Before we begin...

Any questions about the midterm solutions?
Making a Markov Map

Recall that in Markov, we’re trying to make a map of all k-grams to all k-grams that follow them in the source text. We also note which k-gram is followed by the end of the file in our map.

For example, consider the String “aababaa” and k = 3. The chain of 3-grams is:

“aab” → “aba” → “baa” → “aab” → “aba” → “baa”

We could represent this as the following table:

<table>
<thead>
<tr>
<th>3-gram</th>
<th>Following 3-gram</th>
<th>Following character</th>
</tr>
</thead>
<tbody>
<tr>
<td>aab</td>
<td>aba, aba</td>
<td>a, a</td>
</tr>
<tr>
<td>aba</td>
<td>baa, baa</td>
<td>a, a</td>
</tr>
<tr>
<td>baa</td>
<td>aab</td>
<td>b</td>
</tr>
</tbody>
</table>

In Java, you could represent this as a Map of Strings to ArrayLists of Characters or ArrayLists of Strings
Markov Exercise 1:

Write the table representing Markov map for the following String, with k = 2:

“ababcbabbabc”

You can have the table show the list of following 2-grams, or the list of following characters.
Markov text property

When we make Markov text using \( k = i \), it has the property that any string of \( i+1 \) characters in the text should also occur in the original text. This is because each \( i \)-gram can only be followed by a character that follows it somewhere in the source text.

For example, if our source text is “ababcbbabc” and we use \( k = 2 \), every 3-gram in “abcbbaba” occurs in the source text, so this could be the Markov text we generate.

However, “abcc” or “abbbbb” could not be the Markov text we generate since “bcc”, “abb”, and “bbb” are all 3-grams that don’t occur in the source text.
Markov Exercise 2:

Suppose the source text is “abababab”, and we generate Markov text using k = 2. What 3-grams can occur in our Markov text? What about if we use the source text “aabbaabb”? 
Markov Exercise 3

In generating Markov text, our goal is to create text which is similar to the source text (so that it’s legible), but not the exact same (so that it’s capable of generate sequences of text not in the source text).

Given the property mentioned two slides ago (if we use k-grams as the keys/values in our map, every k+1-gram in the Markov text must also appear in the original text), what type of text will small k values generate? How about large k values?
Recursion review

Recall that in recursion, we’re trying to break down a problem’s answer into a function of the answers of smaller versions of the same problem (i.e. subproblems).

Eventually, we break down the problem into subproblems so small or simple that we can solve them near instantly.

Thus, when trying to write a recursive method, there’s two questions you should ask – how can I break down this problem, and what’s the base case(s)?
Recursion example

e.g. Suppose we want to find the $i^{th}$ Fibonacci number, where the 0$^{th}$ and 1$^{st}$ Fibonacci number are 1, and the $i^{th}$ Fibonacci number is the sum of the previous two. Then, if the problem “find the $i^{th}$ Fibonacci number” is $F(i)$, we can break down the problem recursively pretty easily:

$$F(i) = F(i-1) + F(i-2)$$

Here our base cases are $F(0) = 1$ and $F(1) = 1$

Another example: Suppose we want to find $n! = 1 \times 2 \times 3 \times \ldots \times n$ recursively using a function called factorial.

Then, since $n! = n \times (n-1)!$, we could express $\text{factorial}(n) = n \times \text{factorial}(n-1)$.

Obviously $1! = 1$, so our base case could be $\text{factorial}(1) = 1$. 
Writing recursive methods

The pseudocode for any recursive method is going to be:

```java
public recursiveMethod(input):
    if input is the base case:
        return the base case value
    else:
        call this method recursively to solve subproblems, then return some function of the recursive call
```

(Note that sometimes there are multiple cases for how to use the answer to the subproblems. In this case, you’ll have to have additional if statements to determine which case you’re in.)

e.g. to find the n\textsuperscript{th} Fibonacci number in Java:

```java
public int fib(int n){
    if (n == 0 || n == 1){
        return 1;
    }
    else{
        return fib(n-1) + fib(n-2);
    }
}
```
The pseudocode for any recursive method is going to be:

```java
public recursiveMethod(input):
    if input is the base case:
        return the base case value
    else:
        call this method recursively
        then return some function of the recursive calls
```

In class you discussed mergeSort – here's how we can break that down using this framework:

```java
static void mergeSort(int[] a) {
    if (a.length > 1) {
        // Base case: do nothing
        int len = a.length; // get length
        int mid = a.length/2; // get middle
        int[] lA = Arrays.copyOfRange(a, 0, mid);
        int[] rA = Arrays.copyOfRange(a, mid+1, len);
        mergeSort(lA); // sort left
        mergeSort(rA); // sort right
        a = merge(lA, rA); // merge
    }
}
```
Recursion Practice 1: Count Substrings

Complete the following method using recursion. The method should return the number of non-overlapping occurrences of subStr in str.

For example, if subStr is “ab” and str is “abcab”, you should return 2. If subStr is “aa” and str is “aaaaaaaa”, you should return 3 since while you could “aa” occurs 5 times, these occurrences overlap.

public int countSubstrings(String str, String subStr){
    //Write the code for this method
}

Recursion Practice 2: Collatz Conjecture

The Collatz conjecture states that given any number, if we repeat the following steps over and over again:

- If the number is even, we divide it by 2
- If the number is odd, we multiply it by 3 and add 1

We will eventually get to 1.

Write code for the following method, which determines how many of the steps in the Collatz conjecture we have to take to get from $n$ to 1. E.g. for $n=8$, we should return 3, and for $n=5$, we should return 5 – see the picture for why.

```java
public int collatz(int n){
    //return the number of steps from n to 1
}
```
For more recursion practice

You can get more practice with small recursion problems at
Problem Statement

A psychologist wants to study instinctive route-making behavior in lab rats. She builds a two-dimensional enclosure and places blocks (each one foot wide and one foot deep) at random points inside the enclosure. She then places a piece of cheese at a random point in the enclosure and a lab rat at another random point. Here is an example setup, where a square foot of space is represented by a period, a block is represented by an X, the rat is represented by an R, and the piece of cheese is represented by a C (spaces are added for easy reading).

```
 R . .
.. X ..
... X
X X X X
R . . .
```

Before putting the rat in the enclosure, the psychologist shows the location of the piece of cheese to the rat. Naturally, the rat will attempt to take a path with the shortest possible distance to the cheese. But the rat isn’t that intelligent so the rat will only move towards the cheese and never move away from the cheese. Assume that the rat can only move horizontally or vertically, one point at a time. In the above example, there are 3 possible routes the rat can take to get to the cheese:

1. East 2 points, South 4 points
2. South 2 points, East 2 points, South 2 points
3. South 4 points, East 2 points

Create a class RatRoute that contains the method numRoutes, which takes a String[] representing the configuration of the test (each string represents one row of the maze), and returns an int representing the number of possible routes the rat can take to get to the cheese without ever increasing the distance between itself and the cheese at any point on its path (see first NOTE).
Examples

For each of the examples to the right, how many valid paths are there? Remember, the rat can’t ever make a move that takes them away from the cheese.
<table>
<thead>
<tr>
<th>R . . X . C</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>C . . . . R</td>
<td>R + + + + +</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
<tr>
<td>. . . . R</td>
<td>R + + + + +</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
<tr>
<td>. . X . X . X</td>
<td>X + X . X</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
<tr>
<td>X . . R .</td>
<td>X + + R .</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
<tr>
<td>C + + X .</td>
<td>C . . X .</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
<tr>
<td>. . . . .</td>
<td>. . . . .</td>
</tr>
</tbody>
</table>
Breaking down the problem

Most path-finding problems can be solved using recursion. Let’s think about how to break down the problem into subproblems. First, for these two examples see if you can determine the number of paths to the cheese from every point between the rat and the cheese (note this is just examples 3 and 4 from the last slide, but with only points between the rat and cheese).
Solutions

Below are the number of paths from each point. What’s the common pattern amongst the red numbers?

\[
\begin{array}{cccc}
R_3 & 1 & 1 & 1 \\
X & 2 & 0 & 1 \\
X & 2 & 1 & 1 \\
C & 0 & 0 & 1 \\
1 & 1 & C_1
\end{array}
\]
Recursive formulation

Suppose \((i, j)\) are the coordinates of the square we want to find the number of paths from, and the cheese is at some \((r, c)\) where \(r > i, c > j\) (where \((0,0)\) is the top-left).

Then if \(f(i, j)\) is the number of paths from \((i, j)\), \(f(i, j) = f(i+1, j) + f(i, j+1)\).

This makes sense intuitively – every path to the cheese has to either go down or right to start.

The number of paths that start with going down is the same as number of paths from the square below us, and the number of paths that start with going right is same as the number of paths from the square to the right of us.

So, the number of paths from the current square is just the sum of these two.
Accounting for direction

If we’re on the same row as the cheese, we should only move horizontally. So, $f(i, j) = f(i, j+1) - f(i+1, j)$ takes us out of the row we want. If we’re in the same column, $f(i, j) = f(i, j+1)$.

Also, note that the recursive formula on the previous slide only works if the rat is above and to the left of the cheese.

If the rat is in a different direction, we need to change the $f(i+1, j)$ and $f(i, j+1)$ calls to go in a different direction.
Base cases

Now that we’ve identified the recursive formula, we just need to identify base cases.

First off, how many valid paths are there to the cheese from the cheese?

How many valid paths are these to the cheese from an X?
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
    }
}
Recursion is just iteration

At this point, you should know most of the theory needed to solve this problem recursively.

It turns out it’s possible to solve this problem without recursion – in the next discussion, we’ll talk about a way to solve RatRoute without recursion, using the same logic.