CSI1102
Introduction to Software Design

Chapter 11:
Recursion

Chapter 11: Recursion

- Recursion is a fundamental programming technique that can provide elegant solutions to certain kinds of problems
- Learning objectives:
  - Learn to think in a recursive manner
  - Learn to program in a recursive manner
  - Understand the correct use of recursion versus iteration
  - Understand the examples using recursion
  - Know about fractals

2

What is recursion: Recursive Thinking

- *Recursion* is a programming technique in which **a method can call itself to solve a problem**
- Before applying recursion to programming, it is best to practice thinking recursively
- Consider the following list of numbers:
  `24, 88, 40, 37`
- A list can be defined recursively as?

3

Recursive Definition of a List

- A list can be defined recursively as

```
A LIST is a:  number
      or a:  number  comma  LIST
```

- That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST
- The concept of a LIST is used to define itself

4

Recursive Definitions

- The recursive part of the LIST definition is used several times, ultimately terminating with the non-recursive part:

```
number comma LIST
  24     ,   88, 40, 37

             number comma LIST
               88     ,   40, 37

                          number comma LIST
                            40     ,   37

                                       number
                                         37
```

5

Avoiding Infinite Recursion: The Base Case

- All recursive definitions must have a base case, i.e. the non-recursive part
- If they don't, there is no way to terminate the recursive path
- The code of a recursive method must be structured to handle both the base case and the recursive case
- A definition without a non-recursive part causes *infinite recursion*
- *What is the base case in a List?*

6

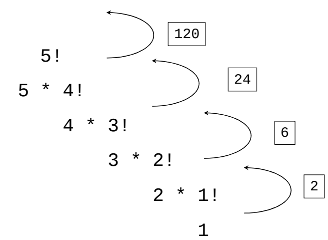
Recursive Definitions of Mathematical equations

- N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive
- This definition can be expressed recursively as:

```
1!  =  1
N!  =  N * (N-1)!
```

- The concept of the factorial is defined in terms of another factorial **until the base case of 1! is reached**

Recursive Definitions: N!



Recursive Programming: Another example

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N, inclusive
- This problem can be expressed recursively as:

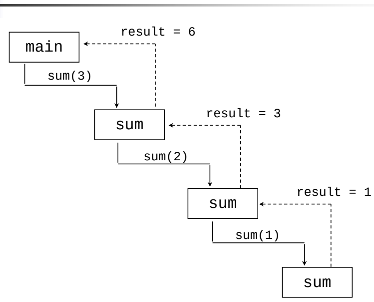
$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

$$= \ldots = N + (N-1) + (N-2) + \ldots + 2 + 1$$

Recursive Programming: Calculating the sum

```
public int sum (int num)
  {
     int result;
     if (num == 1)
          result = 1;
     else
          result = num + sum (num - 1);
     return result;
  }
```


Recursive Programming: Program Execution



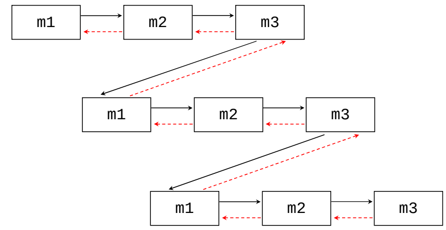
Recursion vs. Iteration

- Just because we can use recursion to solve a problem, doesn't mean we should
- For example, the sum (or the product) of the numbers between 1 and any positive integer N can be calculated with a for loop
- You must be able to determine **when recursion** is the correct technique to use

Indirect Recursion

- A method invoking itself is considered to be *direct recursion*
- A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again
- For example, method `m1` could invoke `m2`, which invokes `m3`, which in turn invokes `m1` again until a base case is reached
- This is called *indirect recursion*, and requires all the same care as direct recursion

Indirect Recursion



An example of recursion: Maze Traversal

- We can use recursion to find a path through a maze; a path can be found from any location if a path can be found from any of the location's neighboring locations
- At each location we encounter, we mark the location as "visited" and we attempt to find a path from that location's "unvisited" neighbors
- Recursion keeps track of the path through the maze
- The base cases are an prohibited move or arrival at the final destination

Maze Traversal: Maze Grid and Output

- The Grid

```
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 0 0 1
0 0 0 0 1 0 1 0 1 0 1 0 0
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1
```

- The output
  The maze was successfully traversed!

```
7 7 7 0 1 1 0 0 0 1 1 1 1
3 0 7 7 7 0 7 7 7 1 0 0 1
0 0 0 0 7 0 7 0 7 0 3 0 0
7 7 7 0 7 7 7 0 7 0 3 3 3
7 0 7 0 0 0 0 7 7 3 0 0 3
7 0 7 7 7 7 7 7 0 3 3 3 3
7 0 0 0 0 0 0 0 0 0 0 0 0
7 7 7 7 7 7 7 7 7 7 7 7 7
```


Maze Traversal: MazeSearch.java

```
public class MazeSearch
{
   public static void main (String[] args)
   {
      Maze labyrinth = new Maze();

      System.out.println (labyrinth);

      if (labyrinth.traverse (0, 0))
         System.out.println ("The maze was successfully traversed!");
      else
         System.out.println ("There is no possible path.");

      System.out.println (labyrinth);
   }
}
```


Maze Traversal: Maze.java

```
public class Maze
{
      private final int TRIED = 3;
      private final int PATH = 7;

      private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                               {1,0,1,1,1,0,1,1,1,1,0,0,1},
                               {0,0,0,0,1,0,1,0,1,0,1,0,0},
                               {1,1,1,0,1,1,1,0,1,0,1,1,1},
                               {1,0,1,0,0,0,0,1,1,1,0,0,1},
                               {1,0,1,1,1,1,1,1,0,1,1,1,1},
                               {1,0,0,0,0,0,0,0,0,0,0,0,0},
                               {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

Continued...

Maze Traversal: Maze.java (cont.)

```
public boolean traverse (int row, int column)
{
      boolean done = false;

      if (valid (row, column))
      {
         grid[row][column] = TRIED;  // this cell has been tried

         if (row == grid.length-1 && column == grid[0].length-1)
            done = true;  // the maze is solved
         else
         {
            done = traverse (row+1, column);     // down
            if (!done)
               done = traverse (row, column+1);  // right
            if (!done)
               done = traverse (row-1, column);  // up
            if (!done)
               done = traverse (row, column-1);  // left
         }

         if (done)  // this location is part of the final path
            grid[row][column] = PATH;
      }
      return done;   }
```

Continued...

Maze Traversal: Maze.java (cont.)

```
//-----------------------------------------------------------------
//  Determines if a specific location is valid.
//-----------------------------------------------------------------
private boolean valid (int row, int column)
{
   boolean result = false;

   // check if cell is in the bounds of the matrix
   if (row >= 0 && row < grid.length &&
       column >= 0 && column < grid[row].length)

      //  check if cell is not blocked and not previously tried
      if (grid[row][column] == 1)
         result = true;

   return result;
}
```

Continued...

Maze Traversal: Maze.java

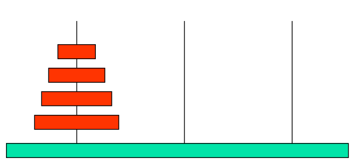
```
   //-----------------------------------------------------------------
   //  Returns the maze as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      String result = "\n";

      for (int row=0; row < grid.length; row++)
      {
         for (int column=0; column < grid[row].length; column++)
            result += grid[row][column] + "";
         result += "\n";
      }

      return result;
   }
}
```


Classic recursive problem: Towers of Hanoi

- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on the pegs
- The goal is to move all of the disks from one peg to another according to the following rules:
  - We can move only one disk at a time
  - We cannot place a larger disk on top of a smaller disk
  - All disks must be on some peg except for the disk in transit between pegs

Towers of Hanoi

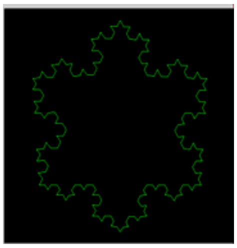
- We use 3 pegs to accomplish this task
- See p. 616 of the text book

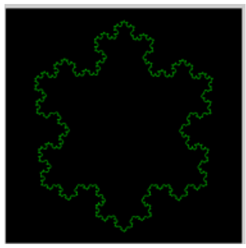


Recursion in Graphics: Fractals

- A *fractal* is a geometric shape than can consist of the same pattern repeated in different scales and orientations
- The *Koch Snowflake* is a particular fractal that begins with an equilateral triangle
- To get a higher order of the fractal, the middle of each edge is replaced with two angled line segments

Fractals:
Modeling Chaos et cetera

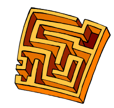




25

Summary:
Chapter 11

- Learning objectives:
  - Learn to think in a recursive manner
  - Learn to program in a recursive manner
  - Understand the correct use of recursion versus iteration
  - Understand the examples using recursion
  - Know about fractals



26