CS314 Spring 2017 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. 1
   B. 32
   C. Adjacency Matrix. Simply need to check all N vertices. With adjacency list must check each Vertex's list of edges.
   D. Because most graphs are sparse and an adjacency matrix wastes a lot of space given sparse graphs.

   E.  

   F.  

   G. O(N) when we have to do a linear search and there are no or few null values left in the hash table. (or words to that affect)
   H. Each distinct path from a node to each null link must contain the same number of black nodes
   I. 5 bits
   J. .00048 seconds
   K. 52
   L. 80 seconds. (Radix sort O(NlogMax) and logMax hasn't changed.)
   M. 2 seconds (best case with min heap implementation)
   N. O(N^2logN)
   O. O(NlogN)
   P. 14 (D to C to I to H to G)
Q. 37
R. 30 14 6
S. 30 to 50 inclusive
T. 42 / 5 seconds or 8.4 seconds
2. Comments.

```java
public int insertAndRemove(E insertAfter, E insertVal, E remove) {
    // remove targets at front;
    while (first != null && first.data.equals(remove))
        first = first.next;
    if (first != null)
        first.prev = null;
    int numElements = 0;
    Node<E> temp = first;
    while (temp != null) {
        numElements++;
        if (temp.data.equals(insertAfter)) {
            Node<E> newNode = new Node<>();
            newNode.data = insertVal;
            newNode.prev = temp;
            newNode.next = temp.next;
            if (temp.next != null)
                temp.next.prev = newNode;
            temp.next = newNode;
        } else if (temp.data.equals(remove)) {
            numElements--;
            temp.prev.next = temp.next;
            if (temp.next != null)
                temp.next.prev = temp.prev;
        }
        temp = temp.next;
    }
    return numElements;
}
```

16 points, Criteria:
- special case for removing elements from front and adjusting front (can be special case in general loop), 3 points
- Loop until end of list, 2 points
- if target add, add correctly, 3 points
- remove correctly, 3 points
- move temp reference down in list correctly, 3 points
- count and return number of elements in list, 2 points

Other penalties:
- == instead of .equals -4 once
- disallowed methods, add (-6), remove(-6)
- worse than O(N), -6
- doesn’t handle last case / nulls -2
- Comparing Nodes not data in Nodes - 5
3. Suggested Solution:

```java
public int makePerfectToGivenDepth(int depth, int newVal) {
    int oldSize = size;
    root = perfectHelp(root, depth, 0, newVal);
    return size - oldSize;
}

private BNode perfectHelp(BNode n, int targetDepth, int currentDepth, int newVal) {
    if (currentDepth <= targetDepth) {
        if (n == null) {
            // need a node at this level
            n = new BNode(newVal);
            size++;
        }
        int newDepth = currentDepth + 1;
        n.left = perfectHelp(n.left, targetDepth, newDepth, newVal);
        n.right = perfectHelp(n.right, targetDepth, newDepth, newVal);
    }
    // else too deep, nothing to do
    return n;
}
```

16 points, Criteria:
- create helper with correct params, 2 points
- return correct value of nodes added, 2 points
- base case for beyond target depth, 2 points
- If null node, create new node, 3 points
- if null node increment size instance var, 2 points
- recursive calls with new depth and child nodes correct, 3 points

Other deductions:
- doesn't return / set references or use look ahead correctly, -6
public static void updateRefCount(int objectRef,
                        Map<Integer, int[]> refMap) {
    if (objectRef > 0) {
        // adding a reference
        int[] count = refMap.get(objectRef);
        if (count == null) {
            // first reference
            refMap.put(objectRef, new int[]{1});
        } else {
            // increment count
            count[0]++; // no need to put back due to reference
        }
    } else {
        // decrementing a reference
        objectRef *= -1;
        int[] count = refMap.get(objectRef);
        if (count != null) {
            count[0]--;
            if (count[0] == 0) {
                // Uh-oh. Need to remove and decrement things I refer to
                refMap.remove(objectRef);
                for (int i = 1; i < count.length; i++) {
                    updateRefCount(-count[i], refMap);
                }
            }
        }
    }
}

16 points, Criteria:
- determine increment or decrement case correctly, 1 point
- increment reference case
  - check if key present correctly, 1 point
  - if key not present, put key and new array with 1 element and value of 1, 2 points
  - if key present, increment ref count, element 1 in array, no need to put again due to refs, 1 point
- decrement reference case
  - flip int back to positive, 1 point
  - check key present, 1 point
  - get array of ints correctly, 1 point
  - decrement first element of array, 1 point (no need to put back due to refs)
  - if count is 0, remove key from map, 2 points
  - loop correctly through references, 2 points (must start at 1)
  - correctly decrementing references, 3 points (iterative or recursive)

Other:
- any iteration through all of keys or values, -6
- using contains, -2
- incrementing all ref counts, -3
public double dfsPathCost(String start, String dest) {
    if (vertices.get(start) == null || vertices.get(dest) == null) {
        return -1;
    } else {
        clearAll();
        return dfsHelp(vertices.get(start), dest);
    }
}

private double dfsHelp(Vertex current, String dest) {
    if (current.name.equals(dest)) {
        return 0;
    } else {
        if (current.scratch == 1) {
            return -1; // going in a circle
        }
        // mark the node so we don't go in circles
        current.scratch = 1;
        for (Edge e : current.adjacent) {
            // try the edge
            double result = dfsHelp(e.dest, dest);
            if (result != -1) {
                // found a solution, add my cost
                return result + e.cost;
                e.dest.prev = current;
            }
            // loop just tries the next one
            // nothing to undo
        }
        return -1; // never found a solution
    }
}

16 points, Criteria:
- return -1 if either vertex not present, 1 point
- call to setPrev, 1 point
- call recursive helper method, 1 point
- success base case in recursive helper (can be in loop), 2 points
- failure base case, scratch != 0, going in a circle return -1, 3 points
- mark scratch to prevent cycles, 1 point
- loop through edges of current, 2 points
- recursive call with new current, 2 points
- check result of recursive call, set prev, return cost of edge if != -1, 2 points
- return false after loop, 1 point

Other:
- not recursive backtracking, -6, other data structures, -6, not calling clearAll - 2
- early return -6, not stopping on success, -6,
6. Comments:

QUESTION: Compression if average run length > 9 (or words to that affect)

```java
public static int bitsSaved(BitInputStream in) {
    int originalBits = 0;
    int compressedBits = 0;
    int previousBit = in.readBits(1);
    int count = 1;
    while (previousBit != -1) {
        originalBits++;
        int currentBit = in.readBits(1);
        if (currentBit != previousBit) {
            // End of a run.
            compressedBits += bitsNeeded(count);
            count = 1;
            previousBit = currentBit;
        } else {
            count++;
        }
    }
    // take care of the last run
    compressedBits += bitsNeeded(count);
    return originalBits - compressedBits;
}

private static int bitsNeeded(int runLength) {
    // How many chunks to write?
    int chunks = runLength / 256;
    int result = 9 * chunks;
    if (runLength % 256 > 0) {
        result += 9;
    }
    return result;
}
```

16 points, Criteria:
- answer question correctly, 2 point
- variables for original bits, compressed bits, current count: 1 point
- loop until read returns -1, 1 point
- read and compare current bit to previous bit read, 2 point
- continue run, increment counter, 1 point
- end of run (or 256 bits), calculate compressed bits correctly, 4 points
- change previous bit at end of run, 1 point
- get last run of bits, 3 points
- return correct answer, 1 point

Other:
- any other classes or data structures, -2 to -8 depending on severity
- read a byte at a time, -2, missing mod -2, uses Strings (efficiency) -6, using nonexistent iterator - 5