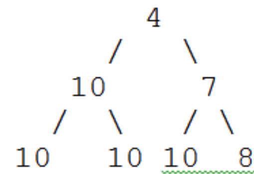
CS314 Spring 2016 Final Exam Solution and Grading Criteria.
Grading acronyms:
AIOBE - Array Index out of Bounds Exception may occur
BOD - Benefit of the Doubt. Not certain code works, but, can't prove otherwise
Gacky or Gack - Code very hard to understand even though it works. (Solution is not elegant.)
LE - Logic error in code.
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. 16 seconds **X. --->**



B. 8

C. 8

D. 25 false

E. 44

F. 0.42 seconds

G. O(N)

H. O(N)

I. Use a linked list instead of an array based list. (Or words to that effect. If linked list used, should use linear search instead of binary)

J. Make the end of the list the front of the priority queue. (or words to that effect.)

K. in-order traversal must yield ascending order, no duplicates

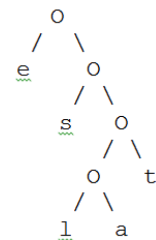
L. 13 0 -5 5 12 11

M. 4

N. 60 seconds (Method is O(N^2), String concat is O(N) by itself.)

O. No, it is not a binary search tree. (17 is misplaced.)

P. --→



Q. 44 seconds

R. Because most graphs are sparse and the adjacnecy matrix result in a lot of wasted space. (or words to that effect.)

S. arrays **T. 80** **U. No** **V. 12 seconds**

W. 10,000 - 10,240 seconds **X. See above** **Y. O(N^3)**

2. Comments. A simple exercise using stacks.

Common problems:

- not dealing with empty case or case with one element
- not restoring stack

```java
public static <E extends Comparable<E>> boolean
                                    isDescending(Stack<E> st) {
    Stack<E> temp = new Stack<E>();
    boolean descending = true;
    if (!st.isEmpty()) {
        // put top element on temp
        temp.push(st.pop());

        while (descending && !st.isEmpty())
            E current = st.pop();
            // is current element less than or equal to previous?
            descending = current.compareTo(temp.top()) <= 0;
            temp.push(current);
        }

        // restore st
        while (!temp.isEmpty()) {
            st.push(temp.pop());
        }
    }
    return descending;
}
```

15 points , Criteria:

- handle empty case, 1 point
- create temp stack, 1 point
- handle first value, 1 point
- loop present and correct, 2 points
- ensure current element less than or equal to previous in loop, 3 points
- stop when answer know, 3 points (efficiency)
- restore stack correctly, 3 points
- return result, 1 point

Other penalties:

- using compareTo == -1, -2
- lose a value, off by one on restore, -2
- calling top() or pop() on empty stack, -3

3.

```
public int remove(int value) {
    int numRemoved = 0;
    Node lead = first;
    Node trailer = null;
    while (lead != null && lead.value <= value) {
        if (lead.value == value) {
            numRemoved++;
        }
        if (numRemoved == 0) {
            trailer = lead;
        }
        lead = lead.next;
    }
    if (numRemoved != 0) {
        if (trailer == null) {
            first = lead;
        } else {
            trailer.next = lead;
        }
    }
    return numRemoved;
}
```

ALTERNATE SOLUTION:

15 points, Criteria:

- handle case when list empty, 2 points
- loop through list to find initial node with value and node after last node with value, 2 points
- move through list correctly, 3 points
- count number of items removed correctly, 1 point
- handle case when first must be moved, 2 points
- remove nodes that contain value from list correctly, 2 points
- efficiency, stop when value in current node > target value, 2 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.

Suggested Solution:

```
public int numNodesLessThanSumOfAncestors() {
      return helper(root, 0);
}

private int helper(BNode current, int sumOfAncestors) {
      int result = 0;
      if (current != null) {
            // check if current node's less than sum of ancestors
            if (current.value < sumOfAncestors) {
                  result++;
            }
            int newSum = sumOfAncestors + current.value;
            result += helper(current.left, newSum);
            result += helper(current.right, newSum);
      }
      return result; // if current == null, result still 0
}
```

15 points, Criteria:

- create helper method and call it correctly with sum of 0, 2 points
- base case when current is null, 3 points
- if current not null, check current node's value with sum of ancestors,2 points
- calculate new sum, 2 points
- correct recursive calls added to result, 5 points
- return correct answer

Other:

error in calculation -6

use of array or O(N) space -5

5. Suggested Solution:

```
private int updateIndegree() {
    // reset all scratch variables to 0
    for (String name : vertices.keySet()) {
        vertices.get(name).scratch = 0;
    }

    int max = 0;

    for (String name : vertices.keySet()) {
        Vertex currentVertex = vertices.get(name);
        List<Edge> currentEdges = currentVertex.adjacent;
        for (Edge e : currentEdges) {
            Vertex destination = e.dest;
            destination.scratch++;
            if (destination.scratch > max) {
                max = destination.scratch;
            }
        }
    }

    return max;
}
```

15 points, Criteria:

- reset all scratch variables to 0 correctly, 3 points
- loop through each vertex again, 2 points
- loop through current vertices edges correctly, 3 points
- increment Edge's destination correctly, 3 points
- check if largest in degree and update max correctly, 3 points
- return result, 1 point

Other:

extra data structure / use of more space -3

6. Comments: Since the algorithm was given, grading was quite strict.

Suggested Solution:

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

    public ArrayList<String> getToposort() {
        updateIndegree();
        ArrayList<String> result = new ArrayList<String>();
        Queue<Vertex> indegreeZeroVertices = new LinkedList<Vertex>();

        // add all vertices with indegree of 0 to initial queue
        for (String name : vertices.keySet()) {
            Vertex currentVertex = vertices.get(name);
            if (currentVertex.scratch == 0) {
                indegreeZeroVertices.add(currentVertex);
            }
        }

        while (!indegreeZeroVertices.isEmpty()) {
            Vertex currentVertex = indegreeZeroVertices.remove();
            result.add(currentVertex.name);
            for (Edge e : currentVertex.adjacent) {
                e.dest.scratch--;
                if (e.dest.scratch == 0) {
                    indegreeZeroVertices.add(e.dest);
                }
            }
        }
        return result;
    }
```

15 points, Criteria:

- create result, 1 point
- create queue, 1 point
- add all vertices with indegree of 0 to queue, 3 points
- keep processing while queue is not empty, 2 points
- add result, 2 points
- loop through edges of current vertex, 1 points
- update scratch variable correctly, 2 points
- add if new indegree is 0, 2 points
- return correct result, 1 point