This test has 13 pages, 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. Don’t worry about getting function or method names exactly right. Assume that all libraries and packages we’ve discussed are imported in any code you write.

Unless indicated otherwise, here’s the Node class for this test.

```java
public class Node{
    public String info;
    public Node next;
    public Node(String s, Node link){
        info = s;
        next = link;
    }
}
```
PROBLEM 1:  (It Depends (8 points))

Part A (3 points)
What value is returned by the call calculate(2043)? What is the complexity (big-Oh, in terms of N) of the call calculate(N)? Briefly justify your answers.

```java
public int calculate(int n){
    int prod = 1;
    while (prod < n){
        prod *= 2;
    }
    return prod;
}
```

Part B (5 points)
Consider method docalc below, the call docalc(6) evaluates to 21.

```java
public int docalc(int n){
    if (n == 0) return 0;
    return n + docalc(n-1);
}
```

Using big-Oh what is the running time of the call docalc(n)? Justify your answer.

Using big-Oh what is the value returned by the call docalc(n) (note: complexity of value returned, not running time: use big-Oh)

Using big-Oh what is the running time of the call docalc(docalc(n)) based on your answers to the previous two questions. Justify.

Using big-Oh what is the value returned by the call docalc(docalc(n)) (again, based on previous answers, justify).
PROBLEM 2:  (Reversal of Fortune (12 points))

Write the method reverse whose header is given below. The method reverse reverses the elements of the parameter queue q. For example, if q is represented by (a,b,c,d), with a the first element and d the last element of the queue, then after the call reverse(q) q is represented by (d,c,b,a).

Part A (4 points)

Write the method below that changes parameter q so that it is reversed. You may only define variables of type Stack, Queue, String, or int (no arrays, no ArrayList, etc.). Note that all Stack and Queue objects have a method size() that returns the number of elements in the object.

```java
Queue<String> q = new LinkedList<String>();
q.add("ant");
q.add("bat");
q.add("cat");
q.add("dog");
// contents of q are ("ant", "bat", "cat", "dog");

reverse(q);
// contents of q are ("dog", "cat", "bat", "ant");

public void reverse(Queue<String> q) {
```
Part B (4 points)
The code below is designed to reverse a queue as described earlier. When run, the code below generates an error message via an exception: No Such Element Exception.
Fix the code below so that it reverses a queue by adding one if statement in the code. You cannot add code other than an if statement whose guard/test uses parameter q somehow. You can use open/close braces in writing code.

```java
public static void reverse(Queue<String> q){

    String elt = q.remove();

    reverse(q);

    q.add(elt);

}
```

Part C (4 points)
Why does the LinkedList class implement the Queue interface, but the ArrayList class does not. Explain this in terms of performance using big-Oh to justify your answer.
PROBLEM 3: (Make-A-Wishlist (18 points))

The code below creates a linked-list of two Nodes each containing the string "2". The list’s first node is pointed to by first.

```java
Node first = new Node("2", null);
first.next = new Node("2", null);
```

Part A (4 points)
Write the method `makeSubList` so that `makeSubList(5)` returns a pointer to the first node of a linked list containing five nodes each with the value "5"; the call `makeSubList(3)` returns a pointer to a linked list with three nodes in it, each containing the value "3" and so on; i.e., so that conceptually `makeSubList(n)` returns a pointer to a linked list containing `n` Nodes each with the value "n".

```java
public Node makeSubList(int n) {
```
Part B (6 points)
Write the method `makeWholeList` so that `makeWholeList(4)` returns a pointer to a linked list containing the values below in the order shown

("1", "2", "2", "3", "3", "3", "4", "4", "4", "4")

In general `makeWholeList(n)` returns a list with \( n \times (n + 1)/2 \) nodes. There are \( k \) nodes containing the string "k" for each \( 1 \leq k \leq n \), with all nodes containing \( k-1 \) appearing before any node containing \( k \), i.e., so that the linked-list is in order. In writing you should call `makeSubList`, you should not call `new` in writing `makeWholeList`. Assume that `makeSubList` works as specified.

```java
public Node makeWholeList(int n) {
    // Implementation here
}
```
Part C (8 points)
As discussed above, the list returned by `makeWholeList(n)` contains $O(n^2)$ elements. Using O-notation, indicate the number of elements in each of the lists described below. For full credit you **must justify your answer**.

- The number of elements in the list returned by `makeWholeList(n)` if every node containing a value less than $n/2$ is removed from the list.

- The number of elements in the list returned by `makeWholeList(n)` if every node containing a value greater than or equal to $n/2$ is removed from the list.

- The number of elements in the list returned by `makeWholeList((int) Math.sqrt(n))`

- The number of elements in the list returned by `makeWholeList(n)` if every node containing a value divisible by 3 is removed.
PROBLEM 4: (Boggle My Mind (16 points))

These questions refer to classes used in the Boggle assignment.

Part A (3 points)
The first line of code in LexiconFirstAutoPlayer.findAllValidWords is a call to clear() – the entire body of the method is shown below:

```java
    clear();
    for(String word : lex) {
        if (word.length() < minLength) continue;
        List<BoardCell> list = myFinder.cellsForWord(board,word);
        if (list.size() > 0) {
            // found word
            add(word);
        }
    }
```

In writing BoardFirstAutoPlayer.findAllValidWords a student forgets to call clear(). The class/implementation works correctly in playing the first game of Boggle, i.e., obtains the same score and words as when a LexiconFirstAutoPlayer is used to play the same board. But in each subsequent game the score reported by the program gets increasingly larger than the corresponding score reported for the same board by a LexiconFirstAutoPlayer.

In a few words, explain why leaving out the call to clear() causes this behavior.

Part B (3 points)
When implemented reasonably, a BoardFirstAutoPlayer is much faster than a LexiconFirstAutoPlayer in finding words on the same board. A student suggests that LexiconFirstAutoPlayer could be made faster as follows:

- store all the letters/cubes found on a board in a set.
- Before calling myFinder.cellsForWord as shown in the code above, the word being searched for on the board is checked against the set of all letters on the board.
- If the letters in the word are not all present in the set of letters on the board, then the call to myFinder.cellsForWord is skipped.

Do you think this idea will make the LexiconFirstAutoPlayer faster? Why?
Part C (5 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” cannot be formed unless letters are re-used within a word. 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”.

Here’s a correct version of method `cellsForWord` in class `WordFinder` 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 (find(word, 0, r, c, board, list)) {
                return list;
            }
        }
    }
    return list; // or null to indicate not found
}
```

This method calls `find` 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 one line of the method `find` on the next page so that letters can be re-used thus allowing “TOMATO” for the board shown above.
private boolean find(String word, int index, int row, int col, BoggleBoard board, List<BoardCell> list) {
    if (index >= word.length()) {
        return true;
    }
    if (row < 0 || row >= board.size() || col < 0 || col >= board.size()){
        return false;
    }
    char currentChar = word.charAt(index);
    if (board.getFace(row, col).charAt(0) == currentChar) {
        BoardCell value = new BoardCell(row, col);
        if (list.contains(value)) return false;
        list.add(value);
        if (find(word,index+1,row+1,col,board,list)) return true;
        if (find(word,index+1,row+1,col+1,board,list)) return true;
        if (find(word,index+1,row+1,col-1,board,list)) return true;
        if (find(word,index+1,row, col-1,board,list)) return true;
        if (find(word,index+1,row, col+1,board,list)) return true;
        if (find(word,index+1,row-1,col,board,list)) return true;
        if (find(word,index+1,row-1,col+1,board,list)) return true;
        if (find(word,index+1,row-1,col-1,board,list)) return true;
    }
    list.remove(list.size() - 1);
    return false;
}
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 C correctly, the code above can be modified by adding one new recursive call to `find` 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.
In this problem you’ll write a method hasPath to determine if there’s a path connecting the left side of a grid to the right side of the grid. A path consists of a sequence of horizontally and vertically adjacent ’x’ characters. The grid contains only ’.’ and ’x’ characters.

For example, consider the grids below.

```java
String[] a = { "...x....x", "........", "...x....", "...x....x", "...x.....", "........", "...x....", "...x.....", "...xxx...", "xxxxxxxx", "...x....", "...x.x...", "xxx.x.xx", "........", "...x....", "...x.x.xx", "xxxx.xxx.", "xxxxxxx.", "...x.....", ".........", "........."};
String[] b = { "...x.....", "........", "...x....", "...x....x", "...x.....", "........", "...x....", "...x....", "...xxx...", "xxxxxxx", "...x....", "...x.x...", "xxx.x.xx", "........", "...x....", "...x.x.xx", "xxxx.xxx.", "xxxxxxx.", "...x.....", ".........", "........."};
String[] c = { "...x....x", "........", "...x....", "...x....x", "...x.....", "........", "...x....", "...x.....", "...xxx...", "xxxxxxxx", "...x....", "...x.x...", "xxx.x.xx", "........", "...x....", "...x.x.xx", "xxxx.xxx.", "xxxxxxx.", "...x.....", ".........", "........."};
String[] d = { "...x....x", "........", "...x....", "...x....x", "...x.....", "........", "...x....", "...x.....", "...xxx...", "xxxxxxxx", "...x....", "...x.x...", "xxx.x.xx", "........", "...x....", "...x.x.xx", "xxxx.xxx.", "xxxxxxx.", "...x.....", ".........", "........."};
```

hasPath(a) == true
hasPath(b) == true
hasPath(c) == false
hasPath(d) == false

You can write hasPath in any way, but here’s a hint:

- Start at every grid location on the left and explore paths from it using a helper method.
- Exploring a grid location means: if it’s off the grid stop, if it doesn’t contain an ’x’ stop (no path, return false).
- Exploring a grid location on the right means: if it’s an x return true, otherwise return false
- Exploring a grid location means: if it contains an ’x’ it might be part of a path. Mark it as explored, check four neighbors, unmark it as explored.

Complete the method hasPath so that it works as specified. The code below uses a helper method help that is designed to use the hint above. You don’t need to use this hint, but you can.

```java
char[][] myGrid;
public boolean hasPath(String[] grid){
    myGrid = new char[grid.length][grid[0].length()];
    for(int r=0; r < grid.length; r++){
        for(int c=0; c < grid[0].length(); c++){
            myGrid[r][c] = grid[r].charAt(c);
        }
    }
    for(int r=0; r < grid.length; r++){
        if (help(r,0)){
            return true;
        }
    }
    return false;
}
```

(continued)
private boolean help(int row, int col){

}