Outline

- principles of recursion
- printing integers
- fibonacci numbers
- problem solving with recursion
- towers of hanoi
- backtracking
- memory management

Reminder: Principles of recursion

- **many problems have:**
  - an obvious solution when they are small (**base case**), and
  - a way to make a big problem into a smaller problem (**recursive case**)
- **if this is the case, we can solve any size problem!**
- **The base case immediately returns the answer we can figure out right away**
  - there may be multiple base cases, for example in binary search one base case is where we find the value, the other base case is where we didn't find the value and the sub-array has size 1
- **The recursive case(s) recursively call the method with parameters such that the problem becomes smaller**
  - for example in binary search, the sub-array is at most half the size of the original array
  - in factorial, the number is one less than the original number

How to be sure that a recursive method terminates

- **prove that every recursive case gets closer to a base case**
- **for example, in binary search:**
  - the base cases are:
    - when the value is found, or
    - first >= last
  - on each recursive call, either first moves towards last, or last moves towards first, by at least one: because middle is not less than first, middle + 1 is bigger than first, and because middle is not greater than last, middle - 1 is less than last
- **in general, to be sure our code terminates we must ensure that the recursive case approaches the base case**
- **for factorial, a correctness depends on whether the base case is (n <= 1) or (n == 1)**
- **if the base case is n==1, when n < 1 the recursion never terminates!**

Proving that recursive factorial terminates

- *!n* (n factorial) is *n * (n – 1) * (n – 2) * … * 2 * 1*
- another way of writing this is to say !n = n * !(n – 1)
- this second formulation is recursive, and gives us an intuitive recursive implementation:

```
public int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n – 1);
}
```

- if n <= 1, the call terminates immediately
- if n > 1, the parameter to the recursive call is closer than n to the base case

Example of recursion: printing integers

- **print an integer in binary (base 2) or in any other base (base 8, base 10)**
- **must print the most significant bit or digit first**
    - but modulo only gives us access the **least significant** bit or digit
- **very easy to do recursively**
- **this code reaches the base case (`toPrint < base`) <u>before</u> printing anything**
- **once the base case is reached, the remaining bits/digits are printed after the inner call completes**

```
private static void
  printInt(int toPrint,int base) {
    if (toPrint < base) {
      // first (or only) bit/digit
      System.out.print(toPrint);
    } else
       /* print the preceding
          bits/digits */
      printInt(toPrint / base, base);
      /* print last bit or digit */
      System.out.print(toPrint % base);
    }
}
```

Computing Fibonacci numbers

- the fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, ...
- the n-th fibonacci number fib(n) is defined as fib(n-1) + fib(n-2)
- this is a recursive definition
- `fib(1) == fib(2) == 1`
- 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), fib(97) is computed 4 times, etc
- there are ways around this -- instead of computing from the large numbers, compute the small values first

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

Efficient computation of Fibonacci numbers: recursively

- to efficiently compute fibonacci numbers, start low and use the previous results in computing the new value
- this can be done either recursively or in a loop
- this recursive solution only takes linear time

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

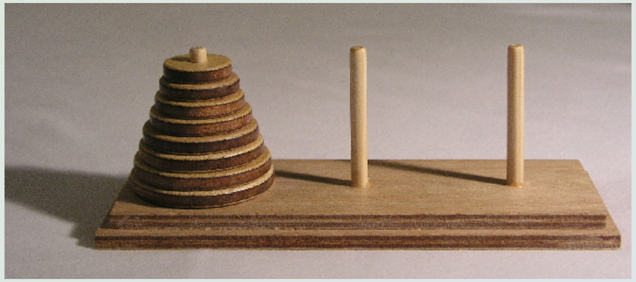
Efficient computation of Fibonacci numbers: iteratively

- to efficiently compute fibonacci numbers, start low and use the previous results in computing the new value
- this 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)

```
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;
}
```

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 disc can go on any peg
  - but each disc can only go on a larger disc
- **the game starts with all the discs on one peg**
- **only the top disc on a peg can move**
- **the objective of the game is to move all the discs from one peg to another**
- **it has been said that there are monks in a monastery in Hanoi patiently engaged in moving 64 discs from one peg to another -- when they finish, the world might come to an end**
- **if this were true, how long would it take for the world to come to an end?**



https://en.wikipedia.org/wiki/File:Tower_of_Hanoi.jpeg, image credit CC-BY-SA: wikipedia User:Evanherk

Recursive solution for Towers of Hanoi

- **how do we move a set of n discs from peg 1 to peg 2?**
- **base case: move the top disc from one peg to another**
- **recursive case:**
    - recursively move n-1 discs from peg 1 to peg 3, using peg 2 as a spare, then
    - move the bottom disc from peg 1 to the empty peg 2, then
    - recursively move n-1 discs from peg 3 to peg 2, using peg 1 as a spare
- **This is very easy to do recursively, and much harder to do iteratively -- see Towers.java**

https://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif

- **image credit CC-BY-SA: wikipedia User:Aka**

Finding a path through a maze

- **finding a path through a maze is easy:**
  - base case: at the exit, conclude that the path has been found
  - base case: at a dead end, conclude that the path has not been found
  - whenever there is a choice, recursively explore all choices
  - however, if the maze has cycles, the recursion might not end

Finding a path through a maze that has cycles

- **if the maze may have cycles, the recursive algorithm may continue forever**
- **so whenever the algorithm visits a location in the maze, it must mark the location, and when it finds a marked location, treat it as a dead end**
- **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 have been visited already, or
  - not visited yet
- **this requires a few bits of information 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 <u>backtracking</u>**
- **if we have multiple choices, we try one of them, and look at the result**
  - the result is from the program **back-tracking** to this choice point
- **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**
- **backtracking can be useful in some games, such as chess or go**
- **however, exhaustively searching all possible moves in chess or go is impractical**
- **instead, the search goes to a point, estimates how good the play is at that point, and reports that result**
- **in a two-player game, I have to consider my best move in response to my opponent's best move**
  - my opponent's best move is the one that gives me the lowest score
  - my best move is the one that gives me 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 all the global variables of the program that refer to objects
  - follow each global reference, marking each object as visited
  - recursively 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 mark each of the memory areas as being in use or not**
- **and this mark also tells us whether we're in a cycle**