Part I. Recursion

1. Write a recursive method `public static int sumDigits(int n)` that returns the sum of the digits of its parameter. (If you're stuck, look at Part II of this lab for inspiration.)

2. Write a recursive method `public static String reverse(String s)` that reverses the order of characters in a string.

3. Complete the `Recursion-1/changePi` problem on codingbat.com


Part II. Recursion in the debugger

1. Type the following recursive method into Eclipse:

   ```java
   public static int prodDigits(int n)
   {
       if(n < 10)
       {
           return n;
       }
       else
       {
           return (n%10) * prodDigits(n/10);
       }
   }
   ```

2. Write a main method that calls `prodDigits(1234567)`.

3. Set a breakpoint at the first line of `prodDigits` (the if).

4. Run your program and press step several times. You will see that each time the `prodDigits` method is called, a new line is added to the Debug view in Eclipse. (If you don't have a Debug view – a little window with a tab on it saying Debug – then go to the Window menu and find it under Show View.)

5. After you have several lines with `prodDigits` on them, open up the Variables view as well.

6. Click on the different `prodDigits` lines in the Debug view. Notice how the local variables displayed in the Variables view change. This is because you're selecting
different recurrences of `prodDigits`; each recurrence has its own copy of local variables.

7. Continue stepping to watch how the `Variables` window can also track what value a method call returns.

**Part III. Recursive linked lists**

Linked lists are a *recursive data structure*. Look at the `Node` class (available from the syllabus page):

```java
public class Node {
    public String data;
    public Node next;

    ...
}
```

We see that `Node` contains a field of type `Node`, just like a recursive method calls itself. Indeed, it's wonderfully easy to write linked-list manipulation functions recursively.

1. Make a new Eclipse project, downloading the `Node.java` file on the syllabus for it.

2. Make a new class, and start a `main` method there like this:

```java
public static void main(String[] args) {
    Node head = Node.makeList("a","b","c","d");

    // you will add code here
}
```

Note the call of `Node`'s `static makeList` method. This method's parameter is declared to be `(String... elts)`, using Java's so-called `varargs` facility (short for *variable number of arguments*). It means that you can pass any number of strings (including 0 of them) to `makeList`; the `elts` parameter will be an array of all the strings. You won't have to master this technique, but it sure is useful for creating lists.

3. In the "add code here" spot, write a loop that prints out the elements of the list headed by `head`. Run your `main` to make sure all the parts are working.

4. Write the following recursive method in your class:

```java
public static void printList(Node head) {
```
if (head == null)
{
    return; // list is empty
}
else
{
    System.out.println(head.data);
    printList(head.next);
}

Run this method from `main` to show that it prints out the elements of the list.

5. Modify the `printList` method by swapping the order of the two lines in the `else`. How does this change the behavior? Why? Put your answers in comments in the file.

6. Write the following recursive method in your class:

   public static String allTogether(Node head)
   {
      if (head == null)
      {
         return "";
      }
      else
      {
         return head.data + allTogether(head.next);
      }
   }

   What does this method do? Test your hypothesis by calling the method and printing its result.

7. Write the following recursive method in your class:

   public static Node removeOdds(Node head)
   {
      if (head == null || head.next == null)
      {
         return head;
      }
      else
      {
         return new Node(head.data,
                          removeOdds(head.next.next));
      }
   }
What does this method do? Test it on a variety of inputs.

8. In removeOdds, what would go wrong if you dropped the head.next == null check? What would go wrong if you reversed the order of the two checks in the if condition? (Look up "Java short circuit" to learn more.)

9. To understand how removeOdds works, it's best to trace an evaluation of the method. Evaluation is a concept we have not talked about in the context of Java, because evaluation doesn't really make sense with mutation (the ability to change the value of a variable). But, with recursive methods, we generally don't have mutation, and so evaluation works great. By evaluation, I mean to consider how the Java expression \((1 + 2)\) evaluates to 3. More elaborately, the Java expression \(((4 \times 3) + (2 \times 3))\) evaluates to \((12 + (2 \times 3))\), which evaluates to \((12 + 6)\), which evaluates to 18. Using standard notation for evaluation:

\[
((4 \times 3) + (2 \times 3)) \rightarrow \\
(12 + (2 \times 3)) \rightarrow \\
(12 + 6) \rightarrow \\
18
\]

Let's do the same for removeOdds, using \{"a", "b", "c"\}, for example, to denote the list returned by Node.makeList("a", "b", "c"). The lines below are labeled to be discussed below.

1) \(\text{removeOdds}\{"a", "b", "c", "d"}\) \(\rightarrow\)
2) \(\text{new Node("a", removeOdds}\{"c", "d"}\}) \(\rightarrow\)
3) \(\text{new Node("a", new Node("c", removeOdds\{\})}) \(\rightarrow\)
4) \(\text{new Node("a", new Node("c", \{\}) =\}
5) \{"a", "c"\}

In line (1), the if condition in removeOdds is false, and so we go into the else. This says that removeOdds returns new Node(...). In other words, removeOdds\{\} evaluates to that new Node(...). We see this in line (2) above, where the call to removeOdds in line (1) is replaced by new Node("a", removeOdds\{"c", "d"\}). The list becomes \{"c", "d"\} because that line in removeOdds passes head.next.next to the recursive call, looking past two nodes.

Getting from line (2) to (3) is similar.

In line (3), we pass the empty list \{\} (that is, null) to removeOdds and so the if condition is true. We return head, which is just null, which can also be written as \{\}. This gets us to line (4). Between line (4) and (5), I've written ==, because there is no evaluation there – just a change in notation.
Using this as a template, write the series of evaluations that get us from 
removeOdds({"a", "b", "c", "d", "e"}) to an expression with no uses of 
removeOdds in it. Include this in a comment in your code.

10. As we can see, the removeOdds method removes all nodes that have an odd index 
(counting from 0). Using this method as a template, write removeEvens that removes 
all nodes with an even index. Note that we're not really removing anything here, but 
creating a new copy of the list missing certain elements.

11. Write a recursive removeAs that removes all nodes from a linked list whose data string 
contains an a. So, removeAs({"baa", "baa", "black", "sheep", 
"have", "you", "any", "wool"}) evaluates to {"sheep", "you", 
"wool"}.

12. Write a recursive stutter that duplicates every element. So, 
stutter({"a", "b", "c"}) produces {"a", "a", "b", "b", "c", "c"}. 
