Outline

- binary search
- more examples of recursion
- problem solving using recursion
- Towers of Hanoi
- backtracking
- garbage collection

Aside: Binary Search in real life

- suppose you wanted to find out the maximum depth of recursion of your implementation of Java
  - i.e., for what value of n does this crash?

```
static void recurse (int n) {
    if (n <= 0)
        return;
    recurse (n - 1);
}
```

- can start with a large number that crashes
- then keep testing with the number / 2
- eventually, binary search finds the right value

Example of Recursion: Printing Integers

- print an integer in binary
- must print the most significant bit first
  - but easiest to access the least significant bit!
- very easy to do recursively:

```
private static void printBinary(int toPrint) {
    if (toPrint < 2) {              /* first (or only) bit */
        System.out.print(toPrint);
    } else {
        /* print the digits before this one */
        printBinary(toPrint / 2);
        /* print last bit */
        System.out.print(toPrint % 2);
    }
}
```

- this code reaches the base case (toPrint < 2) before printing anything
- once the base case is reached, the remaining digits are printed after the inner call completes

In-class Exercise

- write a recursive method
- to print integers
- printing a comma every 3 digits
- for example, 1,234,567,890

Computing Fibonacci Numbers

- the fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, ...
- the n-th fibonacci number fib(n) is simply defined as fib(n-1) + fib(n-2)
- this is clearly a recursive definition
- fib(1) == fib(2) == 1

```
static int fib(int n) {
     if (n <= 2) {
        return 1;
     }
    return fib(n - 1) + fib(n - 2);
 }
```

- the only problem is this is very inefficient: computing fib(100) must compute fib(99) and fib(98), but computing fib(99) also recomputes fib(98), as well as fib(97)
- there are ways around this -- instead of computing from the large numbers, compute the small values first

Efficient Computation of Fibonacci Numbers: recursively

- the efficient way to compute fibonacci numbers is to start low and use the previous results in computing the new value
- this can be done either recursively or in a loop

```
public static int fib(int n) {
    if (n < 2) {
        return 1;
    }
    fibHelper(1, 1, n - 2);
}
private static int fibHelper(int first, int second, int n) {
    if (n == 0) {  // base case, end of recursion
        return first + second;
    }
    return fibHelper(second, first + second, n - 1);
}
```

- this recursive solution only takes linear time

Efficient Computation of Fibonacci Numbers: iteratively

```
public static int fib(n) {
   int first = 1;
   int second = 1;
   while (n >= 2) {
      int third = first + second;
      first = second;
      second = third;
      n--;
    }
    return second;
}
```

- the iterative solution also only takes linear time
- in-class exercise: convince yourself that the iterative and recursive solutions both compute the sequence 1, 1, 2, 3, 5, 8, 13, 21, ... (or find any bugs)

Problem Solving with Recursion

- many classical puzzles have recursive solutions
- for example, there is a game, Towers of Hanoi, which has three pegs and a number of discs of varying size
  - each disk can go on any peg
  - but a disk must be smaller than any disks below it
- the game starts with all the disks on one peg
- disks are moved one at a time
- any disk D can be moved to any peg on which the topmost disk is larger than D
- the objective of the game is to move all the disks from one peg to another
- see https://en.wikipedia.org/wiki/File:Tower_of_Hanoi.jpeg for an example of the actual game
- it has been said that there are monks in a monastery in Hanoi patiently engaged in moving 64 disks from one peg to another -- when they are done, the world might come to an end
- if this were true, how long would it take for the world to come to an end?

Recursive solution for Towers of Hanoi

- it is easy to move the smallest disk from one peg to another
- it is easy to move any disk D to another peg where the topmost disk is larger than D
- how do we move a set of n disks from peg 1 to peg 2?
- assume (recursively) that I know how to move a set of n-1 disks from peg 1 to peg 3
- and also from peg 3 to peg 2
- then, I do the first step (move the n-1 disks), then move the nth disk from peg 1 to peg 2, and finally move the n-1 disks back from peg 3 to peg 2
- of course, to move the n-1 disks from peg 1 to peg 3, I have to move the top n-2 disks from peg 1 to peg 2, then move the n-1th disk from peg 1 to peg 3, and finally move the n-2 disks from peg 2 to peg 3
- I can continue this until my base case, n == 1, where I can move the disk directly
- This is very easy to do recursively, and much harder to do iteratively
  - See Towers.java

Finding a path through a maze

- finding a path through a maze is easy:
  - at the exit, conclude that the path has been found
  - at a dead end, conclude that the path has not been found
  - whenever there is a choice, recursively explore all choices
  - after marking the current location so we will not explore it again
- a tricky part of this is to find ways to mark a location so we can tell, when we visit it, whether it is:
  - known to be part of the path, or
  - known to have been visited already, or
  - not visited yet
- this requires a few bits/booleans at each position in the maze
- the program in the book represents these using colors

Backtracking

- finding a path through a maze is an example of a technique called **backtracking**
- if we have several choices, and we are not sure which one will work, we try of them, and look at the result
- if the result is not favorable, we try the next choice
- this is done recursively, because we have more choices once we have made the first choice
- this can be useful in some games, such as chess or go
- however, exhaustively searching all possible moves is impractical
- instead, the search goes up to a point, estimates how good the play is at that point, and reports that
- in a two-player game, have to consider my best move in response to my opponent's best move -- my opponent's best move is the one that gives the lowest score, my best move is the one that gives the highest score

Garbage Collection: recycling unused memory

- garbage collection means finding the parts of memory that are not reachable by the program:
  - start with the system stack and all the global variables
  - follow each of the pointers, marking each location as visited
  - follow each reference in each of the objects visited
  - at the end, any location that has not been visited, is free and can be reused
- we need to mark each of the memory areas as being in use or not
- and also whether we've visited it before, otherwise circular structures will be hard to garbage collect