NUMBER 1, SHORT ANSWER:

A. $3N^3 + 4N^2 + 4N + 4$ **(+/- 1 on each term)**

B. **9 7 3 -80 8 4 2**

C. **15**

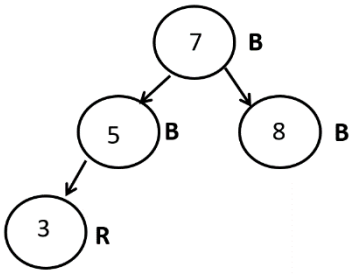
D. **40 seconds (method is $O(N^3)$)**

E. **5 3 0 2 4 9 6**

F. **{1=1, 2=3, 3=4, 4=5}**

G. **O P C M N D**

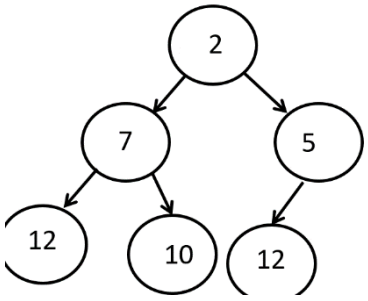
H. **1 5 5 8 (note, the size logic error)**



I. **3**

J. **---------->**

K. **If the Graph is dense. (Number of actual edge close to the maximum number of edges.) Or words to that affect.**



L. **9 (C -> B -> A -> G -> H -> F)**

M. **16**

N. **48 seconds (method is $O(N^4)$)**

O. **---------->**

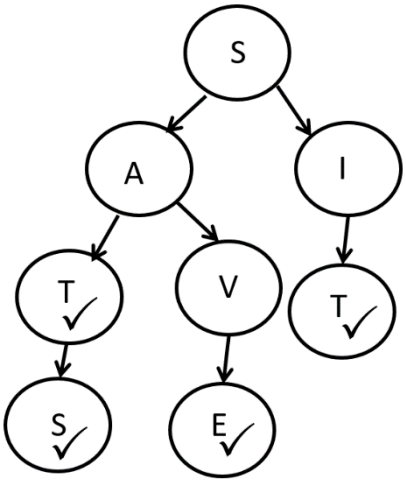
P. **PACK PI**

Q. **2 seconds (method is O(N))**

R. **The recursive backtracking solution repeatedly calculated the answer for certain sub problems a large number of times. (Or words to that affect.)**

S. **---------->**

T. **3**



2. Suggested Solution

```
        private double help(Vertex cur, String dest, double minEdgeSoFar) {
         if (cur.name.equals(dest)) {
             return minEdgeSoFar;
         } else if (cur.scratch == 1) {
             return INFINITY;
         } else {
             cur.scratch = 1;
             // min edge for path through this node
             double myMaxMin = 0;
             for (Edge e : cur.adjacent) {
                 double temp = minEdgeSoFar;
                 if (e.cost < minEdgeSoFar) {
                     temp = e.cost;
                 }
                 double minThisPath = help(e.dest, dest, temp);
                 if (minThisPath != INFINITY && minThisPath > myMaxMin) {
                     myMaxMin = minThisPath;
                 }
             }
             // undo so we can try other paths through this vertex
             cur.scratch = 0;
             return myMaxMin == 0 ? INFINITY : myMaxMin;
         }
    }
```

16 points, Criteria:

- base case destination, 2 points
- base case, been to this node before, return INFINITY, 3 points
- recursive case, set scratch to something other than 0, 1 point
- track largest min for this path, 1 point
- if current edge smaller than smallest so far, replace it, 3 points
- correct recursive call, 1 point
- if return value greater than local max, replace it, 2 points
- undo scratch, 2 points
- return, INFINITY or other flag value if no path found, 1 point

Other penalties:

- altering graph, -6 (except scratch)
- early return, -7
-

- 3. Suggested Solution:

```
public int placeBetween(E first, E second, E val) {
     Node<E> temp = header.next;
     int count = 0;
     // traverse until we are on the last node with data
     while (temp.next != header) {
          if (temp.data.equals(first) && temp.next.data.equals(second) {
               // insert in between
               count++;
               Node<E> n = new Node<>();
               n.data = val;
               n.prev = temp;
               n.next = temp.next;
               temp.next.prev = n;
               temp.next = n;
               temp = temp.next; // so we skip over the new node
          }
          temp = temp.next;
     }
     return count;
}
```

16 points, Criteria:

- var to track num added and incremented correctly, 1 point
- temp node var, initialized correctly, 1 point
- loop until next node is header node, 3 points
- correctly check current node's data is first and next node's data is second, 2 points
- correctly create and insert new node between first and second, 3 points (partial credit possible)
- skip past new node in case val is same as first (or infinite loop), 2 points
- advance temp correctly, 3 points
- return correct result: 1 point

Other deductions:

- call any Node constructor besides default, -2
- worse than O(N), -5
- infinite loop other than failure to skip past new value case, -5
- attempt to alter what node header refers to, -4
- use any methods besides equals, -2 to -6 depending on severity
- OBOE, -2
- assume header is first node with data, -4
- destroy all or most of list, -6
- NPE -4

4. Suggested solution:

```
public int numDifferences(BinaryTree other) {
      return countDiffs(this.root, other.root);
}

private int countDiffs(BNode thisN, BNode otherN) {
      // base case, both null, empty tree no differences
      if (thisN == null && otherN == null) {
            return 0;
      else {
            // one or both of the nodes isn't null
            if (thisN != null && otherN != null) {
                  // same node structure, check descendants
                  return countDiffs(thisN.left, otherN.left) +
                         countDiffs(thisN.right, otherN.right);
            } else if (thisN == null) {
                  // otherN not null, one difference, plus below
                  return 1 + countDiffs(null, otherN.left) +
                             countDiffs(null, otherN.right);
            } else {
                  // thisN not null
                  return 1 + countDiffs(thisN.left, null) +
                             countDiffs(thisN.right, null);
            }
      }
```

16 points, Criteria:

- create helper and pass root nodes of trees, 2 points
- base case, both nodes null, return 0, 3 points
- recursive case check both nodes not null, 2 points
- recursive case, both nodes not null, return 0 + correct recursive calls, 3 points
- handle case when this node is null and other isn't, return 1 + correct recursive calls, 3 points
- handle case when other node is null and other isn't, return 1 + correct recursive calls, 3 points
- Note, it is possible with check which reference is null to combine the last two cases into, full credit if done correctly.

Other:

- Null Pointer Exception, -4
- int parameter for count, over count, -4
- comparing data in nodes, -4
- early return, -5
- use any methods, -2 to -6 depending on severity
- early return, -5
- using array, -4 (even length 1)
- return null -3

5. Suggested Solution:

```
public void decode(BitInputStream bis, BitOutputStream bos) {
      int numBits = 0;
      int encodeVal = 0;
      boolean goOn = true;
      while (goOn) {
            int bit = bis.readBits(1);
            numBits++;
            encodeVal = encodeVal * 2 + bit;
            // now check to see if this matches a HuffCode
            for (int i = 0; i < codes.length; i++) {
                  HuffCode hc = codes[i];
                  if (numBits == hc.numBits && encodeVal == hc.encodeVal) {
                        // A-HA!! We found it
                        // short circuit loop
                        i = codes.length; // GACK
                        numBits = 0;
                        encodeVal = 0;
                        if (hc.decodeVal == PEOF) {
                              goOn = false; // done!!
                        } else {
                              bos.writeBits(BPW, hc.decodeval);
                        }
                  }
            }
      }
}
```

16 points, Criteria:

- variables to track current numBits and current decodeValue, 1 point
- loop until PEOF appears, 2 points
- read current bit and update numBits this chunk / code, 1 point
- correctly update current encode value, 3 points
- loop through codes array, 1 point
- correctly check if current values match current HuffCode, 1 point
- reset current numBits and current decodeValue, 2 points
- correctly write out to output file, 2 points
- checking PEOF to stop outer loop, 1 point
- short circuit inner loop when code found, 2 points

Other:

- no calculation for encode -5
- using Strings -4
- not reading a single bit at a time -6
- incorrect attempts to convert to / from base 2 -3
- reading 8 bits at a time - 8 to -14

6. Comments:

```
private void resize() {
     E[] temp = (E[]) (new Object[con.length * 2]);
     for (int i = 0; i < con.length; i++) {
          if (con[i] != null) {
               // rehash and place in temp
               E val = con[i];
               int index = val.hashCode();
               index %= temp.length;
               index = Math.abs(index);
               // we know we will have a spot
               while (temp[index] != null) {
                    index++;
                    if (index == temp.length) {
                         index = 0;
                    }
               }
               temp[index] = val;
          } // end of if
     } // end of for
     con = temp;
}
```

16 points, Criteria:

- create new array, double length of old con, (not size), okay if don't cast, 1 point
- loop through all elements of old con correctly, 2 points
- correctly check if current element not null, 2 points
- call hashCode, 1 point
- remainder by temp.length, 1 point
- Math.abs to ensure positive, 1 point
- find next open spot based on index (!= null check), 3 points
- wrap correctly, 3 points
- assign con to new array, 2 points

Other:

- NPE, -4
- AIOBE, -4
- using new as identifier , -1
- not including size of new array, new Object[], -2
- alter size -2
- call resize -5
- add duplicates -4