CS314 Fall 2016 Final Exam 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.

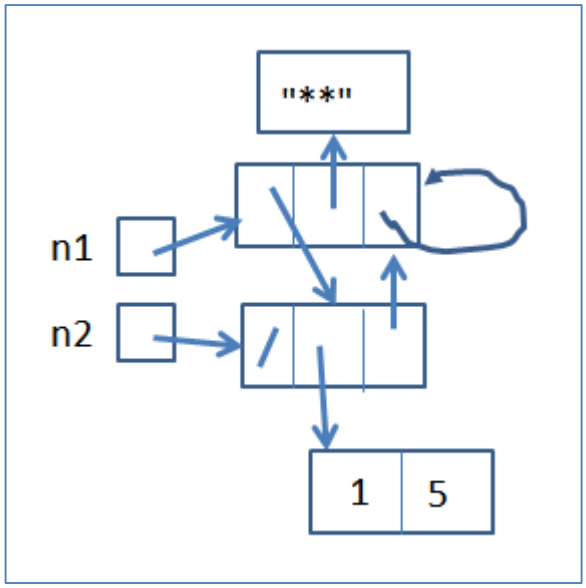
NN - Not necessary. Code is unneeded. Generally no points off

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. 12 seconds

B. Runtime error or Stack Overflow Exception

C. 21

D. 8 seconds

E. 20 seconds

F. 320 seconds

G. 44 seconds

H. ---------->

I. -5 5 8 10 12 7

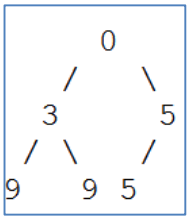
J. 90 seconds

K. Height is too close to the perfect height. Actual height given random data was on the order of 2 x perfect (or minimum height) OR WORDS TO THAT EFFECT

L. 20 seconds

M. O(N) (if have to resize)

N. ---------->



O. All written to the file will have roughly equal frequencies. Each new code will be the same length as the original values and with the header information the file will be larger than the original

P. No, path rule violated (1 path 2 black nodes, all others 3)

Q. 21

R. Easier to write algorithm or implementation easier to understand

S. sparse

T. **DP solution will often run faster / take less time.**

2. Comments.

```
public int removeEveryNth(int n) {
    int numRemoved = 0;
    if (first != null) {
        final int MOVES = n - 1;
        Node<E> temp = first;
        while (temp != null) {
            // move temp down n - 1 times if possible
            int count = 0;
            while (count < MOVES && temp != null) {
                temp = temp.next;
                count++;
            }
            if (temp != null) { // if temp IS null, not enough nodes
                if (temp.next != null) {
                    temp.next = temp.next.next;
                    numRemoved++;
                }
                else
                    // on last node, move temp to stop
                    temp = null;
            }
        }
        // now move first and remove first node
        first = first.next;
        numRemoved++;
    }
    return numRemoved;
}
```

16 points , Criteria:

- handle empty case / first = null, 2 point
- create temp node variable, 1 point
- move temp to node before spot to remove (or scout to node to remove or 1 past), 6 points
  (THIS INCLUDES moving temp correctly, - 4 if don't move correctly)
- correctly remove node by updated temp.next, 3 points
- move first correctly, 2 points
- correctly track and return number removed: 2 points

Other penalties:

- destroy whole list: -8
- other large data structure: -8 (array, other list)
- use other methods unless write: size - 5, remove -6, getNode - 8
- possible null pointer exceptions, -3
- doesn't handel case when n = 1, clears list, -4

3.

```
public IntTree combine(IntTree other) {
```

```
        IntTree result = new IntTree();
        result.root = help(root, other.root);
        return result;
    }

    private BNode help(BNode thisNode, BNode otherNode) {
        if (thisNode == null && otherNode == null) {
            return null;
        }
        BNode result = new BNode();
        if (otherNode == null) {
            result.value = 1;
            result.left = help(thisNode.left, null);
            result.right = help(thisNode.right, null);
        } else if (thisNode == null) {
            result.value = 2;
            result.left = help(null, otherNode.left);
            result.right = help(null, otherNode.right);
        } else {
            result.value = 3;
            result.left = help(thisNode.left, otherNode.left);
            result.right = help(thisNode.right, otherNode.right);
        }
        return result;
    }
```

16 points, Criteria:

- create helper with correct params, 2 points
- create result and set root to result of helper, 1 point
- base case both nodes null, 3 points
- recursive case, create new node, 2 points
- both not null, 3 points
- this not null case, 2 points
- other not null case, 2 points
- return new node, 1 point

Other deductions:

- destroy / alter either original tree: -6
- value parameter trap, -6
- no new nodes created, -8

4.
Suggested Solution:

```
public HuffmanTree(Map<Integer, String> codes) {
      root = new TreeNode();
      for (int leafValue : codes.keySet()) {
            String code = codes.get(leafValue);
            TreeNode temp = root;
            for (int i = 0; i < codes.length(); i++) {
                  if (codes.charAt(i) == '0') {
                        if (temp.left == null) {
                              temp.left = new TreeNode();
                        }
                        temp = temp.left;
                  } else {
                        // must be a 1, we need to go right
                        if (temp.right == null) {
                              temp.right = new TreeNode();
                        }
                        temp = temp.right;
                  }
            }
            // temp now on new leaf node
            temp.value = leafValue;
      }
}
```

16 points, Criteria:

- create root, 2 points
- loop through keys and get String value, 3 points
- loop through chars of String, 2 points
- create new nodes if necessary, 4 points
- move down tree correctly, 3 points
- assign value to correct node, 2 points

Other:

- creating unnecessary substrings, -3
- using char array, efficiency, -3
- delete preexisting nodes / always create nodes when moving, -5
- not resetting current to root, -2

5. Suggested Solution:

```
private double help(Vertex current, String goal) {
    if (current.name.equals(goal)) {
        // Base case! No cost to get to myself! Done!
        return 0.0;
    } else if (current.scratch != 0) {
        // already been here, can't reuse
        return -1;
    } else {
        current.scratch = 1; // Mark we are here so we don't reuse
        // recursive case, try all current Vertex edges
        double max = -1.0;
        for (Edge e : current.adjacent) {
            Vertex next = e.dest;
            double costForNext = help(next, goal);
            double totalCost = e.cost + costForNext;
            if (costForNext != -1 && totalCost > max) {
                max = totalCost;
            }
        }
        current.scratch = 0; // undo so we can try other paths
        return max;
    }
}
```

16 points, Criteria:

- base case reached goal, 2 points
- base case, already been to node, 2 points
- recursive case:
- mark and unmark scratch for node, 2 points
- track max cost from this node (parameter won't work in most cases), 5 points
- loop through edges, 2 points
- make correct recursive call, 2points
- return best, 1point

Other:

- early return -5
- create array, -3
- visit start vertex again, -2

6. Comments:

Suggested Solution:

```
public static int coinsDP(int[][] coins) {
    int[][] best = new int[coins.length][coins[0].length];
    best[0][0] = coins[0][0]; // fill in start
    // Fill in top row and left column.
    // Only one way to reach these cells.
    for (int c = 1; c < best[0].length; c++) {
        best[0][c] = coins[0][c] + best[0][c - 1];
    }
    for (int r = 1; r < best.length; r++) {
        best[r][0] = coins[r][0] + best[r - 1][0];
    }

    for (int r = 1; r < best.length; r++) {
        for (int c = 1; c < best[0].length; c++) {
            // Only two ways to get to current cell,
            // from row above or from column to the left.
            int coinsFromAbove = best[r - 1][c];
            int coinsFromLeft = best[r][c - 1];
            // Best result for this cell is coins in this cell
            // plus max of cell above or to left.
            best[r][c] = coins[r][c]
                    + Math.max(coinsFromAbove, coinsFromLeft);
        }
    }
    return best[best.length - 1][best[0].length - 1];
}
```

16 points, Criteria:

- create aux 2d array of correct size: 1 point
- fill in [0][0] or aux correctly, 3 points
- fill in top row and left most column with cumulative sum
  OR check in primary nested loop if cell exists: 4 points
- primary nested loop moving correctly left to right top to bottom OR top to bottom left to right, 2 points
- for each cell, add coins from original 2d array and max of above and left, 4 points
- return correct result, bottom, right cell, 2 points