Chapter 12 - Recursion

Section 12.1 - Recursion: Introduction

An ***algorithm*** is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

Figure 12.1.1: Algorithms are like recipes.

Make lemonade:

- Add sugar to pitcher
- Add lemon juice
- Add water
- Stir

Some problems can be solved using a recursive algorithm. A ***recursive algorithm*** solves a problem by breaking that problem into smaller subproblems, solving these subproblems, and combining the solutions.

Figure 12.1.2: Mowing the lawn can be broken down into a recursive process.

- Mow the lawn
  - Mow the frontyard
    - Mow the left front
    - Mow the right front
  - Mow the backyard
    - Mow the left back
    - Mow the right back

An algorithm that is defined by repeated applications of the same algorithm on smaller problems is a ***recursive*** algorithm. The mowing algorithm consists of applying the mowing algorithm on smaller

pieces of the yard.

At some point, a recursive algorithm must describe how to actually do something, known as the ***base case***. The mowing algorithm could thus be written as:

- *Mow* the lawn
    - *If lawn is less than 100 square meters*
        - Push the lawnmower left-to-right in adjacent rows
    - *Else*
        - Mow one half of the lawn
        - Mow the other half of the lawn

Participation Activity

12.1.1: Recursion.

Which are recursive definitions/algorithms?

#	Question	Your answer
1	Helping N people: If N is 1, help that person. Else, help the first N/2 people, then help the second N/2 people.	True False
2	Driving to the store: Go 1 mile. Turn left on Main Street. Go 1/2 mile.	True False
3	Sorting envelopes by zipcode: If N is 1, done. Else, find the middle zipcode. Put all zipcodes less than the middle zipcode on the left, all greater ones on the right. Then sort the left, then sort the right.	True False

Section 12.2 - Recursive methods

A method may call other methods, including calling itself. A method that calls itself is a ***recursive method***.

Participation Activity

12.2.1: A recursive method example.

Start

```
public class CountDownTimer {
   public static void countDown(int countInt) {
      if (countInt == 0) {
         System.out.println("GO!");
      }
      else {
         System.out.println(countInt);
         countDown(countInt-1);
      }
      return;
   }

   public static void main (String[] args) {
      countDown(2);
      return;
   }
}
```

```
2
1
GO!
```

```
public static void main (String[] args) {
   CountDown(2);
   return;
}
```

```
public static void countDown(int countInt) {
   if (countInt == 0) {
      System.out.println("GO!");
   }
   else {
      System.out.println(countInt);
      countDown(countInt-1);
   }
   return;
}
```

countInt: 2

```
public static void countDown(int countInt) {
   if (countInt == 0) {
      System.out.println("GO!");
   }
   else {
      System.out.println(countInt);
      countDown(countInt-1);
   }
   return;
}
```

countInt: 1

```
public static void countDown(int countInt) {
   if (countInt == 0) {
      System.out.println("GO!");
   }
   else {
      System.out.println(countInt);
      countDown(countInt-1);
   }
   return;
}
```

countInt: 0

Each call to countDown() effectively creates a new "copy" of the executing method, as shown on the right. Returning deletes that copy.

The example is for demonstrating recursion; counting down is otherwise better implemented with a loop.

Participation Activity

12.2.2: Thinking about recursion.

Refer to the above countDown example for the following.

#	Question	Your answer
1	How many times is countDown() called if main() calls CountDown(5)?	
2	How many times is countDown() called if main() calls CountDown(0)?	
3	Is there a difference in how we define the parameters of a recursive versus non-recursive method? Answer yes or no.	

Challenge Activity

12.2.1: Calling a recursive method.

Write a statement that calls the recursive method backwardsAlphabet() with parameter startingLetter.

```
4           System.out.println(currLetter);
5        }
6        else {
7           System.out.print(currLetter + " ");
8           backwardsAlphabet(--currLetter);
9        }
10       return;
11    }
12
13    public static void main (String [] args) {
14       char startingLetter = '-';
15
16       startingLetter = 'z';
17
18       /* Your solution goes here  */
19
20       return;
21    }
22 }
```

Run

Section 12.3 - Recursive algorithm: Search

Consider a guessing game program where a friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses?

A first try might implement an algorithm that simply guesses in increments of 1:

- Is it 0? Higher
- Is it 1? Higher
- Is it 2? Higher

This algorithm requires too many guesses (50 on average). A second try might implement an algorithm

that guesses by 10s and then by 1s:

- Is it 10? Higher
- Is it 20? Higher
- Is it 30? Lower
- Is it 21? Higher
- Is it 22? Higher
- Is it 23? Higher

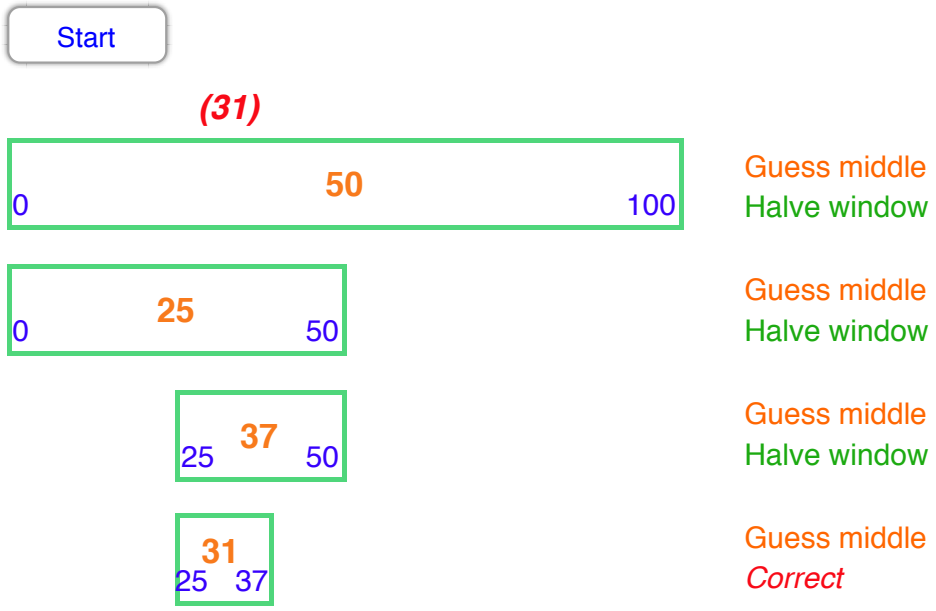
This algorithm does better but still requires about 10 guesses on average: 5 to find the correct tens digit and 5 to guess the correct ones digit. An even better algorithm uses a binary search. A ***binary search*** algorithm begins at the midpoint of the range and halves the range after each guess. For example:

- Is it 50 (the middle of 0-100)? Lower
- Is it 25 (the middle of 0-50)? Higher
- Is it 37 (the middle of 25-50)? Lower
- Is it 31 (the middle of 25-37).

After each guess, the binary search algorithm is applied again, but on a smaller range, i.e., the algorithm is recursive.

Participation Activity

12.3.1: Binary search: A well-known recursive algorithm.



A recursive method is a natural match for the recursive binary search algorithm. A method guessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range. The method guesses at the midpoint of the range. If the user says lower, the method calls guessNumber(lowVal, midVal). If the user says higher, the method calls guessNumber(midVal + 1, highVal)[mid].

Figure 12.3.1: A recursive Find() carrying out a binary search algorithm.

```
import java.util.Scanner;

public class NumberGuessGame {
   public static void guessNumber(int lowVal, int highVal) {
      Scanner scnr = new Scanner(System.in);
      int midVal = 0;          // Midpoint of low..high
      char userAnswer = '-'; // User response

      midVal = (highVal + lowVal) / 2;

      // Prompt user for input
      System.out.print("Is it " + midVal + "? (l/h/y): ");
      userAnswer = scnr.next().charAt(0);

      if ((userAnswer != 'l') && (userAnswer != 'h')) { // Base case: found number
         System.out.println("Thank you!");
```

```
         }
         else {                                          // Recursive case: split int
            if (userAnswer == 'l') {                     // Guess in lower half
               guessNumber(lowVal, midVal);              // Recursive call
            }
            else {                                       // Guess in upper half
               guessNumber(midVal + 1, highVal);         // Recursive call
            }
         }

         return;
      }

      public static void main(String[] args) {
         // Print game objective, user input commands
         System.out.println("Choose a number from 0 to 100.");
         System.out.println("Answer with:");
         System.out.println("   l (your num is lower)");
         System.out.println("   h (your num is higher)");
         System.out.println("   any other key (guess is right).");

         // Call recursive function to guess number
         guessNumber(0, 100);

         return;
      }
   }
```

```
Choose a number from 0 to 100.
Answer with:
   l (your num is lower)
   h (your num is higher)
   any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): l
Is it 32? (l/h/y): l
Is it 29? (l/h/y): h
Is it 31? (l/h/y): y
Thank you!
```

Participation Activity

12.3.2: Binary search tree tool.

The following program guesses the hidden number known by the user.

```
import java.util.Scanner;

public class NumberGuess {
   public static void Find(int low, int high) {
      Scanner scnr = new Scanner(System.in);
      int mid = 0; // Midpoint of low..high
      char answer = '\0';

      mid = (high + low)/2;
```

```
        System.out.print("Is it " + mid + "? (l/h/y): ");
        answer = scnr.next().charAt(0);

        if( (answer != 'l')
           && (answer != 'h') ) {            // Base case:
           System.out.println("Thank you!"); // Found number!
        }
        else { // Recursive case: Guess in
               // lower or upper half of range
           if (answer == 'l') {  // Guess in lower half
              Find(low, mid);    // Recursive call
           }
           else {                // Guess in upper half
              Find(mid+1, high); // Recursive call
           }
        }

        return;
     }

     public static void main (String[] args) {
        System.out.println("Choose a number from 0 to 100.");
        System.out.println("Answer with:");
        System.out.println("   l (your num is lower)");
        System.out.println("   h (your num is higher)");
        System.out.println("   any other key (guess is right).");

        Find(0, 100);

        return;
     }
  }
```

Current function	Function code

→main()	`public static void main (String[] args) {`
	`  System.out.println("Choose a number from 0 to 100.");`
	`  System.out.println("Answer with:");`
	`  System.out.println("  l (your num is lower)");`
	`  System.out.println("  h (your num is higher)");`
	`  System.out.println("  any other key (guess is right).");`
	
	`  Find(0, 100);`
	
	`  return;`
	`}`

The recursive method has an if-else statement. The if branch ends the recursion, known as the ***base case***. The else branch has recursive calls. Such an if-else pattern is common in recursive methods.

Search is commonly performed to quickly find an item in a sorted list stored in an array or ArrayList. Consider a list of attendees at a conference, whose names have been stored in alphabetical order in an array or ArrayList. The following quickly determines whether a particular person is in attendance.

Figure 12.3.2: Recursively searching a sorted list.

```
import java.util.Scanner;
import java.util.ArrayList;

public class NameFinder {
   /* Finds index of string in vector of strings, else -1.
      Searches only with index range low to high
      Note: Upper/lower case characters matter
   */
   public static int findMatch(ArrayList<String> stringList, String itemMatch,
                               int lowVal, int highVal) {
      int midVal = 0;     // Midpoint of low and high values
      int itemPos = 0;    // Position where item found, -1 if not found
      int rangeSize = 0; // Remaining range of values to search for match

      rangeSize = (highVal - lowVal) + 1;
      midVal = (highVal + lowVal) / 2;

      if (itemMatch.equals(stringList.get(midVal))) {           // Base case 1: item
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                                // Base case 2: matc
         itemPos = -1;
      }
      else {                                                    // Recursive case: s
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower half
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal);
         }
         else {                                                 // Search upper half
            itemPos = findMatch(stringList, itemMatch, midVal + 1, highVal);
         }
      }
```

```
      return itemPos;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<String> attendeesList = new ArrayList<String>(); // List of attendee
      String attendeeName = "";                                  // Name of attendee
      int matchPos = 0;                                          // Matched position

      // Omitting part of program that adds attendees
      // Instead, we insert some sample attendees in sorted order
      attendeesList.add("Adams, Mary");
      attendeesList.add("Carver, Michael");
      attendeesList.add("Domer, Hugo");
      attendeesList.add("Fredericks, Carlos");
      attendeesList.add("Li, Jie");

      // Prompt user to enter a name to find
      System.out.print("Enter person's name: Last, First: ");
      attendeeName = scnr.nextLine(); // Use nextLine() to allow space in name

      // Call function to match name, output results
      matchPos = findMatch(attendeesList, attendeeName, 0, attendeesList.size() - 1)
      if (matchPos >= 0) {
         System.out.println("Found at position " + matchPos + ".");
      }
      else {
         System.out.println("Not found.");
      }

      return;
   }
}
```

```
Enter person's name: Last, First: Meeks, Stan
Not found.

...

Enter person's name: Last, First: Adams, Mary
Found at position 0.

...

Enter person's name: Last, First: Li, Jie
Found at position 4.
```

findMatch() restricts its search to elements within the range lowVal to highVal. main() initially passes a range of the entire list: 0 to (list size - 1). findMatch() compares to the middle element, returning that element's position if matching. If not matching, findMatch() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found. If neither of those two base cases are satisfied, then findMatch() recursively searches either the lower or upper half of the range as appropriate.

Participation Activity

12.3.3: Recursive search algorithm.

Consider the above findMatch() method for finding an item in a sorted list.

#	Question	Your answer
1	If a sorted list has elements 0 to 50 and the item being searched for is at element 6, how many times will findMatch() be called?	
2	If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many recursive calls to findMatch() will be made during the failed search for "Apples"?	

Participation Activity

12.3.4: Recursive calls.

A list has 5 elements numbered 0 to 4, with these letter values: 0: A, 1: B, 2: D, 3: E, 4: F.

#	Question	Your answer
1	To search for item C, the first call is findMatch(0, 4). What is the second call to findMatch()?	findMatch(0, 0)
		findMatch(0, 2)
		findMatch(3, 4)
2	In searching for item C, findMatch(0, 2) is called. What happens next?	Base case 1: item found at midVal.
		Base case 2: rangeSize == 1, so no match.
		Recursive call: findMatch(2, 2)

Exploring further:

- Binary search from wikipedia.com

(*mid) Because midVal has already been checked, it need not be part of the new window, so midVal + 1 rather than midVal is used for the window's new low side, or midVal - 1 for the window's new high side. But the midVal - 1 can have the drawback of a non-intuitive base case (i.e., midVal < lowVal, because if the current window is say 4..5, midVal is 4, so the new window would be 4..4-1, or 4..3). rangeSize == 1 is likely more intuitive, and thus the algorithm uses midVal rather than midVal - 1. However, the algorithm uses midVal + 1 when searching higher, due to integer rounding. In particular, for window 99..100, midVal is 99 ((99 + 100) / 2 = 99.5, rounded to 99 due to truncation of the fraction in integer division). So the next window would again be 99..100, and the algorithm would

repeat with this window forever. midVal + 1 prevents the problem, and doesn't miss any numbers because midVal was checked and thus need not be part of the window.

Section 12.4 - Adding output statements for debugging

Recursive methods can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a findMatch() method that searches a sorted list for an item. All of findMatch()'s print statements start with `System.out.print(indentAmt + ...);`. Indent is typically some number of spaces. main() sets indent to three spaces. Each recursive call *adds* three more spaces. Note how the output now clearly shows the recursion depth.

Figure 12.4.1: Output statements can help debug recursive methods, especially if indented based on recursion depth.

```
import java.util.Scanner;
import java.util.ArrayList;

public class NameFinder {
   /* Finds index of string in vector of strings, else -1.
      Searches only with index range low to high
      Note: Upper/lower case characters matter
   */
   public static int findMatch(ArrayList<String> stringList, String itemMatch,
                               int lowVal, int highVal, String indentAmt) { // inden
      int midVal = 0;     // Midpoint of low and high values
      int itemPos = 0;    // Position where item found, -1 if not found
      int rangeSize = 0; // Remaining range of values to search for match

      System.out.println(indentAmt + "Find() range " + lowVal + " " + highVal);
      rangeSize = (highVal - lowVal) + 1;
      midVal = (highVal + lowVal) / 2;

      if (itemMatch.equals(stringList.get(midVal))) {           // Base case 1: item
         System.out.println(indentAmt + "Found person.");
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                                // Base case 2: matc
         System.out.println(indentAmt + "Person not found.");
         itemPos = -1;
      }
      else {                                                    // Recursive case: s
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower half
            System.out.println(indentAmt + "Searching lower half.");
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal, indentAmt + "
         }
         else {                                                 // Search upper half
            System.out.println(indentAmt + "Searching upper half.");
            itemPos = findMatch(stringList, itemMatch, midVal + 1, highVal, indentAm
```

```
            }
        }

        System.out.println(indentAmt + "Returning pos = " + itemPos + ".");
        return itemPos;
    }

    public static void main(String[] args) {
        Scanner scnr = new Scanner(System.in);
        ArrayList<String> attendeesList = new ArrayList<String>(); // List of attendee
        String attendeeName = "";                                  // Name of attendee
        int matchPos = 0;                                          // Matched position

        // Omitting part of program that adds attendees
        // Instead, we insert some sample attendees in sorted order
        attendeesList.add("Adams, Mary");
        attendeesList.add("Carver, Michael");
        attendeesList.add("Domer, Hugo");
        attendeesList.add("Fredericks, Carlos");
        attendeesList.add("Li, Jie");

        // Prompt user to enter a name to find
        System.out.print("Enter person's name: Last, First: ");
        attendeeName = scnr.nextLine(); // Use nextLine() to allow space in name

        // Call function to match name, output results
        matchPos = findMatch(attendeesList, attendeeName, 0, attendeesList.size() - 1,
        if (matchPos >= 0) {
            System.out.println("Found at position " + matchPos + ".");
        }
        else {
            System.out.println("Not found.");
        }

        return;
    }
}
```

```
Enter person's name: Last, First: Meeks, Stan
   Find() range 0 4
   Searching upper half.
      Find() range 3 4
      Searching upper half.
         Find() range 4 4
         Person not found.
         Returning pos = -1.
      Returning pos = -1.
   Returning pos = -1.
Not found.

...

Enter person's name: Last, First: Adams, Mary
   Find() range 0 4
   Searching lower half.
      Find() range 0 2
      Searching lower half.
         Find() range 0 1
         Found person.
         Returning pos = 0.
      Returning pos = 0.
   Returning pos = 0.
```

```
Found at position 0.
```

Some programmers like to leave the output statements in the code, commenting them out with "//" when not in use. The statements actually serve as a form of comment as well.

Participation Activity

12.4.1: Recursive debug statements.

Refer to the above code using indented output statements.

#	Question	Your answer
1	The above debug approach requires an extra parameter be passed to indicate the amount of indentation.	True
		False
2	Each recursive call should add a few spaces to the indent parameter.	True
		False
3	The method should remove a few spaces from the indent parameter before returning.	True
		False

Participation Activity

12.4.2: Output statements in a recursive function.

- Run the recursive program, and observe the output statements for debugging, and that the person is correctly not found.
- Introduce an error by changing `itemPos = -1` to `itemPos = 0` in the range size == 1 base case.
- Run the program, notice how the indented print statements help isolate the error of the person incorrectly being found.

```
1
2  import java.util.Scanner;
3  import java.util.ArrayList;
4
5  public class NameFinder {
6     /* Finds index of string in vector of strings, else -
7        Searches only with index range low to high
8        Note: Upper/lower case characters matter
9     */
10    public static int findMatch(ArrayList<String> stringL
11                                int lowVal, int highVal,
12       int midVal = 0;    // Midpoint of low and high val
13       int itemPos = 0;   // Position where item found, -
14       int rangeSize = 0; // Remaining range of values to
15
16       System.out.println(indentAmt + "Find() range " + l
17       rangeSize = (highVal - lowVal) + 1;
18       midVal = (highVal + lowVal) / 2;
19
```

Run

Section 12.5 - Creating a recursive method

Creating a recursive method can be accomplished in two steps.

- **Write the base case** -- Every recursive method must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may write that part of the method first, and then test. There may be multiple base cases.
- **Write the recursive case** -- The programmer then adds the recursive case to the method.

The following illustrates for a simple method that computes the factorial of N (i.e. N!). The base case is N = 1 or 1! which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for N > 1, and written as `else { fact = N * NFact( N - 1 ); }`.

Participation Activity

12.5.1: Writing a recursive method for factorial: First write the base case, then add the recursive case.

Start

Write and test base case (non-recursive case)

```
public static int nFact(int N) {
   int factResult = 0;
   if (N == 1) { // Base case
         factReslt = 1;
   }
   // FIXME: Finish
   return factResult;
}
// main(): Get N, print nFact(N)
```

```
Enter N: 1
N! is: 1
```

Add and test recursive case

```
public static int nFact(int N) {
   int factResult = 0;
   if (N == 1) { // Base case
         factResult = 1;
   }
   else { // Recursive case
         factResult = N * nFact(N - 1);
   }
   return factResult;
}
//main(): Get N, print nFact(N)
```

```
Enter N: 1
N! is: 1
Enter N: 6
N! is: 720
```

A common error is to not cover all possible base cases in a recursive method. Another common error is to write a recursive method that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Typically, programmers will use two methods for recursion. An "outer" method is intended to be called from other parts of the program, like the method `int calcFactorial(int inVal)`. An "inner" method is intended only to be called from that outer method, for example a method `int _calcFactorial(int inVal)` (note the "_"). The outer method may check for a valid input value, e.g., ensuring inVal is not negative, and then calling the inner method. Commonly, the inner method has parameters that are mainly of use as part of the recursion, and need not be part of the

outer method, thus keeping the outer method more intuitive.

Participation Activity

12.5.2: Creating recursion.

#	Question	Your answer
1	Recursive methods can be accomplished in one step, namely repeated calls to itself.	True
		False
2	A recursive method with parameter N counts up from any negative number to 0. An appropriate base case would be `N == 0`.	True
		False
3	A recursive method can have two base cases, such as `N == 0` returning 0, and `N == 1` returning 1.	True
		False

Before writing a recursive method, a programmer should determine:

1. Does the problem naturally have a recursive solution?
2. Is a recursive solution better than a non-recursive solution?

For example, computing N! (N factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution. The figure below illustrates how the factorial computation can be implemented as a loop. Conversely, binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

Figure 12.5.1: Non-recursive solution to compute N!

```
for (i = inputNum; i > 1; --i) {
   facResult = facResult * i;
}
```

Participation Activity

12.5.3: When recursion is appropriate.

#	Question	Your answer
1	N factorial (N!) is commonly implemented as a recursive method due to being easier to understand and executing faster than a loop implementation.	True
		False

Participation Activity

12.5.4: Output statements in a recursive function.

Implement a recursive method to determine if a number is prime. Skeletal code is provided in the isPrime method.

```
1
2 public class PrimeChecker {
3 // Returns 0 if value is not prime, 1 if value is prime
4    public static int isPrime(int testVal, int divVal) {
5       // Base case 1: 0 and 1 are not prime, testVal is
6
7       // Base case 2: testVal only divisible by 1, testV
8
9       // Recursive Case
10         // Check if testVal can be evenly divided by di
11         // Hint: use the % operator
12
13         // If not, recursive call to isPrime with testV
14      return 0;
15   }
16
17   public static void main(String[] args) {
18      int primeCheckVal = 0; // Value checked for prime
19
```

Run

Challenge Activity

12.5.1: Recursive method: Writing the base case.

Write code to complete doublePennies()'s base case. Sample output for below program:

```
Number of pennies after 10 days: 1024
```

Note: These activities may test code with different test values. This activity will perform three tests, wi
= 10, then with startingPennies = 1 and userDays = 40, then with startingPennies = 1 and userDays

Also note: If the submitted code has an infinite loop, the system will stop running the code after a few
never reached." The system doesn't print the test case that caused the reported message.

```
8         else {
9            totalPennies = doublePennies((numPennies * 2), numDays - 1);
10        }
11        return totalPennies;
12     }
13
14  // Program computes pennies if you have 1 penny today,
15  // 2 pennies after one day, 4 after two days, and so on
16     public static void main (String [] args) {
17        long startingPennies = 0;
18        int userDays = 0;
19
20        startingPennies = 1;
21        userDays = 10;
22        System.out.println("Number of pennies after " + userDays + " days: "
23              + doublePennies(startingPennies, userDays));
24        return;
25     }
26  }
```

Run

Challenge Activity

12.5.2: Recursive method: Writing the recursive case.

Write code to complete printFactorial()'s recursive case. Sample output if userVal is 5:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

```
13            System.out.print(factCounter + " * ");
14            nextCounter = factCounter - 1;
15            nextValue = nextCounter * factValue;
16
17            /* Your solution goes here  */
18
19         }
20      }
21
22      public static void main (String [] args) {
23         int userVal = 0;
24
25         userVal = 5;
26         System.out.print(userVal + "! = ");
27         printFactorial(userVal, userVal);
28
29         return;
30      }
31   }
```

Run

Section 12.6 - Recursive math methods

Fibonacci sequence

Recursive methods can solve certain math problems, such as computing the Fibonacci sequence. The ***Fibonacci sequence*** is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.; starting with 0, 1, the pattern is to compute the next number by adding the previous two numbers.

Below is a program that outputs the Fibonacci sequence values step-by-step, for a user-entered number of steps. The base case is that the program has output the requested number of steps. The recursive case is that the program needs to compute the number in the Fibonacci sequence.

Figure 12.6.1: Fibonacci sequence step-by-step.

```
import java.util.Scanner;

public class FibonacciSequence {
   /* Output the Fibonacci sequence step-by-step.
      Fibonacci sequence starts as:
      0 1 1 2 3 5 8 13 21 ... in which the first
      two numbers are 0 and 1 and each additional
      number is the sum of the previous two numbers
   */
   public static void computeFibonacci(int fibNum1, int fibNum2, int runCnt) {
      System.out.println(fibNum1 + " + " + fibNum2 + " = " +
                         (fibNum1 + fibNum2));

      if (runCnt <= 1) { // Base case: Ran for user specified
                         // number of steps, do nothing
      }
      else {             // Recursive case: compute next value
         computeFibonacci(fibNum2, fibNum1 + fibNum2, runCnt - 1);
      }

      return;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int runFor = 0; // User specified number of values computed

      // Output program description
      System.out.println("This program outputs the\n" +
                         "Fibonacci sequence step-by-step,\n" +
                         "starting after the first 0 and 1.\n");

      // Prompt user for number of values to compute
      System.out.print("How many steps would you like? ");
      runFor = scnr.nextInt();

      // Output first two Fibonacci values, call recursive function
      System.out.println("0\n1");
      computeFibonacci(0, 1, runFor);

      return;
   }
}
```

```
This p
Fibona
starti

How ma
0
1
0 + 1 =
1 + 1 =
1 + 2 =
2 + 3 =
3 + 5 =
5 + 8 =
8 + 13
13 + 2
21 + 3
34 + 5
```

Participation Activity

12.6.1: Recursive Fibonacci.

Complete computeFibonacci() to return F_N, where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, F_4 is 3, and continuing: F_N is F_{N-1} + F_{N-2}. Hint: Base cases are N == 0 and N == 1.

```
1
2  public class FibonacciSequence {
3     public static int computeFibonacci(int N) {
4
5        System.out.println("FIXME: Complete this method.");
6        System.out.println("Currently just returns 0.");
7
8        return 0;
9     }
10
11    public static void main(String[] args) {
12       int N = 4; // F_N, starts at 0
13
14       System.out.println("F_" + N + " is " + computeFibonacci(N));
15
16       return;
17    }
18 }
19
```

Run

Greatest common divisor (GCD)

Recursion can solve the greatest common divisor problem. The ***greatest common divisor*** (GCD) is the largest number that divides evenly into two numbers, e.g. GCD(12, 8) = 4. One GCD algorithm (described by Euclid around 300 BC) subtracts the smaller number from the larger number until both numbers are equal. Ex:

- GCD(12, 8): Subtract 8 from 12, yielding 4.
- GCD(4, 8): Subtract 4 from 8, yielding 4.
- GCD(4, 4): Numbers are equal, return 4

The following recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

Figure 12.6.2: Calculate greatest common divisor of two numbers.

```
import java.util.Scanner;

public class GCDCalc {
   /* Determine the greatest common divisor
      of two numbers, e.g. GCD(8, 12) = 4
   */
   public static int gcdCalculator(int inNum1, int inNum2) {
      int gcdVal = 0; // Holds GCD results

      if (inNum1 == inNum2) {    // Base case: Numbers are equal
         gcdVal = inNum1;        // Return value
      }
      else {                     // Recursive case: subtract smaller from larger
         if (inNum1 > inNum2) { // Call function with new values
            gcdVal = gcdCalculator(inNum1 - inNum2, inNum2);
         }
         else { // n1 is smaller
            gcdVal = gcdCalculator(inNum1, inNum2 - inNum1);
         }
      }

      return gcdVal;
   }

   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      int gcdInput1 = 0; // First input to GCD calc
      int gcdInput2 = 0; // Second input to GCD calc
      int gcdOutput = 0; // Result of GCD

      // Print program function
      System.out.println("This program outputs the greatest \n" +
                         "common divisor of two numbers.");

      // Prompt user for input
      System.out.print("Enter first number: ");
      gcdInput1 = scnr.nextInt();

      System.out.print("Enter second number: ");
      gcdInput2 = scnr.nextInt();

      // Check user values are > 1, call recursive GCD function
      if ((gcdInput1 < 1) || (gcdInput2 < 1)) {
         System.out.println("Note: Neither value can be below 1.");
      }
      else {
         gcdOutput = gcdCalculator(gcdInput1, gcdInput2);
         System.out.println("Greatest common divisor = " +  gcdOutput);
      }

      return;
   }
}
```

```
This proc
common di
Enter fir
Enter sec
Greatest

...

This proc
common di
Enter fir
Enter sec
Greatest

...

This proc
common di
Enter fir
Enter sec
Note: Nei
```

Participation Activity

12.6.2: Recursive GCD example.

#	Question	Your answer
1	How many calls are made to gcdCalculator() method for input values 12 and 8?	1
		2
		3
2	What is the base case for the GCD algorithm?	When both inputs to the method are equal.
		When both inputs are greater than 1.
		When inNum1 > inNum2.

Exploring further:

- Fibonacci sequence from wikipedia.com
- GCD algorithm from wikipedia.com

Challenge Activity

12.6.1: Writing a recursive math method.

Write code to complete raiseToPower(). Sample output if userBase is 4 and userExponent is 2 is sho
practicing recursion; a non-recursive method, or using the built-in method pow(), would be more com

```
4^2 = 16
```

```
8         else {
9            resultVal = baseVal * /* Your solution goes here  */;
10        }
11
12        return resultVal;
13     }
14
15     public static void main (String [] args) {
16        int userBase = 0;
17        int userExponent = 0;
18
19        userBase = 4;
20        userExponent = 2;
21        System.out.println(userBase + "^" + userExponent + " = "
22          + raiseToPower(userBase, userExponent));
23
24        return;
25     }
26  }
```

Run

Section 12.7 - Recursive exploration of all possibilities

Recursion is a powerful technique for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples.

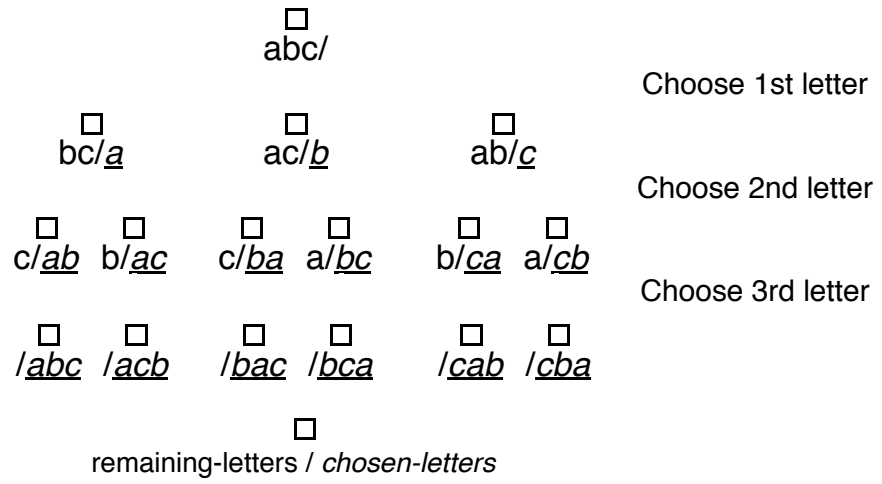
Word scramble

Consider printing all possible combinations (or "scramblings") of a word's letters. The letters of abc can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be listed by making

three choices: Choose the first letter (a, b, or c), then choose the second letter, then choose the third letter. The choices can be depicted as a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right.

Participation Activity

12.7.1: Exploring all possibilities viewed as a tree of choices.



The tree guides creation of a recursive exploration method to print all possible combinations of a string's letters. The method takes two parameters: unchosen letters, and already chosen letters. The base case is no unchosen letters, causing printing of the chosen letters. The recursive case calls the method once for each letter in the unchosen letters. The above animation depicts how the recursive algorithm traverses the tree. The tree's leaves (the bottom nodes) are the base cases.

The following program prints all possible ordering of the letters of a user-entered word.

Figure 12.7.1: Scramble a word's letters in every possible way.

```
import java.util.Scanner;

public class WordScrambler {
   /* Output every possible combination of a word.
      Each recursive call moves a letter from
```

```
   remainLetters" to scramLetters".
*/
public static void scrambleLetters(String remainLetters,  // Remaining letters
                                   String scramLetters) { // Scrambled letters
   String tmpString = ""; // Temp word combinations
   int i = 0;             // Loop index

   if (remainLetters.length() == 0) { // Base case: All letters used
      System.out.println(scramLetters);
   }
   else {                             // Recursive case: move a letter from
                                      // remaining to scrambled letters
      for (i = 0; i < remainLetters.length(); ++i) {
         // Move letter to scrambled letters
         tmpString = remainLetters.substring(i, i + 1);
         remainLetters = RemoveFromIndex(remainLetters, i);
         scramLetters = scramLetters + tmpString;

         scrambleLetters(remainLetters, scramLetters);

         // Put letter back in remaining letters
         remainLetters = InsertAtIndex(remainLetters, tmpString, i);
         scramLetters = RemoveFromIndex(scramLetters, scramLetters.length() - 1);
      }
   }
   return;
}

// Returns a new String without the character at location remLoc
public static String RemoveFromIndex(String origStr, int remLoc) {
   String finalStr = ""; // Temp string to extract char

   finalStr = origStr.substring(0, remLoc);                    // Copy before lc
   finalStr += origStr.substring(remLoc + 1, origStr.length()); // Copy after loc

   return finalStr;
}

// Returns a new String with the character specified by insertStr
// inserted at location addLoc
public static String InsertAtIndex(String origStr, String insertStr, int addLoc)
   String finalStr = ""; // Temp string to extract char

   finalStr = origStr.substring(0, addLoc);                // Copy before locati
   finalStr += insertStr;                                  // Copy character to
   finalStr += origStr.substring(addLoc, origStr.length()); // Copy after locatic

   return finalStr;
}

public static void main(String[] args) {
   Scanner scnr = new Scanner(System.in);
   String wordScramble = ""; // User defined word to scramble

   // Prompt user for input
   System.out.print("Enter a word to be scrambled: ");
   wordScramble = scnr.next();

   // Call recursive method
   scrambleLetters(wordScramble, "");
```

```
            return;
        }
    }
```

```
Enter a word to be scrambled: cat
cat
cta
act
atc
tca
tac
```

Participation Activity

12.7.2: Letter scramble.

#	Question	Your answer
1	What is the output of scrambleLetters("xy", "")? Determine your answer by manually tracing the code, not by running the program.	yx xy
		xx yy xy yx
		xy yx

Shopping spree

Recursion can find all possible subsets of a set of items. Consider a shopping spree in which a person can select any 3-item subset from a larger set of items. The following program prints all possible 3-item subsets of a given larger set. The program also prints the total price of each subset.

shoppingBagCombinations() has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items, which prints the items. The recursive case moves one of the remaining items to the bag, recursively calling the method, then moving the item back from the bag to the remaining items.

Figure 12.7.2: Shopping spree in which a user can fit 3 items in a shopping bag.

```
Milk  Belt  To
Milk  Belt  Cu
Milk  Toys  Be
```

GroceryItem.java:

```
public class GroceryItem {
   public String itemName;  // Name of item
   public int priceDollars; // Price of item
}
```

```
Milk  Toys  Cu
Milk  Cups  Be
Milk  Cups  To
Belt  Milk  To
Belt  Milk  Cu
Belt  Toys  Mi
Belt  Toys  Cu
Belt  Cups  Mi
Belt  Cups  To
Toys  Milk  Be
Toys  Milk  Cu
Toys  Belt  Mi
Toys  Belt  Cu
Toys  Cups  Mi
Toys  Cups  Be
Cups  Milk  Be
Cups  Milk  To
Cups  Belt  Mi
Cups  Belt  To
Cups  Toys  Mi
Cups  Toys  Be
```

ShoppingSpreeCombinations.java:

```
import java.util.ArrayList;

public class ShoppingSpreeCombinations {
   public static final int MAX_SHOPPING_BAG_SIZE = 3; // Max number of items in sho

   /* Output every combination of items that fit
      in a shopping bag. Each recursive call moves
      one item into the shopping bag.
   */
   public static void shoppingBagCombinations(ArrayList<GroceryItem> currBag,
                                              ArrayList<GroceryItem> remainingItems
      int bagValue = 0;           // Cost of items in shopping bag
      GroceryItem tmpGroceryItem; // Grocery item to add to bag
      int i = 0;                  // Loop index

      if (currBag.size() == MAX_SHOPPING_BAG_SIZE) {   // Base case: Shopping bag f
         bagValue = 0;
         for (i = 0; i < currBag.size(); ++i) {
            bagValue += currBag.get(i).priceDollars;
            System.out.print(currBag.get(i).itemName + "   ");
         }
         System.out.println("= $" + bagValue);
      }
      else {                                           // Recursive case: move one
         for (i = 0; i < remainingItems.size(); ++i) { // item to bag
            // Move item into bag
            tmpGroceryItem = remainingItems.get(i);
            remainingItems.remove(i);
            currBag.add(tmpGroceryItem);

            shoppingBagCombinations(currBag, remainingItems);

            // Take item out of bag
            remainingItems.add(i, tmpGroceryItem);
            currBag.remove(currBag.size() - 1);
         }
```

```
        }
        return;
    }

    public static void main(String[] args) {
        ArrayList<GroceryItem> possibleItems = new ArrayList<GroceryItem>(); // Possi
        ArrayList<GroceryItem> shoppingBag = new ArrayList<GroceryItem>();    // Curre
        GroceryItem tmpGroceryItem;                                           // Temp

        // Populate grocery with different items
        tmpGroceryItem = new GroceryItem();
        tmpGroceryItem.itemName = "Milk";
        tmpGroceryItem.priceDollars = 2;
        possibleItems.add(tmpGroceryItem);

        tmpGroceryItem = new GroceryItem();
        tmpGroceryItem.itemName = "Belt";
        tmpGroceryItem.priceDollars = 24;
        possibleItems.add(tmpGroceryItem);

        tmpGroceryItem = new GroceryItem();
        tmpGroceryItem.itemName = "Toys";
        tmpGroceryItem.priceDollars = 19;
        possibleItems.add(tmpGroceryItem);

        tmpGroceryItem = new GroceryItem();
        tmpGroceryItem.itemName = "Cups";
        tmpGroceryItem.priceDollars = 12;
        possibleItems.add(tmpGroceryItem);

        // Try different combinations of three items
        shoppingBagCombinations(shoppingBag, possibleItems);

        return;
    }
}
```

Participation Activity

12.7.3: All letter combinations.

#	Question	Your answer
1	When main() calls shoppingBagCombinations(), how many items are in the remainingItems list?	None
		3
		4

2	When main() calls shoppingBagCombinations(), how many items are in currBag list?	None
		1
		4
3	After main() calls ShoppingBagCombinations(), what happens first?	The base case prints Milk, Belt, Toys.
		The method bags one item, makes recursive call.
		The method bags 3 items, makes recursive call.
4	After shoppingBagCombinations() returns back to main(), how many items are in the remainingItems list?	None
		4
5	How many recursive calls occur before the first combination is printed?	None
		1
		3
6	What happens if main() only put 2, rather than 4, items in the possibleItems list?	Base case never executes; nothing printed.
		Infinite recursion occurs.

Traveling salesman

Recursion is useful for finding all possible paths. Suppose a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those three cities,

starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 12.7.3: Find distance of traveling to 3 cities.

```
import java.util.ArrayList;

public class TravelingSalesmanPaths {
   public static final int NUM_CITIES = 3;                                  // Number
   public static int[][] cityDistances = new int[NUM_CITIES][NUM_CITIES]; // Distanc
   public static String[] cityNames = new String[NUM_CITIES];               // City na

   /* Output every possible travel path.
      Each recursive call moves to a new city.
   */
   public static void travelPaths(ArrayList<Integer> currPath,
                                  ArrayList<Integer> needToVisit) {
      int totalDist = 0; // Total distance given current path
      int tmpCity = 0;   // Next city distance
      int i = 0;         // Loop index

      if ( currPath.size() == NUM_CITIES ) { // Base case: Visited all cities
         totalDist = 0;                      // Return total path distance
         for (i = 0; i < currPath.size(); ++i) {
            System.out.print(cityNames[currPath.get(i)] + "   ");

            if (i > 0) {
               totalDist += cityDistances[currPath.get(i - 1)][currPath.get(i)];
            }
         }

         System.out.println("= " + totalDist);
      }
      else {                                 // Recursive case: pick next city
         for (i = 0; i < needToVisit.size(); ++i) {
            // add city to travel path
            tmpCity = needToVisit.get(i);
            needToVisit.remove(i);
            currPath.add(tmpCity);

            travelPaths(currPath, needToVisit);

            // remove city from travel path
            needToVisit.add(i, tmpCity);
            currPath.remove(currPath.size() - 1);
         }
      }

      return;
   }

   public static void main (String[] args) {
      ArrayList<Integer> needToVisit = new ArrayList<Integer>(); // Cities left to v
      ArrayList<Integer> currPath = new ArrayList<Integer>();    // Current path tra

      // Initialize distances array
      cityDistances[0][0] = 0;
```

```
            cityDistances[0][1] = 960;  // Boston-Chicago
            cityDistances[0][2] = 2960; // Boston-Los Angeles
            cityDistances[1][0] = 960;  // Chicago-Boston
            cityDistances[1][1] = 0;
            cityDistances[1][2] = 2011; // Chicago-Los Angeles
            cityDistances[2][0] = 2960; // Los Angeles-Boston
            cityDistances[2][1] = 2011; // Los Angeles-Chicago
            cityDistances[2][2] = 0;

            cityNames[0] = "Boston";
            cityNames[1] = "Chicago";
            cityNames[2] = "Los Angeles";

            needToVisit.add(new Integer(0)); // Boston
            needToVisit.add(new Integer(1)); // Chicago
            needToVisit.add(new Integer(2)); // Los Angeles

            // Explore different paths
            travelPaths(currPath, needToVisit);

            return;
        }
    }
```

```
Boston   Chicago   Los Angeles   = 2971
Boston   Los Angeles   Chicago   = 4971
Chicago   Boston   Los Angeles   = 3920
Chicago   Los Angeles   Boston   = 4971
Los Angeles   Boston   Chicago   = 3920
Los Angeles   Chicago   Boston   = 2971
```

Participation Activity

12.7.4: Recursive exploration.

#	Question	Your answer
1	You wish to generate all possible 3-letter subsets from the letters in an N-letter word (N>3). Which of the above recursive methods is the closest?	shoppingBagCombinations
		scrambleLetters
		main()

Exploring further:

- Recursion trees from wikipedia.org

Section 12.8 - Stack overflow

Recursion enables an elegant solution to some problems. But, for large problems, deep recursion can cause memory problems. Part of a program's memory is reserved to support function calls. Each method call places a new ***stack frame*** on the stack, for local parameters, local variables, and more method items. Upon return, the frame is deleted.

Deep recursion could fill the stack region and cause a ***stack overflow***, meaning a stack frame extends beyond the memory region allocated for stack, Stack overflow usually causes the program to crash and report an error like: stack overflow error or stack overflow exception.

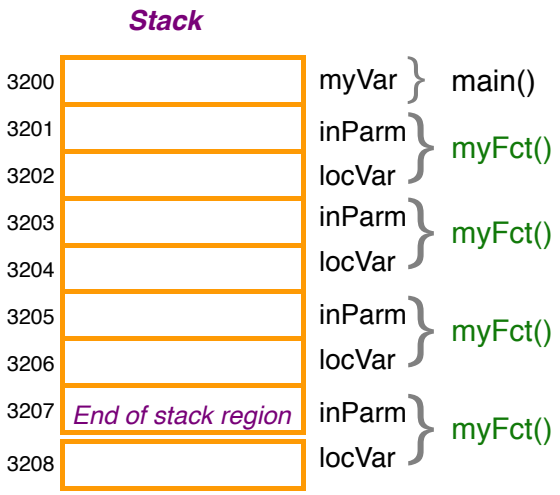
Participation Activity

12.8.1: Recursion causing stack overflow.

Start

```
public static void myFct(int inParm) {
   int locVar;
   ...
   myFct(...);
   ...
   return;
}

public static void main (String[] args) {
   int myVar;
   myFct(...);
   ...
}
```



Deep recursion may cause stack overflow, causing program to crash

The animation showed a tiny stack region for easy illustration of stack overflow.

The number (and size) of parameters and local variables results in a larger stack frame. Large ArrayLists, arrays, or Strings defined as local variables can lead to faster stack overflow.

A programmer can estimate recursion depth and stack size to determine whether stack overflow might occur. Sometime a non-recursive algorithm must be developed to avoid stack overflow.

Participation Activity | 12.8.2: Stack overflow.

#	Question	Your answer
1	A memory's stack region can store at most one stack frame.	True
		False
2	The size of the stack is unlimited.	True
		False
3	A stack overflow occurs when the stack frame for a method call extends past the end of the stack's memory.	True
		False

4	The following recursive method will result in a stack overflow.
	`int recAdder(int inValue) {`
	`  return recAdder(inValue + 1);`
	`}`
		True

		False

Section 12.9 - Java example: Recursively output permutations

Participation

Participation Activity

12.9.1: Recursively output permutations.

The below program prints all permutations of an input string of letters, one permutation per line. Ex: The six permutations of "cab" are:

```
cab
cba
acb
abc
bca
bac
```

Below, the permuteString method works recursively by starting with the first character and permuting the remainder of the string. The method then moves to the second character and permutes the string consisting of the first character and the third through the end of the string, and so on.

1. Run the program and input the string "cab" (without quotes) to see that the above output is produced.
2. Modify the program to print the permutations in the opposite order, and also to output a permutation count on each line.
3. Run the program again and input the string cab. Check that the output is reversed.
4. Run the program again with an input string of abcdefg. Why did the program take longer to produce the results?

Reset

```
1  import java.util.Scanner;
2
3  public class Permutations {
4     // FIXME: Use a static variable to count permutations. Why must it be static?
5
6     public static void permuteString(String head, String tail) {
7        char current = '?';
8        String newPermute = "";
9        int len = tail.length();
10       int i = 0;
11
12       if (len <= 1) {
13          // FIXME: Output the permutation count on each line too
14          System.out.println(head + tail);
15       }
16       else {
17          // FIXME: Change the loop to output permutations in reverse order
18          for (i = 0; i < len; ++i) {
19             current = tail.charAt(i);           // Get next leading character
```

```
cab
```

Run

Participation Activity

12.9.2: Recursively output permutations (solution).

Below is the solution to the above problem.

Reset

```
1  import java.util.Scanner;
2
3  public class PermutationsSolution {
4     static int permutationCount = 0;
5
6     public static void permuteString(String head, String tail) {
7        char current = '?';
8        String newPermute = "";
9        int len = tail.length();
10       int i = 0;
11
12       if (len <= 1) {
13          ++permutationCount;
14          System.out.println(permutationCount + ") " + head + tail);
15       }
16       else {
17          for (i = len - 1; i >= 0; --i) {
18             current = tail.charAt(i);           // Get next leading character
19             newPermute = tail.substring(0, i) + tail.substring(i + 1);
```

cab
abcdefg

Run