CS314 Spring 2015 Final 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.)
GCE - Gross Conceptual Error. Did not answer the question asked or showed fundamental misunderstanding
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. **32 seconds**

B. **50,000 seconds (52,428.8 accepted but not required!)**

C. **7 13 0 42**

D. **15**

E. **sparse**

F. **180 250**

G. **M 180 M 150**

H. **[10, 20][10, 20]**

I. **7 2 -3 5 6 9**

J. **O(N)**

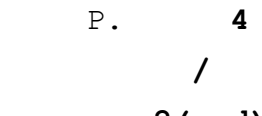
K. **$O(N^2)$**

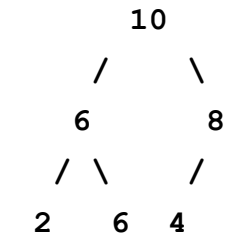
L. **$O(N^2)$**

M. **-3 2 5 6**

N. **45 seconds**

O. **21 seconds**

P.



Q.



Q. **(Heap shown above and to the right)**

R. **Less time to complete because HashSet add is O(1) while TreeSet add is O(logN). (if order not specified -1)**

S. **0 1 1 0 1 0 0 0**

T. **120**

2. Comments. A good linked list question. Similar to method from linked list assignment, but this was with a singly linked list

Common problems:

- not handling case when list empty correctly
- not updating size instance variable
- poor efficiency
- not handling case when first must refer to different node (first node in range being removed)

Suggested Solution:

```
public void removeRange(int start, int stop) {
    if(start < stop) {
        Node<E> tempBeforeStart = first;
        for(int i = 1; i < start; i++)
            tempBeforeStart = tempBeforeStart.getNext();
        Node<E> tempAtStop = tempBeforeStart;
        int numMovesForStop = stop - start;
        if(start != 0)
            numMovesForStop++;
        for(int i = 0; i < numMovesForStop; i++)
            tempAtStop = tempAtStop.getNext();
        if(start == 0)
            first = tempAtStop;
        else
            tempBeforeStart.setNext(tempAtStop);
        size -= (stop - start);
    }
    // else start == stop, nothing to do
}
```

20 points , Criteria:

- do nothing when start == stop, 1 point
- get node before start position, must move correctly, 5 points
- get node at stop position, must move correctly, 5 points
- handle case when start is 0, removing first node, moving one less time, 3 points
- correctly change next reference of node before start position (general case), 3 points
- update size correctly, 3 points

Other deductions:

-2 not handling empty case correctly
-1 double traversal

3. Comments: A very interesting problem. Not much code required, but many students failed to check all the nodes of each subtree. Solutions expected to be $O(N^2)$ given the restrictions.

Common problems:

- not using return values from methods
- not checking all elements in subtree for repeat of root of subtree (in a helper method)
- not handling empty tree correctly
- not checking roots of ALL subtrees
- solution immediately fails because root compared against itself

Suggested Solution:

```
    public boolean rootsNotInSubtrees() {
        return rootsNotInSubtree(root);
    }

    private boolean rootsNotInSubtree(BinaryNode<Integer> n) {
        if(n == null)
            return true;
        else
            return valueNotInSubtree(n.getData(), n.getLeft())
                && valueNotInSubtree(n.getData(), n.getRight())
                && rootsNotInSubtree(n.getLeft())
                && rootsNotInSubtree(n.getRight());
    }

    private boolean valueNotInSubtree(Integer data, BinaryNode<Integer> n)
{
        if(n == null)
            return true;
        else if(data == n.getData())
            return false;
        else
            return valueNotInSubtree(data, n.getLeft())
                    && valueNotInSubtree(data, n.getRight());
    }
```

20 points , Criteria:

- kickoff recursion is given method, 1 point
- return correct answer, 1 point
- create method that checks roots of ALL subtrees and calls second helper to ensure root of subtree not repeated in any descendant node of that subtree, 9 points (partial credit possible)
- create method that checks subtree to ensure a give value is not present, 9 points

other:

not using get methods: -1

4A. Comments: Solution was VERY simply even though it was used recursive backtracking

Common problems:

- not accessing current.adjacent correctly

```
private void addConnectedVertices(Set<Vertex> verts, Vertex current) {
    if(!verts.contains(current)) {
        verts.add(current);
        for(Edge e : current.adjacent)
            addConnectedVertices(verts, e.dest);
    }
}
```

10 points, Criteria:

- base case, when vertex already present, do nothing, 2 points
- if vertex not present, added to set, 2 points
- loop through edges, 3 points
- make recursive call with destination of edge correctly, 3 points

4B. Comments: Again, fairly simple problem. The real difficulty was understanding the abstraction of the graph, but if you completed the graph assignment, that was easy.

Common problems:

- assuming scratch variables in vertices set to a given value at the start. Okay to use scratch, but had to write code to ensure scratch was set to desired value
- Creating multiple sets. Restriction was to create a single set.
- Many students had a logic error where the method simply counted the number of vertices.

```
public int getNumSubGraphs() {
    Set<Vertex> verts = new HashSet<Vertex>();
    int numSubgraphs = 0;
    for(Vertex v : vertices.values()) {
        if(!verts.contains(v)) {
            numSubgraphs++;
            addConnectedVertices(verts, v);
        }
    }
    assert (verts.size() == vertices.size()); // NN
    return numSubgraphs;
}
```

10 points, Criteria:

- create single HashSet, 1 point
- correctly count number of subgraphs with local variable, 2 points
- loop through all keys or values, 2 points
- correct check for vertex not in set, incrementing number of subgraphs, and making call to recursive helper, 4 points
- return correct value, 1 point

5. Comments: A good problem dealing with a new data structure. Should have been easy if you came to class the dat we covered heaps.

Common problems:

- not stopping on correct element (deepest in tree)
- not stopping bubble up of new element when it is the root. (A lot of off by one errors as well)
- not starting from the back of the heap to try and find the deepest the fastest
- changing multiple instances of elements instead of just the deepest.

Suggested Solution:

```
public void decreaseKey(int elements, int amount) {
      int pos = size;
      while(pos > 0 && con[pos] != elements)
          pos--;
      if(pos != 0) {
          con[pos] -= amount;
          int newValue = con[pos];

          while(pos > 1 && newValue < con[pos / 2]) {
              con[pos] = con[pos / 2];
              pos /= 2;
          }
          con[pos] = newValue;
      }
       }
```

20 points, Criteria:

- find correct key to alter, 6 points
- efficiency of finding key, 2 points
- alter key correctly, 2 points
- move key up as necessary, 4 points
- stopping condition for root of heap correct, 3 points (-2 for oboe)
- stopping condition for parent less than or equal to new value correct, 3 points