes with the target number of children. In other words, if the current node has the target number of children that is NOT a base case
- looking ahead and missing the root node
- treating children as an array instead of an ArrayList

Suggested Solution:

```
private int helper(TNode<E> n, int numChildrenTarget) {
      int result = 0;
      if (n != null) {
            if (n.children.size() == numChildrenTarget) {
                  result++;
            }
            for (int i = 0; i < n.children.size(); i++) {
                  result += helper(n.children.get(i), numChildrenTarget);
            }
      }
      return result;
}
```

16 points, Criteria:

- null check, 2 points
- check current node has correct number of children and count it, 4 points
- loop through child nodes, 3 points
- make recursive call and add result to running total, 6 points (early return -6)
- return result, 1 point

Other:

6. Comments: An example of a recursive backtracking problem that does not use a loop to make the choices because there are only two choices. Almost every reasonable solution assumed we should loop through the proper divisors as the choices. While there are valid solutions that do this, they take a much longer time to complete. Recall when implementing a recursive backtracking algorithm we normally we want to deal with **one item at a time**. In this case one proper divisor at a time. The choices for the current proper divisor are simple. Include it in the subset or don't include it in the subset.

Common problems:

- not stopping when target negative or sum > value. This is a base case, all proper divisors are positive
- looping through the proper divisors. If you start the loop at index and pass index + 1 this leads to incorrect answers because proper divisors will be used multiple times
- looping through the proper divisors, but starting at the current loop control variable + 1. This does find the right answer but after testing I saw this approach take about 20 times as long as the solution below
- returning early
- not using return value to decide to keep trying or whether we should stop

Suggested Solution:

```
public boolean isSemiperfect(int[] properDivisors, int value) {
      return semiPerfectHelper(properDivisors, 0, tgt);
}

private static boolean semiPerfectHelper(int[] properDivisors,
                                            int index, int tgt) {
    if (tgt == 0) {
        return true; // found a solution!
    } else if (tgt < 0 || index == properDivisors.length) {
        return false; // no more values OR sum too big
    } else {
         // try to reach target WITH and WITHOUT using the
        // the current proper divisor
         int newIndex = index + 1;
        int newTarget = tgt - properDivisors[index];
        return semiPerfectHelper(properDivisors, newIndex, newTarget)
                ||  semiPerfectHelper(properDivisors, newIndex, tgt);
    }
}
```

16 points, Criteria:

- call helper, 1 point
- base case, target reached, 2 points
- base case no more numbers, 2 points
- base case, target negative or current sum TOO BIG, 2 points
- recursive case, try with and without number, 8 points
    - loop starting at 0 or current index, - 6 efficiency, incorrect
    - loop starting at current loop control variable +1, -3 efficiency
- return result, 1 point