CSC220 (CSI)
Computational Problem Solving

Recursion

The College of New Jersey

Please turn off your cell phone!

Recursive Thinking

- A recursive definition is one which uses the word or concept being defined in the definition itself
- Consider the following list of numbers:

  24, 88, 40, 37

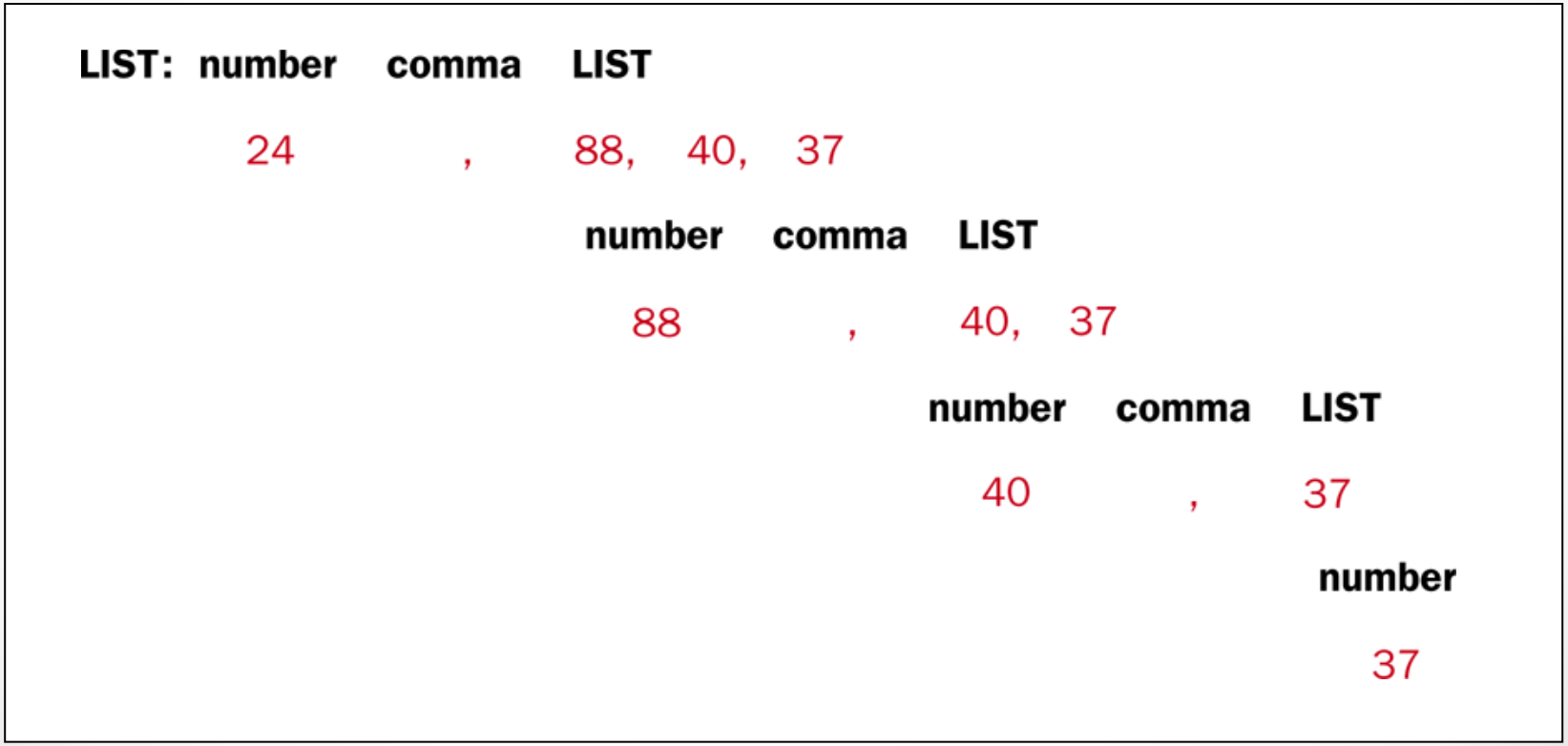
- Such a list can be defined as follows:

  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

Recursive Definitions

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



Infinite Recursion

- All recursive definitions have to have a non-recursive part called the base case
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause infinite recursion
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself

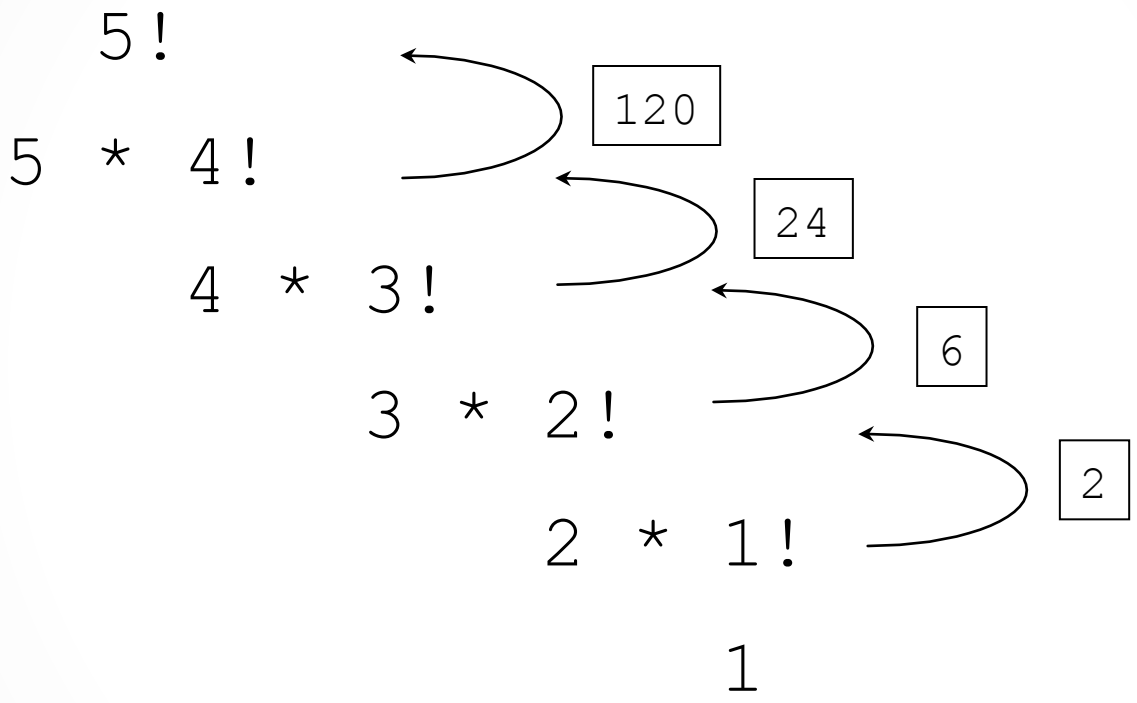
Recursive Factorial

- 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)!
```

- A factorial is defined in terms of another factorial
- Eventually, the base case of 1! is reached

Recursive Factorial



Quick Check

Write a recursive definition of 5 * n, where n > 0.

Recursive Programming

- A recursive method is a method that invokes itself
- A recursive method must be structured to handle both the base case and the recursive case
- Each call to the method sets up a new execution environment, with new parameters and local variables
- As with any method call, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)

Sum of 1 to N

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

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

$$= N + N - 1 + N - 2 + \sum_{i=1}^{N-3} i$$

$$\vdots$$

$$= N + N - 1 + N - 2 + \cdots + 2 + 1$$

Sum of 1 to N

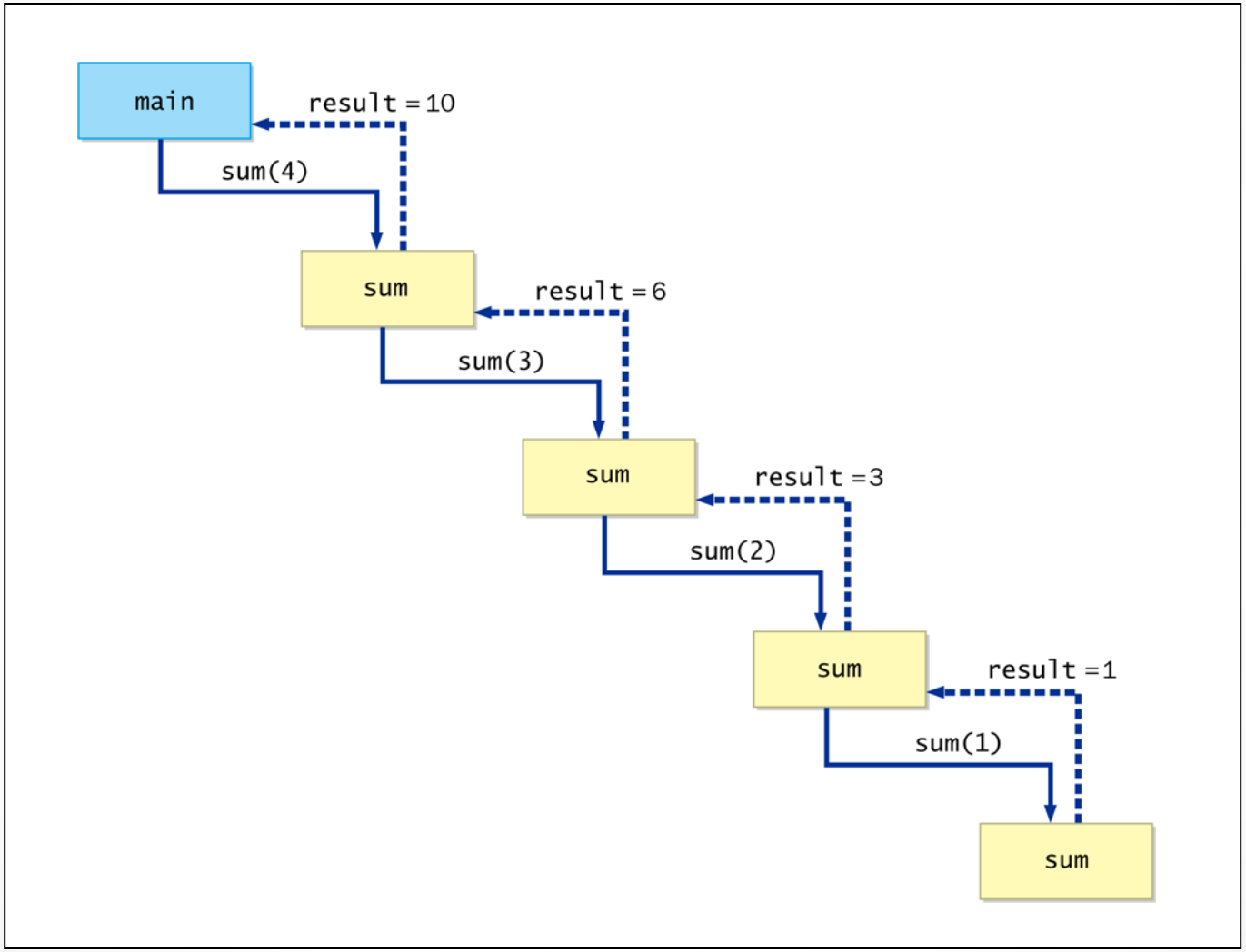
- The summation could be implemented recursively as follows:

```
// This method returns the sum of 1 to num
public int sum(int num)
{
   int result;

   if (num == 1)
      result = 1;
   else
      result = num + sum(num-1);

   return result;
}
```

Sum of 1 to N


main
result = 10
sum(4)
sum
result = 6
sum(3)
sum
result = 3
sum(2)
sum
result = 1
sum(1)
sum

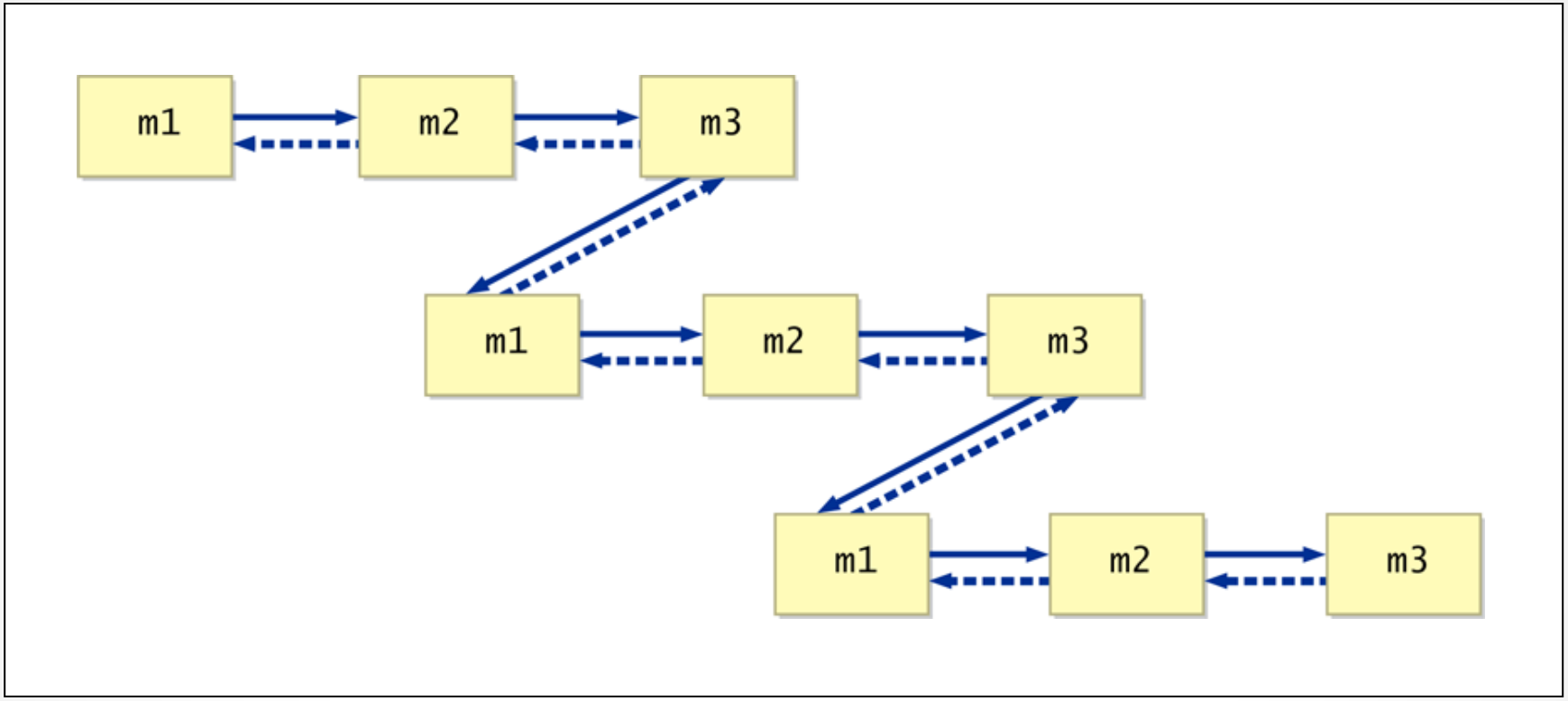
Recursive Programming

- Note that just because we can use recursion to solve a problem, doesn't mean we should
- We usually would not use recursion to solve the summation problem, because the iterative version is easier to understand
- However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version
- You must carefully decide whether recursion is the correct technique for any problem

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… This is called indirect recursion, and requires all the same care as direct recursion
  - For example, method `m1` could invoke `m2`, which invokes `m3`, which in turn invokes `m1` again
- It is often more difficult to trace and debug

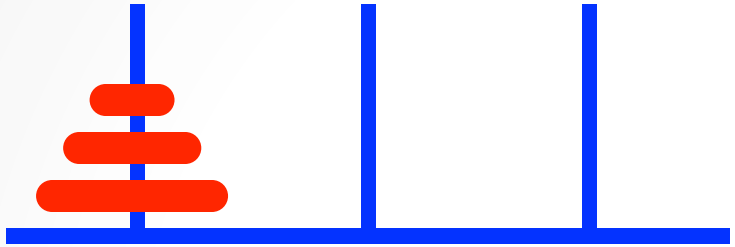
Indirect Recursion



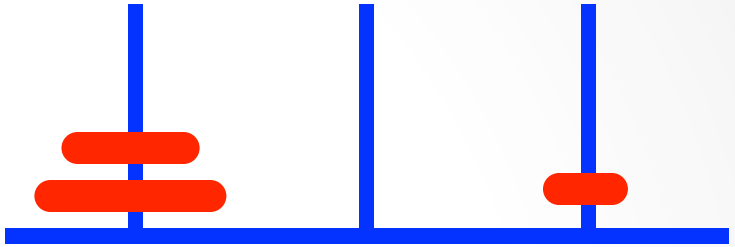
Towers of Hanoi

- The Towers of Hanoi is a puzzle made up of three vertical pegs and several disks that slide onto the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another under the following rules:
  - Move only one disk at a time
  - A larger disk cannot be put on top of a smaller one

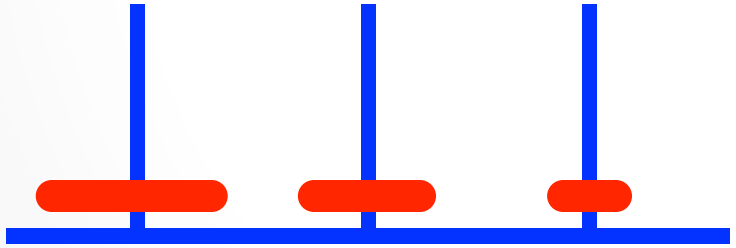
Towers of Hanoi



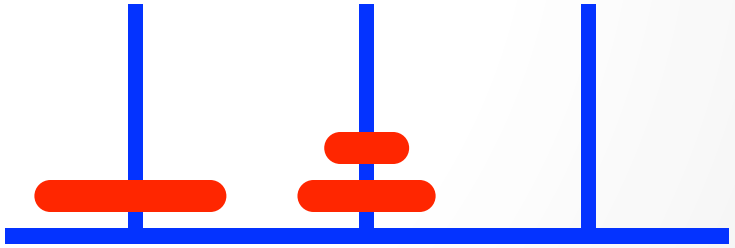
Original Configuration



Move 1

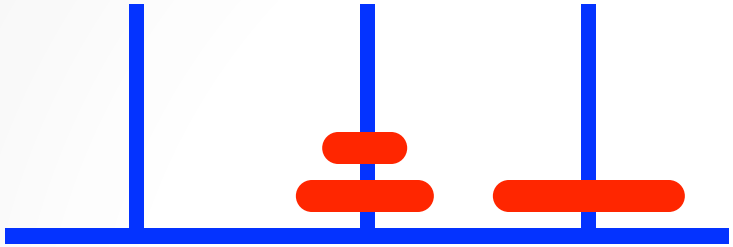


Move 2

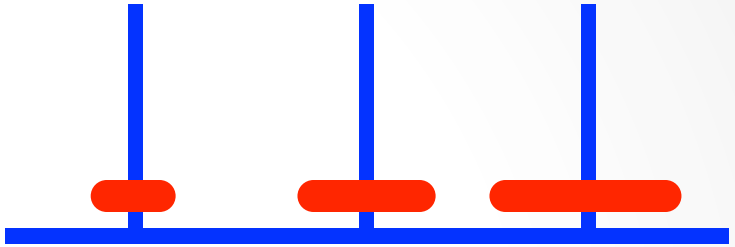


Move 3

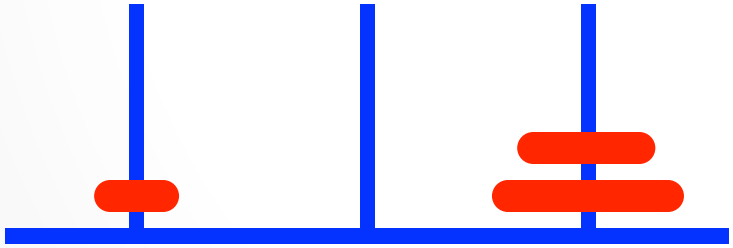
Towers of Hanoi



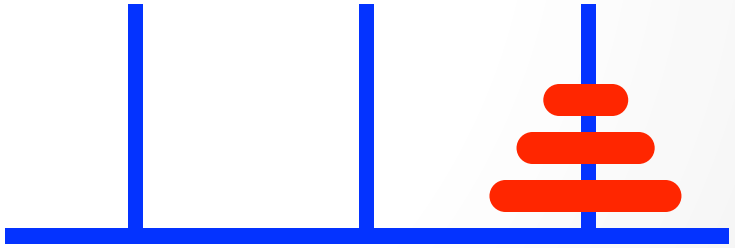
Move 4



Move 5



Move 6



Move 7 (done)

Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant

```
//********************************************************************
//  SolveTowers.java       Author: Lewis/Loftus
//  Demonstrates recursion.
//********************************************************************

public class SolveTowers {
   //-----------------------------------------------------------------
   //  Creates a TowersOfHanoi puzzle and solves it.
   //-----------------------------------------------------------------
   public static void main(String[] args)   {
      TowersOfHanoi towers = new TowersOfHanoi(4);
      towers.solve();
   }
}
```

```
//********************************************************************
//  SolveTowers.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

public class SolveTowers{
   //  Creates a TowersOfHanoi puzzle and solves it.
   public static void main (String[] args){
      TowersOfHanoi towers = new TowersOfHano
      towers.solve();
   }
}
```

Output

```
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 3 to 1
Move one disk from 3 to 2
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 2 to 1
Move one disk from 3 to 1
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
```

```
//********************************************************************
//  TowersOfHanoi.java       Author: Lewis/Loftus
//
//  Represents the classic Towers of Hanoi puzzle.
//********************************************************************

public class TowersOfHanoi{
   private int totalDisks;

   //-----------------------------------------------------------------
   //  Sets up the puzzle with the specified number of disks.
   //-----------------------------------------------------------------
   public TowersOfHanoi(int disks) {
      totalDisks = disks;
   }

   //-----------------------------------------------------------------
   //  Performs the initial call to moveTower to solve the puzzle.
   //  Moves the disks from tower 1 to tower 3 using tower 2.
   //-----------------------------------------------------------------
   public void solve(){
      moveTower(totalDisks, 1, 3, 2);
   }
```

continued

continued

```
   //-----------------------------------------------------------------
   //  Moves the specified number of disks from one tower to another
   //  by moving a subtower of n-1 disks out of the way, moving one
   //  disk, then moving the subtower back. Base case of 1 disk.
   //-----------------------------------------------------------------
   private void moveTower(int numDisks, int start, int end, int temp){
      if (numDisks == 1)
         moveOneDisk(start, end);
      else{
         moveTower(numDisks-1, start, temp, end);
         moveOneDisk(start, end);
         moveTower(numDisks-1, temp, end, start);
      }
   }

   //-----------------------------------------------------------------
   //  Prints instructions to move one disk from the specified start
   //  tower to the specified end tower.
   //-----------------------------------------------------------------
   private void moveOneDisk(int start, int end){
      System.out.println("Move one disk from " + start + " to " +
                         end);
   }
}
```

Maze Traversal

- We can use recursion to find a path through a maze
- From each location, we can search in each direction
- The recursive calls keep track of the path through the maze
- The base case is an invalid move or reaching the final destination

```
//********************************************************************
//  MazeSearch.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

public class MazeSearch{
   //-----------------------------------------------------------------
   //  Creates a new maze, prints its original form, attempts to
   //  solve it, and prints out its final form.
   //-----------------------------------------------------------------
   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.java       Author: Lewis/Loftus
//  Represents a maze of characters. The goal is to get from the
//  top left corner to the bottom right, following a path of 1s.
//********************************************************************
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},
                            {1,0,1,1,1,1,1,1,1,0,1,1,1},
                            {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} };

  // Attempts to recursively traverse the maze. Inserts special
  // characters indicating locations that have been tried and that
  // eventually become part of the solution.
  public boolean traverse (int row, int column) { }

  //  Returns the maze as a string.
  public String toString() {}

}
```

```
//********************************************************************
//  Maze.java       Author: Lewis/Loftus
//  Represents a maze of characters. The goal is to get from the
//  top left corner to the bottom right, following a path of 1s.
//********************************************************************
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},
                            {1,0,1,1,1,1,1,1,1,0,1,1,1,1},
                            {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} };
   //  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

```
// Attempts to recursively traverse the maze. Inserts special
// characters indicating locations that have been tried and that
// eventually become part of the solution.
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

continued

```
    //-----------------------------------------------------------------
    //  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;
    }
}
```

```
//**********                                                  **********
//  MazeSear
//
//  Demonstr                                                  **********
//**********

public class                                                 ----------
   //-------                                                  mpts to
   //  Creat
   //  solve                                                  ----------
   //-------
   public st
   {
      Maze l

      System

      if (la
         Sys                                                 raversed!");
      else
         Sys                                                 );

      System
   }
}
```

Output

```
1110110001111
1011101111001
0000101010100
1110111010111
1010000111001
1011111101111
1000000000000
1111111111111

The maze was successfully traversed!

7770110001111
3077707771001
0000707070300
7770777070333
7070000773003
7077777703333
7000000000000
7777777777777
```