RatRoute continued

Last discussion, we talked about how RatRoute, and how the number of paths from any point to the cheese is the sum of the number of paths from the two adjacent points in the direction of the cheese:

\[
\begin{array}{ccc}
R & . & 1 & . \\
. & 2 & 0 & 1 \\
. & 2 & 1 & . \\
. & 1 & 0 & 1 \\
. & 1 & . & C \\
\end{array}
\]

\[
\begin{array}{ccc}
C & 1 & . & X \\
. & 1 & 2 & . \\
. & 1 & 3 & . \\
. & 0 & 5 & . \\
. & . & . & R \\
\end{array}
\]
Recursive starter code

```java
public class RatRoute {
    public int numRoutes(String[] enc) {
        //find the rat’s row and column, and the cheese’s row and column
        return numRoutes(enc, ratRow, ratCol, cheeseRow, cheeseCol);
    }

    /*Method that returns number of routes from row, col, to cheeseRow, cheeseCol*/
    private int numRoutes(String[] enc, int row, int col, int cheeseRow, int cheeseCol){
        //check base cases – are we at the cheese? at an X? if so, return
        //determine if we’re on the same row or column as the cheese, and
        //what direction the cheese is in vertically and horizontally. then,
        //make the appropriate recursive calls and return
    }
}
```
Technical details

The method `numRoutes(String[] enc, int row, int col, int cheeseRow, int cheeseCol)` should return the number of valid paths from (row, col) to (cheeseRow, cheeseCol).

(Note: it may be easier to turn `enc` into a 2D character array, and have `numRoutes` take that as an input instead).

How can we get the character in row `i`, column `j` from `enc`?
The method `numRoutes(String[] enc, int row, int col, int cheeseRow, int cheeseCol)` should return the number of valid paths from `(row, col)` to `(cheeseRow, cheeseCol).

(Note: it may be easier to turn `enc` into a 2D character array, and have `numRoutes` take that as an input instead).

How can we get the character in row `i`, column `j` from `enc`?

`enc[i].charAt(j)`
Base cases

The two base cases for `private int numRoutes(String[] enc, int row, int col, int cheeseRow, int cheeseCol){` are:

1. We are at the cheese – return 1
2. We are at an X – return 0

How should we check at the beginning of `numRoutes` for each of these cases?
Base cases

The two base cases for `private int numRoutes(String[] enc, int row, int col, int cheeseRow, int cheeseCol){ are:

1. We are at the cheese – return 1
2. We are at an X – return 0

How should we check at the beginning of numRoutes for each of these cases?

1. Check if row and col are equal to cheeseRow and cheeseCol
2. Check if `enc[row].charAt(col)` is the character X
Recursive cases

If we’re not at a base case in a given call of numRoutes, then we should look at each direction we’re allowed to move in, and sum the number of paths from the squares in these directions.

Some pseudo-code for this:

```python
count = 0
if (we can move left)
    count = count + numRoutes(enc, row, col-1, cheeseRow, cheeseCol)
(Repeat for the other three directions)
return count
```

How do we check if we can move in a given direction?
Recursive cases

If we’re not at a base case in a given call of numRoutes, then we should look at each direction we’re allowed to move in, and sum the number of paths from the squares in these directions. Some pseudo-code for this:

```python
count = 0
if (we can move left)
    count = count + numRoutes(enc, square to the left, cheese square)
(Repeat for the other three directions)
return count
```

How do we check if we can move in a given direction?

Check if row < cheeseRow, row > cheeseRow, col < cheeseCol, col > cheeseCol for up, down, right, left respectively.
Another approach

It’s also possible to solve RatRoute using recursion with memoization, or equivalently, not using recursion at all! Consider this pseudo code:

```plaintext
numRoutes = int array same size as enc
numRoutes[cheeseRow][cheeseCol] = 1
for row from cheeseRow to ratRow:
    for col from cheeseCol to ratCol:
        numRoutes[row][col] = 
            numRoutes[row+1][col]+numRoutes[row][col+1]
        //plus or minus depending on direction
return numRoutes[ratRow][ratCol]
```

We won’t go into the details of how to write this version (though you’re free to write it this way in your solution), but note that it achieves the same result using the same logic without actually making a recursive call.
Linked List exercises

Snarf the code for today. You’ll see an incomplete class named StringLinkedList, which is a class implementing a LinkedList, where the nodes have String values.

For the rest of discussion we’ll have you complete the methods in this class to get practice with linked lists. First, we’ll teach you some common tricks you can use with Linked Lists.
Linked list review

Recall that a linked list is an ordered list stored within a series of nodes. Each node includes a value, and then a pointer to the next node.

We maintain a pointer to the first node (called the head), which we can use to access the entire list. The last node’s pointer points to null, since it has no next node.

We will usually use a linked list by keeping a pointer to a node (we’ll often call this “current”), starting that pointer at the head, and then moving that pointer along the list.
Iterating over nodes in a linked list

The following is Java code to iterate over a linked list (depending on how it’s implemented, some
different variable names may apply). You can see an example of this in the toString method in
LinkedStringList.

```java
Node current = myHead;
while (current != null){
    //do something with current
    current = current.next;
}
```

Note how we take advantage of the fact that the last node’s next variable points to null.
Getting to the last node

Sometimes, you’ll have a pointer to the last node in a linked list. However, if you only have a pointer to the first node, you can still easily get to the last node:

```java
Node current = myHead;
while (current.next != null){
    current = current.next;
}
```

After the while loop, current will point to the last node. Notice that the condition in the while loop is different from the previous slide.
Complete the incomplete methods in LinkedStringList. Use TestLinkedStringList to check if your solutions are correct at the end.

If you finish early, as a challenge try to implement each of them using recursion. (Hint: You’ll want to write a recursive helper method which takes a Node as an argument, and then call it on myHead).