This test has 16 numbered page (there's a three-page insert at the end that isn’t numbered), be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper import statements. Assume that all libraries and packages we’ve discussed are imported in any code you write.

Unless indicated otherwise, the TreeNode class for this test is on the left. Some common recurrences and their solutions are on the right.

```java
public static class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String val, TreeNode lptr, TreeNode rptr) {
        info = val;
        left = lptr;
        right = rptr;
    }
}
```

<table>
<thead>
<tr>
<th>label</th>
<th>recurrence</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T(n) = T(n/2) + O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>B</td>
<td>T(n) = T(n/2) + O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>C</td>
<td>T(n) = 2T(n/2) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>D</td>
<td>T(n) = 2T(n/2) + O(n)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>E</td>
<td>T(n) = T(n-1) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>F</td>
<td>T(n) = T(n-1) + O(n)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>G</td>
<td>T(n) = 2T(n-1) + O(1)</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>
PROBLEM 1:  (Trees (24 points))

The tree below is a search tree, thus its inorder traversal is in alphabetical order:

        platypus
       /    
    echidna    wallaby
   /     
  dingo   koala
     /   
    quokka sawfish

Part A (4 points)
In a postorder traversal of this tree the last node visited/printed is platypus. What is the postorder traversal of the tree shown?

Part B (4 points)
A student makes a mistake in writing code to create a copy of a binary tree. The code with the mistake is shown below. To be correct the recursive calls to copy should be swapped, e.g., the call with root.left should be first.

    public TreeNode copy(TreeNode root){
        if (root == null) return null;
        return new TreeNode(root.info, copy(root.right), copy(root.left));
    }

Draw the tree that is returned by the incorrect copy code shown. The root node of the returned tree is platypus.
**Part C (4 points)**
On the previous page add the nodes *kangaroo* and *prawn* as leaves to the tree at the beginning of the problem so that it remains a search tree.

**Part D (4 points)**
The preorder traversal of a different search tree containing the same seven values is given below. The tree for which this is the preorder traversal is also a search tree with a different shape than the tree at the beginning of this problem.

koala dingo echidna quokka platypus wallaby sawfish

Draw the search tree for which this is the preorder traversal.

**Part E (extra credit, not required, 2 points)**
Draw a picture of one of the animals in the search tree.
Part F (2 points)
Which node in the tree at the beginning of this problem is not height-balanced? Why?

Part G (6 points)
Suppose a field TreeNode parent is added to the TreeNode class so that each node stores a pointer to its parent in the tree in addition to pointers to its left and right children. The root has a null parent, all other nodes have pointers to non-null parents.

Write the method parentize that sets the correct value of each parent field in a binary tree. For example, in the tree at the beginning of the problem the parent of quokka is wallaby and the parent of echidna is platypus. The root node platypus has a null parent. The method sets parent pointers correctly.

    public void parentize(TreeNode root) {

PROBLEM 2:  (Cell Removal (12 points))

The APT Cell Removal is attached at the end of this test. The method `cellsLeft` below is part of an all-green/all-correct solution when the methods `isLeaf` and `onPathToRoot` are completed correctly. You’ll write these methods as part of this question.

```java
public class CellRemoval {

    /**
     * Returns true if and only if index represents a leaf node in the tree
     * represented in array parents, e.g., if index has no children.
     * @param parent is array of parent indexes, i.e., parent[k] is
     * the node/index of the parent of node k.
     * @param index is the node for which leaf-ness is determined
     * @return true iff index represents a leaf in the tree determined
     * by parent relationships in array parent
     */
    boolean isLeaf(int[] parent, int index){
        // part B
    }

    boolean onPathToRoot(int[] parent, int index, int dead){
        // part C
    }

    public int cellsLeft(int[] parent, int dead){
        int alive = 0;
        for(int k=0; k < parent.length; k++){
            if (isLeaf(parent,k) && !onPathToRoot(parent,k,dead)){
                alive++;
            }
        }
        return alive;
    }
}
```

Part A (2 points)
When using nodes, leaves are identified by having no children, i.e., both left and right subtrees are empty. In this problem indexes into the array identify nodes in a tree implicitly. Explain in words how to determine if an index is a leaf in this problem.
Part B (5 points)
Complete the method `isLeaf` below so that it works as specified.

```java
/**
 * Returns true if and only if index represents a leaf node in the tree
 * represented in array parents, e.g., if index has no children.
 * @param parent is array of parent indexes, i.e., parent[k] is
 * the node/index of the parent of node k.
 * @param index is the node for which leaf-ness is determined
 * @return true iff index represents a leaf in the tree determined
 * by parent relationships in array parent
 */
boolean isLeaf(int[] parent, int index){
```
Part C (5 points)
Complete the method `onPathToRoot` below so that it works as specified.

```java
/**
 * Returns true if and only if dead is a node on the path
 * from node index to the root.
 * @param parent represents the tree information by parent relationships
 * @param index is the node at which we start exploring path to root
 * @param dead is node queried: ‘‘is this on path from index to root’’
 * @return true iff path from index to root goes through dead
 */

boolean isOnPathToRoot(int[] parent, int index, int dead){

}}
PROBLEM 3: \((OgBog (18 \text{ points}))\)

Part A (6 points)
In playing regular Boggle letters cannot be re-used within a single word found on the board. The board below illustrates that re-using letters within a word can result in finding more words on a board. The word “TOMATO” can be formed by re-using letters as shown. Note that both “T” and “O” are used twice in the word shown, although it’s possible to make “TOMATO” re-using only the “T”. (It’s possible to make “TOMATO” without re-using letters, but it’s not possible to create “TOOL” shown in the diagram without re-using an “O”).

Here’s a correct version of method \texttt{cellsForWord} in class \texttt{GoodWordOnBoardFinder} for a Boggle program in which letters cannot be re-used.

```java
public List<BoardCell> cellsForWord(BoggleBoard board, String word) {
    ArrayList<BoardCell> list = new ArrayList<BoardCell>();
    for (int r = 0; r < board.size(); r++) {
        for (int c = 0; c < board.size(); c++) {
            if (findHelper(word, 0, r, c, board, list)) {
                return list;
            }
        }
    }
    list.clear();
    return list;
}
```

This method calls a private method \texttt{findHelper} to do the recursive work. The code below is correct for regular Boggle (except for words with the letter ‘Q’ which aren’t accounted for). Comment out (put //

```java
public boolean findHelper(String word, int pos, int r, int c, BoggleBoard board, ArrayList<BoardCell> list) {
    // Code for finding helper function
}
```
before the line) one line of the method findHelper on the next page so that letters can be re-used thus allowing “TOMATO” for the board shown above.
When you indicate which line you're commenting out (e.g., putting // before the code) you should also provide an explanation as to why removing the line causes repeats to be found.

```java
private boolean findHelper(String word, int index, int row, int col, BoggleBoard board, List<BoardCell> list) {
    if (index >= word.length()) return true; // found the word!
    if (row < 0 || row >= board.size() || col < 0 || col >= board.size()){
        return false;
    }
    String current = word.substring(index,index+1);
    if (board.getFace(row, col).equals(current)) {
        BoardCell value = new BoardCell(row, col);
        if (list.contains(value)) return false;
        list.add(value);
        int[] rdelta = {-1,-1,-1, 0, 0, 1, 1, 1};
        int[] cdelta = {-1, 0, 1,-1, 1,-1, 0, 1};
        for(int k=0; k < rdelta.length; k++){
            if (findHelper(word,index+1,row+rdelta[k], col+cdelta[k],board,list) {
                return true;
            }
        }
        list.remove(list.size() - 1);
    }
    return false;
}
```
Part B (6 points)
In the code on the previous page, the same letter cannot be used twice in a row. For example, the word “MAMMAL” cannot be formed using the code from the previous page because the double “M” isn’t found using the code (the corrected version in which repeats are allowed). However, “MAMMAL” can be found as shown below if the same letter can be repeated without intervening letters.

Assuming the correct line is commented out so that repeated letters are allowed, i.e., you’ve done Part A correctly, the code above can be modified by adding one new recursive call to `findHelper` so that the same letter can be repeated without intervening letters, thus allowing “MAMMAL” to be found. Indicate what new recursive call can be added, before the call to `list.remove`, so that “MAMMAL” and other such repeats are found (for example, “ROOT” and “TREE” will both be found with the proper line added). There will now be nine recursive calls.

Please justify briefly why adding the line results in words like “MAMMAL” being found.
Part C (3 points)
Describe in words why the private, helper method `findHelper` fails to work with cubes that contain “Qu” in matching words that appear on a Boggle board and provide a brief explanation of how to fix the code to cope with such cubes and words.

Part D (3 points)
Suppose cubes can be repeated, but a word with repeated cubes generates a lower score than a word with all cubes distinct. For example, for the word “TOMATO” highlighted on the board at the beginning of this problem the list of `BoardCell` objects returned follows.

(1,1) (2,1) (3,0) (2,0) (1,1) (2,1)

However, it’s possible to form “TOMATO” with no repeated cells/cubes by starting at a different “T” as follows.

(3,2) (3,1) (3,0) (2,0) (1,1) (2,1)

Describe at a very high level how to modify `cellsForWord` and `findHelper` so that the maximally scoring representation of a word is found, e.g., the word with the minimal number of repeated cubes. You don’t have to write code, you can describe in words what the modified code will need to do to find the maximally scoring word.
PROBLEM 4: (Fluffer (12 points))

In class we discussed height-balanced trees. An alternative to height-balanced is node-balanced — for example we want the number of nodes in every left subtree to be close to the number of nodes in the corresponding right subtree. When this is the case we call the tree fluffy. More precisely we want the ratio of the number of nodes in each left subtree to the number in the right subtree to be at least 45% (or vice versa). The tree on the left below is fluffy at the root with a ratio of 50%. It might be completely fluffy but we can’t see the smaller subtrees. For example, the subtree labeled with 50 nodes on the left might have the 50 nodes divided into a left subtree of 40 nodes and a right subtree of 10 nodes as shown on the right. Then the ratio is $\frac{1}{4}$ or 25% so the tree isn’t fluffy.

Part A (6 points)

The method isFluffy below correctly determines if a tree is fluffy as described above. The method adds one to the count of the number of nodes in each subtree to avoid division by zero problems. What is the running time of isFluffy for an N-node tree? Justify your answer, use big-O. Provide two answers: one in the average case when trees are roughly balanced, and one when trees are not balanced.

```java
public int nodeCount(TreeNode root) {
    if (root == null) return 0;
    return 1 + nodeCount(root.left) + nodeCount(root.right);
}

public boolean isFluffy(TreeNode root) {
    if (root == null) return true;
    int leftCount = 1 + nodeCount(root.left); 
    int rightCount = 1 + nodeCount(root.right);
    double ratio = leftCount * 1.0 / rightCount;
    if (ratio > 1.0) ratio = 1/ratio;
    if (ratio > 0.45 && isFluffy(root.left) && isFluffy(root.right)){
        return true;
    }
    return false;
}
```
Part B (6 points)
A student turns in the code below for `isFluffy` but calls the method `isBushy`. The method works correctly, returning the same values as `isFluffy`. What is the running time of this method? Use big-O and justify your answer. Provide two answers: one in the average case when trees are roughly balanced, and one when trees are not balanced.

```java
public boolean isBushy(TreeNode root){
    double ratio = 1.0*(1+nodeCount(root.left))/(1 + nodeCount(root.right));
    if (nodeCount(root.left) >= nodeCount(root.right)){
        ratio = 1.0*(1+nodeCount(root.right))/(1+nodeCount(root.left));
    }
    if (ratio > 0.45 && isBushy(root.left) && isBushy(root.right)){
        return true;
    }
    return false;
}
```
PROBLEM 5:  (Nothing in Common (15 points))

The Longest Common Subsequence or lcs of two strings is the longest sequence of characters in order, not necessarily adjacent, that is in common to both strings. This has applications in text processing, genomics, and web searching.

For example, the lcs of the strings “sorting” and “describe” is “sri” and the lcs of the strings “human” and “chimpanzee” is “hman”.

The code below on the left correctly returns the longest common subsequence of two strings.

```java
public String lcs(String a, String b){
    if (a.length() == 0 || b.length() == 0){
        return "";
    }
    // placeholder A
    if (a.charAt(0) == b.charAt(0)){
        String after = lcs(a.substring(1),b.substring(1));
        after = a.charAt(0) + after;
        // placeholder B
        return after;
    }
    String t1 = lcs(a.substring(1),b);
    String t2 = lcs(a,b.substring(1));
    if (t1.length() > t2.length()) return t1;
    return t2;
}
```

Part A (3 points)
Describe in words what the three cases in the code are and why these cases make the code correct.

Part B (4 points)
What is the running time of this code for two strings of N-characters? Use big-O and justify your answer.

(continued)
Part C (8 points)

The list of words on the left of the previous page shows some of the calls for the method \texttt{lcs} for the words “sorting” and “describe” – the list shows the two parameters passed to \texttt{lcs} and the number of calls with these parameters. For example, there are 491 calls with parameters “g” and “be”. For the strings “sorting” and “describe” there are 6,238 calls made to find the longest common subsequence of “sri”.

To make the method faster you will \textit{memoize} so that results are stored and retrieved rather than being recomputed. For this problem you'll describe how to implement memoization and how to make it work. The idea is to make at most one recursive call for each pair of parameters. To do this you'll use the class \texttt{Pair} below.

The idea is to modify \texttt{lcs} so that the result returned for parameters \((a,b)\) as a \texttt{Pair} is stored in a map and retrieved if it is stored rather than recomputed recursively.

You'll do three things for this part of the problem:

- Comment on the definition for the map which would be an instance variable.
- Show how to check the map and return the stored value if it's present.
- Show how to store a value in the map so it will be available for subsequent calls.

Using this memoization technique reduces the calls from 6,238 to 100 for the strings “sorting” and “describe”.

The definition for the map follows:

\[
\text{Map}\langle\text{Pair, String}\rangle \text{ myMemo = new HashMap}\langle\text{Pair, String}\rangle();
\]

**C.1**

Can you replace \texttt{HashMap} with \texttt{TreeMap} in the definition above and have the rest of the code work (after modifying \texttt{lcs} to use the memo/cache)? Justify your answer.

**C.2**

Put code in \texttt{lcs} at the location marked \textit{placeholder A} to check the cache and return the result for a \texttt{Pair} \texttt{p} defined as:

\[
\text{Pair p = new Pair(a,b);}
\]

**C.3**

Put code at the location marked \textit{placeholder B} so that a value is stored properly in the memo/cache for subsequent retrieval. Where else in the code would you also have to store values in the map (label on the \texttt{lcs} code page).
(nothing on this page)