Outline

• Basic recursion concept
• Using recursion in maze solver
• Using recursion in hanoi tower solver
Recursion

- Recursion: a process or a pattern of repeating items in a self-similar way, e.g.

- two parallel mirrors facing each other, the mirrored image is a recursion

- Recursive definition of Fibonacci number:

\[
 f(n) = \begin{cases} 
 0, & n = 0 \\
 1, & n = 1 \\
 f(n - 1) + f(n), & n > 1 
\end{cases}
\]
Recursion Method a Self-calling Method

In programming, a recursive method is a method that calls itself.

```
static int sum(int n) {
    int r = 1;
    if (n > 1)
        r = n + sum(n-1);
    return r;
}
```

Calling itself

- `sum(4)`
  - `r = 4 + 3 + 3 = 10`
- `sum(3)`
  - `r = 3 + 2 + 1 = 6`
- `sum(2)`
  - `r = 2 + 1 = 3`
- `sum(1)`
  - `r = 1`

SimpleRecursion.java
A Case Study: Endless Recursion

```java
static void show() {
    System.out.println("I am recursing");
    show();
}
```

- Each time the method is called, it will always invoke another round the same method
- Is there a terminator? NO!
The Two Cases in a Recursion

static int sum( int n ) {
    if (n < 1)
        return 0;
    else if (n == 1)
        return 1;
    else
        return n + sum( n-1 );
}

• **Base case**: terminators, return concrete value to stop the recursion

• **Recursive case**: invoke self, the built-in implicit loop, make sure the invocation gear towards the base case
Recursion vs. Iteration

- Do you prefer to read code in loop or in recursion?
- Every recursion solution has a corresponding iterative solution
- Recursion is expensive if it involves a long loop. Method invocation costs extra maintenance
- So use recursion only when it significantly simplifies the implementation! See the following two examples.
The Maze Solver Problem

- Find a path that connects from start to end with the each move in any of the four possible directions: left, right, up, or down

```java
int[][] maze = {
{ 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 }  
};
```
Think Recursively

- At any grid inside the maze, isPath(row, col) returns true if the grid at maze[row][col] is on the path, false otherwise.

- First assume the grid is not on the path, initialize the result to false.

- If the grid is a passage, mark the grid as visited, and

- If the grid is the exit, update the result to true.

- Otherwise update the result by isPath(row-1, col), isPath(row+1, col), isPath(row, col-1), or isPath(row, col+1).

- Now if the result is true, mark the grid as path.

- Return the result.
Maze Solver Implementation

MazeSolver.java

```java
public static boolean mazeSolver(int[][] maze, int row, int col) {
    boolean done = false;
    if (row >= maze.length || col >= maze[0].length || row < 0 || col < 0)
        return done;
    if (maze[row][col] != 1)
        return done;
    // mark current one as visited
    maze[row][col] = 2;
    if (row == maze.length - 1 && col == maze[0].length - 1)
        done = true;
    else {
        done = mazeSolver(maze, row, col + 1); // right
        if (!done) done = mazeSolver(maze, row, col - 1); // up
        if (!done) done = mazeSolver(maze, row + 1, col); // left
        if (!done) done = mazeSolver(maze, row - 1, col); // down
    }
    if (done) {
        maze[row][col] = 3; // path
        System.out.printf("maze[%d][%d]\n", row, col);
    }
    return done;
}
```

base case I: invalid grid
base case II: grid of wall, or visited passage
base case III: the exit grid, the most important terminator
recursive cases
The Hanoi Tower Problem

- Move a stack of $N$ disks from the original peg to a destination peg with one additional peg following the rules:
  - only one disk may be moved at a time
  - a disk cannot be placed on top of a smaller disc
  - all discs must be stored on a peg except while being moved
Solving a 3-disk Hanoi Tower

- Four factors in consideration: Disk number, peg index from, peg index to, extra peg index
Think Recursively

• Think iteratively, very hard!
• Think recursively,
  • move the top N-1 disks from the source peg to an extra peg (recursive case)
  • move the Nth disk from the source peg to the destination peg (base case)
  • move the top N-1 disks from the extra peg to the destination peg (recursive case)
The Hanoi Tower Solver

```java
public class HanoiTowerSolver {
    public static int totalSteps = 0;

    public static void hanoiTowerSolver(int n, int from, int to, int extra) {
        if (n > 0) {
            hanoiTowerSolver(n-1, from, extra, to);
            System.out.printf("move disk%d from peg%d to peg%d\n", n, from, to);
            totalSteps ++;
            hanoiTowerSolver(n-1, extra, to, from);
        }
    }

    public static void main(String[] args) {
        System.out.println("How many disks in total?");    
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();

        totalSteps = 0;
        System.out.println("Here is how to solve the hanoi tower:" );
        hanoiTowerSolver(n, 1, 3, 2);
        System.out.println("You need a total of "+totalSteps+" moves.");
    }
}
```

HanoiTowerSolver.java
An Iterative Hanoi Tower Solver

- Both recursive methods use multiple input parameters
- Both hold multiple recursive calls
- Both significantly simplify the implementation compared to direct loop
- An n-disk Hanoi tower involves $2^{N-1}$ moves solved in 3 steps recursively
Summary

- A Recursive method is a method that calls itself within its process
- The recursive method calls brings self maintained loops
  - Control the loop termination by base cases
  - Control the loop continuation by recursive cases
- Use recursion when direct loop cannot be easily formed as shown in the maze and hanoi tower solvers