CS314 Spring 2017 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. 23

B. -11

C. O(logN) // base 2 okay

D. 15

E. O(N)

F. $O(N^2)$

G. $O(N^3)$

H. 84 seconds

I. 90 seconds

J. .004 seconds

K. 9 10 3 4 1 2

L. x / (y - z)

M. -2

N. .21 seconds

O. Because adding to the front for enqueue or removing from the front for dequeue will be O(N). (Or words to that effect. Needed to mention both or -1.)

P. 3

Q. 8 5 13 3 8 0 10

R. 8 5 3 0 10 13 8

S. 0 10 3 5 8 13 8

T. Two problems. 10 is not less than 5 but is in the left subtree of the 5 AND there are two 8's, duplicates not allowed. (Or words to that effect. Needed to mention both or -1)

EXTRA CREDIT: 2

2. Comments. A relatively simple linked list problem that required moving through two linked lists

Common Problems:

- missing the last or first node
- not comparing data correctly
- not handling null data correctly

```
public int getNumDifferences (LinkedList314<E> other) {
      Node<E> nThis = this.first;
      Node<E> nOther = other.first;
      int result = 0;
      while (nThis != null && nOther != null) {
            E dataThis = nThis.data;
            E dataOther = nOther.data;
            if (dataThis == null || dataOther == null) {
                  if (dataThis != dataOther) {
                        result++; // not both null
                  }
            } else if (!dataThis.equals(dataOther)) {
                  result++:
            }
            nThis = nThis.next;
            nOther = nOther.next;
      }
      // count remaining items in longer list
      Node<E> n = (nThis != null) ? nThis : nOther;
      while (n != null) {
            result++;
            n = n.next;
      }
      return result;
}
```

20 points , Criteria:

- 2 temp variables, 1 point
- track number of differences, 1 point
- loop until one of temps is null, 3 points
  Off By One Error (OBOE) due to n.next != null
- correctly handle case when one or both data values is null, 2 points
- correctly call equals on data elements and increment result if not equal, 3 points
  -2 for comparing nodes instead of data, -2 if use compareTo instead of equals, -2 for == instead of .equals
  -2 for counting equal values instead of different values !n1.data.equals(n2.data)
- move both references, 5 points
- move through larger list, counting number of elements left, 4 point
- return result, 1 point

Other penalties:
using disallowed methods, -6 (can be less for severity)
destroy either list, -6

3. Comments: A relatively simple stack question. Very similar to the reversing the stack question from lecture

Common problems:

- using two stacks (not allowed per description)
- not restoring stacks (early return of false)
- not stopping at first difference
- we didn't take off for this (in retrospect we should have) but some people tried to get clever and just push one of the equal items on to the temp stack. The items are equal, so why does it matter if I store one or two. The problem with this is we are storing references and if the objects are mutable putting the same reference onto two different stacks could lead to hard to find logic errors if one of the objects is altered, thus altering the "other one" although there is really only one object
- not both stacks are empty after the first difference

```
public <E> boolean stacksAreEqual(Stack314<E> s1, Stack314<E> s2) {
     Stack314<E> tempStack = new Stack314<>();
     boolean equal = true;
     while (!s1.isEmpty() && !s2.isEmpty() && equal) {
          // OR !(s1.isEmpty() || s2.isEmpty()) && equal
          equal = s1.top().equals(s2.top());
          tempStack.push(s1.pop());
          tempStack.push(s2.pop());
     }

     // make sure both stacks are empty
     equal = equal && s1.isEmpty() && s2.isEmpty();

     // Now put the stuff we took out back.
     while (!tempStack.isEmpty()) {
          s2.push(tempStack.pop());
          s1.push(tempStack.pop());
     }
     return equal;
}
```

20 points, Criteria:

- create temp stack, 1 point
- loop until one or both of stacks are empty, 4 points
- stop when first difference occurs, efficiency, 3 points
- properly remove elements from stacks and place in temp stack, 4 points
  (if equal can, just add one)
- check both stacks empty after main loop and update result correctly, 3 points
- properly put items back in original stacks (order of push's must be reversed), 5 points
  (or if only pushed one, add back to both)

Other deductions:

- using data temp data structure besides a single Stack, -6
- using two stacks instead of 1, -3
- using compareTo instead of equals, -2

4. Comments: An interesting map problem with two parts

Common problems:

- Creating an extra data structure. I chose to take off for this because the question did not allow the use of a Map constructor.
- Co-modification error. Without the ability to use another data structure if you remove from the map using, the map remove method, in the middle of a for-each loop or an iteration, a co-modification error is thrown. It was necessary to explicitly use the iterator for the keyset and remove via the iterator which does in fact remove from the map.
- Not checking if the target value is actually present
- Removing the target point itself.

Suggested Solution:

```
private static void removePointsHalfAverageDistance(Map<String, Point> m,
                                    String target) {
    Point center = m.get(target);
    if (center != null) {
        int x = center.getX();
        int y = center.getY();
        double total = 0.0;
        for (String key : m.keySet()) {
            Point p = m.get(key);
            int x2 = p.getX();
            int y2 = p.getY();
            total += Math.sqrt(Math.pow(x - x2, 2) + Math.pow(y - y2, 2));
        }
        double halfAve = total / (m.size() - 1) / 2;
        Iterator<String> it = m.keySet().iterator();
        while (it.hasNext()) {
            String key = it.next();
            Point p = m.get(key);
            int x2 = p.getX();
            int y2 = p.getY();
            double distance = Math.sqrt(Math.pow(x - x2, 2)
                    + Math.pow(y - y2, 2));
            if (distance < halfAve && !key.equals(target)) {
                it.remove();
            }
        }
    }
}
```

20 points, Criteria:

- check target present, 1 point
- loop through keys, 3 points
- determine distance between current point and target, 4 points
- determine correct average distance, 3 points
- remove key-values that are less than half the average distance, 3 points
- avoid co-modification error by using iterator remove method, 3 points
- does not remove target key-value pair, 3 points

Others: New data structure -3

5. Comments: A very interesting recursive back tracking problem, because what and when we return varies based on if the current player is the goal player or not.

Common problems:

- Incorrect base case (must check gameOver())
- Not handling the two approaches based on goal == cur and goal 1= cur
- returning early
- not using return value to determine what to do.

Suggested Solution: (Gacky with repeated code, but possibly easier to understand)

```
public boolean canWin(ConnectFourBoard b, char goal, char cur) {
	// base case
	if (b.gameOver()) {
		return b.winner() == goal;
	} else {
		char next = cur == 'b' ? 'r' : 'b';
		if (goal == cur) {
			// looking for a winning move to return true
			for (int col = 0; col < ConnectFourBoard.NUM_COL; col++) {
				if (b.columnisOpen(col)) {
					b.dropPiece(col, cur);
					boolean won = canWin(b, goal, next);
					b.pickUpTopChecker(col);
					if (won)
						return true;
				}
			}
			return false; // the goal player can't find a win
		} else {
			// looking for any non-win to prevent goal from winning
			for (int col = 0; col < ConnectFourBoard.NUM_COL; col++) {
				if (b.columnisOpen(col)) {
					b.dropPiece(col, cur);
					boolean won = canWin(b, goal, next);
					b.pickUpTopChecker(col);
					if (!won)
						return false;
				}
			}
			return true; // The goal player can win no matter what we do!!
		}
	}
}
```

20 points, Criteria:

- correctly handle base case, 4 points
- determine color of next checker correctly, 2 points
- loop through choices (columns) in recursive case, 1 point
- place and remove the next checker correctly, 2 points
- make correct recursive call, and use result correctly (not returning early, but returning if desired answer found), 5 points
- handle the two cases of goal == cur and goal != cur correctly, 4 points
- return correctly if none of the choices in the recursive case were desired answer, 2 points