Print Copy

Print functionality is for user convenience only, not all interactive content is fully printable.

This print version is for subscriber's personal use only. Not to be posted or distributed.

Print Cancel
Rendering chapter...

©zyBooks 05/10/21 13:25 728163
Neha Maddali
IASTATECOMS228Spring2021

24.1 Recursion: Introduction

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

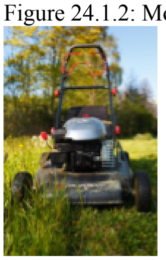
Figure 24.1.1: Algorithms are like recipes.

Make lemonade:

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

Some problems can be solved using a recursive algorithm. A recursive algorithm is an algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.

Figure 24.1.2: 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

The mowing algorithm consists of applying the mowing algorithm on smaller pieces of the yard and thus is a recursive algorithm.

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
24.1.1: Recursion.

Which are recursive definitions/algorithms?

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

24.2 Recursive methods

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

participation activity
24.2.1: A recursive method example.

Animation captions:

1. The first call to countDown( ) method comes from main. Each call to countDown( ) effectively creates a new "copy" of the executing method, as shown on the right.
2. Then, the countDown( ) function calls itself. countDown(1) similarly creates a new "copy" of the executing method.
3. countDown( ) method calls itself once more.
4. That last instance does not call countDown( ) again, but instead returns. As each instance returns, that copy is deleted.

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.

Recursion may be direct, such as f() itself calling f(), or indirect, such as f() calling g() and g() calling f().

participation activity
24.2.2: Thinking about recursion.

Refer to the above countDown example for the following.

1)
How many times is countDown() called if main() calls CountDown(5)?

Check Show answer
2)
How many times is countDown() called if main() calls CountDown(0)?

Check Show answer
3)
Is there a difference in how we define the parameters of a recursive versus non-recursive method? Answer yes or no.

Check Show answer
challenge activity
24.2.1: Calling a recursive method.
Write a statement that calls the recursive method backwardsAlphabet() with parameter startingLetter.

Run

24.3 Recursive algorithm: Search

Recursive search (general)

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 38 (the middle of 26-50)? Lower
- Is it 32 (the middle of 26-38)?

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

participation activity
24.3.1: Binary search: A well-known recursive algorithm.

Animation content:

`undefined`

Animation captions:

1. 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.
2. Using a binary search algorithm, you begin at the midpoint of the lower range. (highVal + lowVal) / 2 = (100 + 0) / 2, or 50.
3. The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
4. After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.
5. A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range.

Recursive search method

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

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.

Figure 24.3.1: A recursive method carrying out a binary search algorithm.

```
import java.util.Scanner;

public class NumberGuessGame {
   public static void guessNumber(int lowVal, int highVal, Scanner scnr) {
      int midVal;            // 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 into lower OR upper half
         if (userAnswer == 'l') {                   // Guess in lower half
            guessNumber(lowVal, midVal, scnr);        // Recursive call
         }
         else {                                     // Guess in upper half
            guessNumber(midVal + 1, highVal, scnr);         // Recursive call
         }
      }
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);

      // 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, scnr);
   }
}
```

```
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): y
Thank you!
```

Calculating the middle value
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.
participation activity
24.3.2: Binary search tree tool.

The following program guesses the hidden number known by the user. Assume the hidden number is 63.

```
import java.util.Scanner;

public class NumberGuessGame {
   public static void guessNumber(int lowVal, int highVal, Scanner scnr) {
      int midVal;            // Midpoint of low..high
      char userAnswer;       // User response

      midVal = (highVal + lowVal) / 2;

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

      if ((userAnswer != 'l') && (userAnswer != 'h')) { // Base case:
         System.out.println("Thank you!");              // Found number
      }
      else { // Recursive case: split into lower OR upper half
         if (userAnswer == 'l') {                 // Guess in lower half
            guessNumber(lowVal, midVal, scnr);      // Recursive call
         }
         else {                                   // Guess in upper half
            guessNumber(midVal + 1, highVal, scnr); // Recursive call
         }
      }
      return;
   }

   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      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).");

      guessNumber(0, 100, scnr);

      return;
```

→ main()

```
public static void main (String[] args) {
   Scanner scnr = new Scanner(System.in);
   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).");

   guessNumber(0, 100, scnr);

   return;
}
```

```
   }
}
```

Recursively searching a sorted list

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.

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.

Figure 24.3.2: Recursively searching a sorted list.

```java
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;        // Midpoint of low and high values
      int itemPos;       // Position where item found, -1 if not found
      int rangeSize;     // 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 found at midVal position
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                                // Base case 2: match not found
         itemPos = -1;
      }
      else {                                                    // Recursive case: search lower or upper half
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower half, recursive call
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal);
         }
         else {                                                 // Search upper half, recursive call
            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 attendees
      String attendeeName;                                       // Name of attendee to match
      int matchPos;                                              // Matched position in attendee list

      // 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.");
      }
   }
}
```

```
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.
```

participation activity
24.3.3: Recursive search algorithm.

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

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?

Check Show answer
2)
If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many calls to findMatch() will be made during the failed search for "Apples"?

Check Show answer
participation activity
24.3.4: 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.

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)

challenge activity
24.3.1: Enter the output of binary search.
Start

Type the program's output

import java.util.Scanner; public class NumberSearch { public static void findNumber(int number, int lowVal, int highVal) { int midVal; midVal = (highVal + lowVal) / 2; System.out.print(number); System.out.print(" "); System.out.print(midVal); if (number == midVal) { System.out.println(" n"); } else { if (number < midVal) { System.out.println(" o"); findNumber(number, lowVal, midVal); } else { System.out.println(" p"); findNumber(number, midVal + 1, highVal); } } } public static void main(String[] args) { Scanner

Input
0

Output

scnr = new Scanner(System.in); int number; number = scnr.nextInt(); findNumber(number, 0, 8); } }

```
0 4 o
0 2 o
0 1 o
0 0 n
```

1	2	3

Check Next

Exploring further:

- Binary search from GeeksforGeeks.org

24.4 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 24.4.1: Output statements can help debug recursive methods, especially if indented based on recursion depth.

```java
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) { // indentAmt used for print debug
      int midVal;         // Midpoint of low and high values
      int itemPos;        // Position where item found, -1 if not found
      int rangeSize;      // 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 found at midVal position
         System.out.println(indentAmt + "Found person.");
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                               // Base case 2: match not found
         System.out.println(indentAmt + "Person not found.");
         itemPos = -1;
      }
      else {                                                   // Recursive case: search lower or upper half
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower half, recursive call
            System.out.println(indentAmt + "Searching lower half.");
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal, indentAmt + "   ");
         }
         else {                                                // Search upper half, recursive call
            System.out.println(indentAmt + "Searching upper half.");
            itemPos = findMatch(stringList, itemMatch, midVal + 1, highVal, indentAmt + "   ");
         }
      }

      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 attendees
      String attendeeName;                                        // Name of attendee to match
```

```
      int matchPos;                                        // Matched position in attendee list

      // 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.");
      }
   }
}
```

```
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
24.4.1: Recursive debug statements.

Refer to the above code using indented output statements.

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

zyDE 24.4.1: 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.

Load default template...
Run

24.5 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 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
24.5.1: Writing a recursive method for factorial: First write the base case, then add the recursive case.

Animation captions:

1. The base case, which returns a value without performing a recursive call, is written and tested first. If N is less than or equal to 1, then the nFact() method returns 1.
2. Next the recursive case, which calls itself, is written and tested. If N is greater than 1, then the nFact() method returns N * nFact(N - 1).

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 calcFactorialHelper(int inVal)`. 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
24.5.2: Creating recursion.
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 24.5.1: Non-recursive solution to compute N!

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

participation activity
24.5.3: When recursion is appropriate.

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

zyDE 24.5.1: 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.

Load default template...

Run

challenge activity
24.5.1: Recursive method: Writing the base case.
Write code to complete doublePennies()'s base case. Sample output for below program with inputs 1 and 10:

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

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

Run

challenge activity
24.5.2: Recursive method: Writing the recursive case.
Write code to complete printFactorial()'s recursive case. Sample output if input is 5:

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

Run

24.6 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 24.6.1: 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);
      }
   }

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

      // Output program description
      System.out.println("This program outputs the\n" +
                         "Fibonacci sequence step-by-step,\n" +
```

```
This program outputs the
Fibonacci sequence step-by-step,
starting after the first 0 and 1.

How many steps would you like? 10
0
1
0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
21 + 34 = 55
34 + 55 = 89
```

```
                          "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);
   }
}
```

zyDE 24.6.1: 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.

Load default template...

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 24.6.2: 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;     // 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;      // First input to GCD calc
      int gcdInput2;      // Second input to GCD calc
      int gcdOutput;      // 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: ");
```

```
This program outputs the greatest
common divisor of two numbers.
Enter first number: 12
Enter second number: 8
Greatest common divisor = 4

...

This program outputs the greatest
common divisor of two numbers.
Enter first number: 456
Enter second number: 784
Greatest common divisor = 8

...

This program outputs the greatest
common divisor of two numbers.
Enter first number: 0
Enter second number: 10
Note: Neither value can be below 1.
```

```
      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);
      }
   }
}
```

participation activity
24.6.1: Recursive GCD example.
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 number from Wolfram.
- Greatest Common Divisor from Wolfram.

challenge activity
24.6.1: Writing a recursive math method.
Write code to complete raiseToPower(). Sample output if userBase is 4 and userExponent is 2 is shown below. Note: This example is for practicing recursion; a non-recursive method, or using the built-in method pow(), would be more common.

```
4^2 = 16
```

Run

24.7 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
24.7.1: Exploring all possibilities viewed as a tree of choices.

Animation captions:

1. Consider printing all possible combinations of a word's letters. Those possibilities can be listed by choosing the first letter, then the second letter, then the third letter.
2. The choices can be depicted as a tree. Each level represents a choice.
3. A recursive exploration function is a natural match to print all possible combinations of a string's letters. Each call to the function chooses from the set of unchosen letters, continuing until no unchosen letters remain.

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 24.7.1: 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;                 // 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);
         }
      }
   }

   // 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 location remLoc
      finalStr += origStr.substring(remLoc + 1, origStr.length()); // Copy after location remLoc

      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 location addLoc
      finalStr += insertStr;                                   // Copy character to location addLoc
      finalStr += origStr.substring(addLoc, origStr.length()); // Copy after location addLoc

      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, "");
   }
}
```

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

```
tca
tac
```

participation activity
24.7.2: Letter scramble.
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 24.7.2: Shopping spree in which a user can fit 3 items in a shopping bag.

GroceryItem.java:

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

```
Milk  Belt  Toys  = $45
Milk  Belt  Cups  = $38
Milk  Toys  Belt  = $45
Milk  Toys  Cups  = $33
Milk  Cups  Belt  = $38
Milk  Cups  Toys  = $33
Belt  Milk  Toys  = $45
Belt  Milk  Cups  = $38
Belt  Toys  Milk  = $45
Belt  Toys  Cups  = $55
Belt  Cups  Milk  = $38
Belt  Cups  Toys  = $55
Toys  Milk  Belt  = $45
Toys  Milk  Cups  = $33
Toys  Belt  Milk  = $45
Toys  Belt  Cups  = $55
Toys  Cups  Milk  = $33
Toys  Cups  Belt  = $55
Cups  Milk  Belt  = $38
Cups  Milk  Toys  = $33
Cups  Belt  Milk  = $38
Cups  Belt  Toys  = $55
Cups  Toys  Milk  = $33
Cups  Toys  Belt  = $55
```

ShoppingSpreeCombinations.java:

```
import java.util.ArrayList;

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

   /* 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,          // Bag contents
                                              ArrayList<GroceryItem> remainingItems) { // Available items
      int bagValue;                // Cost of items in shopping bag
      GroceryItem tmpGroceryItem; // Grocery item to add to bag
      int i;                       // Loop index

      if (currBag.size() == MAX_SHOPPING_BAG_SIZE) {   // Base case: Shopping bag full
         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);
         }
      }
   }

   public static void main(String[] args) {
      ArrayList<GroceryItem> possibleItems = new ArrayList<GroceryItem>(); // Possible shopping items
      ArrayList<GroceryItem> shoppingBag = new ArrayList<GroceryItem>();   // Current shopping bag
      GroceryItem tmpGroceryItem;                                          // Temp item

      // 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);
   }
}
```

participation activity
24.7.3: All letter combinations.

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)
Just before 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 24.7.3: Find distance of traveling to 3 cities.

```java
import java.util.ArrayList;

public class TravelingSalesmanPaths {
   public static final int NUM_CITIES = 3;                                 // Number of cities
   public static int[][] cityDistances = new int[NUM_CITIES][NUM_CITIES]; // Distance between cities
   public static String[] cityNames = new String[NUM_CITIES];             // City names

   /* 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;     // Total distance given current path
      int tmpCity;       // Next city distance
      int i;             // 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);
         }
      }
   }

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

      // 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);
    }
}
```

```
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
24.7.4: Recursive exploration.
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()

challenge activity
24.7.1: Enter the output of recursive exploration.
Start

Type the program's output

import java.util.Scanner; import java.util.ArrayList; public class NumScrambler { public static void scrambleNums(ArrayList<Integer> remainNums, ArrayList<Integer> scramNums) { ArrayList<Integer> tmpRemainNums; int tmpRemovedNum; int i; if (remainNums.size() == 0) { System.out.print(scramNums.get(0)); System.out.print(scramNums.get(1)); System.out.println(scramNums.get(2)); } else { for (i = 0; i < remainNums.size(); ++i) { tmpRemainNums = new ArrayList<Integer>(remainNums); // Make a copy. tmpRemovedNum = tmpRemainNums.remove(i); scramNums.add(tmpRemovedNum); scrambleNums(tmpRemainNums, scramNums); scramNums.remove(scramNums.size() - 1); } } } public static void main(String[] args) { Scanner scnr = new Scanner(System.in); ArrayList<Integer> numsToScramble = new ArrayList<Integer>(); ArrayList<Integer> resultNums = new ArrayList<Integer>(); numsToScramble.add(2); numsToScramble.add(1); numsToScramble.add(5); scrambleNums(numsToScramble, resultNums); } }

```
215
251
125
152
521
512
```

1	2

Check Next

Exploring further:

- Recursive Algorithms from khanacademy.org

24.8 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
24.8.1: Recursion causing stack overflow.

Animation captions:

1. Deep recursion may cause stack overflow, causing a 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 declared 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. Sometimes a non-recursive algorithm must be developed to avoid stack overflow.

participation activity
24.8.2: Stack overflow.
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

24.9 Java example: Recursively output permutations

zyDE 24.9.1: 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 abcdef. Why did the program take longer to produce the results?

Load default template...

cab

Run

zyDE 24.9.2: Recursively output permutations (solution).

Below is the solution to the above problem.

Load default template...

```
cab
abcdef
```

Run

24.10 LAB: Fibonacci sequence (recursion)

This section's content is not available for print.

24.11 LAB: All permutations of names

This section's content is not available for print.

24.12 LAB: Number pattern

This section's content is not available for print.