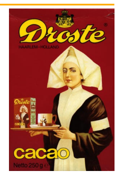
Recursion



Chapter 5.4 in Rosen
Chapter 11 in Savitch

What does this method do?


```
/**
* precondition n>0
* postcondition ??
*/
private void printStars(int n) {
    if (n == 1) {
        System.out.println("*");
    } else {
        System.out.print("*");
        printStars(n - 1);
    }
}
```

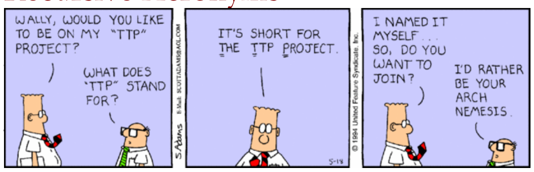
Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.
- **recursive programming**: Writing methods that call themselves
  - directly or indirectly
  - An equally powerful substitute for *iteration* (loops)
  - But sometimes much more suitable for the problem

Definition of recursion

```
recursion: n.
   See recursion.
```

Recursive Acronyms



GNU — GNU's Not Unix
KDE — KDE Desktop Environment
PHP - PHP: Hypertext Preprocessor
PNG — PNG's Not GIF (officially "Portable Network Graphics")
RPM — RPM Package Manager (originally "Red Hat Package Manager")

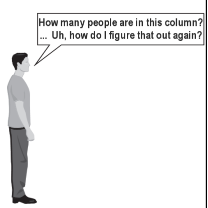
http://search.dilbert.com/comic/Ttp

Why learn recursion?

- A different way of thinking about problems
- Can solve some problems better than iteration
- Leads to elegant, simple, concise code (when used well)
- Some programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

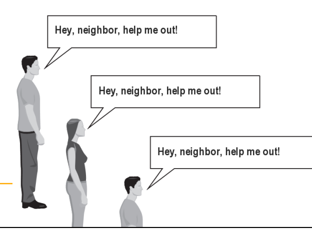
Exercise

- (To a student in the front row)
  How many students are directly behind you?
  - We all have poor vision, and can only see the people right next to us. So you can't just look back and count.
  - But you are allowed to ask questions of the person behind you.
  - How can we solve this problem? (*recursively*)



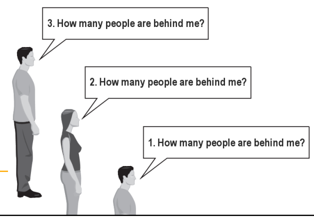
The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help you?



Recursive algorithm

- Number of people behind me:
  - If there is someone behind me,
    ask him/her how many people are behind him/her.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody behind me, I will answer **0**.



Recursive structures

- A directory has
  - files

  and
  - (sub) directories
- An expression has
  - operators
  - operands, which are
    - variables
    - constants
    - (sub) expressions

Expressions represented by trees

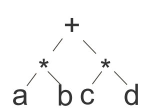
- A tree is
  - a node

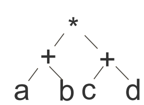
  with
  - zero or more sub trees

examples:

a*b + c*d

(a+b)*(c+d)





Structure of recursion

- Each of these examples has
  - recursive parts (directory, expression, tree)
  - non recursive parts (file, variables, nodes)
- Would we always need non recursive parts?
- Same goes for recursive algorithms.

Cases

- Every recursive algorithm has at least 2 cases:
  - **base case**: A simple instance that can be answered directly.
  - **recursive case**: A more complex instance of the problem that cannot be directly answered, but can instead be described in terms of smaller instances.
  - Can have more than one base or recursive case, but all have at least one of each.
  - A crucial part of recursive programming is identifying these cases.

Base and Recursive Cases: Example

```
public void printStars(int n) {
    if (n == 1) {
        // base case; print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

Recursion Zen

- An even simpler, base case is `n=0`:

```
public void printStars(int n) {
    if (n == 0) {
        // base case; end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

  - **Recursion Zen**: The art of identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Everything recursive can be done non-recursively

```
// Prints a line containing a given number of stars.
// Precondition: n >= 0
public void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

Exercise

- Write a method `reverseLines` that accepts a file `Scanner` prints to `System.out` the lines of the file in reverse order.
  - Write the method recursively and without using loops.
  - Example input:

```
this
is
fun
no?
```

Expected output:

```
no?
fun
is
this
```

  - What are the cases to consider?
    - How can we solve a small part of the problem at a time?
    - What is a file that is very easy to reverse?

Reversal pseudocode

- Reversing the lines of a file:
  - Read a line L from the file.
  - Print the rest of the lines in reverse order.
  - Print the line L.
- If only we had a way to reverse the rest of the lines of the file....

Reversal solution

```java
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case
        String line = input.nextLine();
        reverseLines(input);
        System.out.println(line);
    }
}
```

- Where is the base case?

Tracing our algorithm (Show reverseLines.java)

- **call stack**: The method invocations running at any one time.

```java
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {    // false
        ...
    }
}
```

output:

```
no?
fun
is
this
```

input file:

```
this
is
fun
no?
```

Recursive power example

- Write a method that computes x^n.
  $x^n = x * x * x * ... * x$ (n times)
- An iterative solution:

```
public int pow(int x, int n) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product = product * x;
    }
    return product;
}
```

21

Exercise solution

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public int pow(int x, int n) {
    if (n == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else {
        // recursive case:  x^n = x * x^(n-1)
        return x * pow(x, n-1);
    }
}
```

How recursion works

- Each call sets up a new instance of all the parameters and the local variables
- When the method completes, control returns to the method that invoked it (which might be another invocation of the same method)

```
pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 1
          = 64
```

23

Infinite recursion

- A method with a missing or badly written base case can causes **infinite recursion**

```
public int pow(int x, int y) {
    return x * pow(x, y - 1);  // Oops! No base case
}

pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 4 * pow(4, -1)
          = 4 * 4 * 4 * 4 * 4 * pow(4, -2)
          = ... crashes: Stack Overflow Error!
```

24

An optimization

- Notice the following mathematical property:

$$3^{12} = (3^2)^6 = (9)^6 = (9^2)^3 = (81)^3 = 81 * (81)^2$$

  - How does this "trick" work?
  - Do you recognize it?
  - How can we incorporate this optimization into our `pow` method?
  - What is the benefit of this trick?
  - Go write it.

Exercise solution 2

```java
// Returns base ^ exponent.
// Precondition: exponent >= 0
public int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else if (exponent % 2 == 0) {
        // recursive case 1:  x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {
        // recursive case 2:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

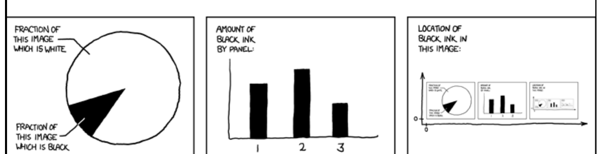
Activation records

- **Activation record**: memory that Java allocates to store information about each running method
  - return point ("RP"), argument values, local variables
  - Java stacks up the records as methods are called; a method's activation record exists until it returns
  - Eclipse debug draws the act. records and helps us *trace* the behavior of a recursive method

```
| x  = [ 4 ]        n = [ 0 ] | pow(4, 0)
| RP = [pow(4,1)]             |
| x  = [ 4 ]        n = [ 1 ] | pow(4, 1)
| RP = [pow(4,2)]             |
| x  = [ 4 ]        n = [ 2 ] | pow(4, 2)
| RP = [pow(4,3)]             |
| x  = [ 4 ]        n = [ 3 ] | pow(4, 3)
| RP = [main]                 |
|                             | main
```

27

More Recursion!



http://xkcd.com/688/

http://xkcd.com/981/

Recursion - examples

- Problem: given a string as input, write it backward
- Base case?
- Recursion

What questions to ask?

- What is a good base case? Perhaps more than one.
- What is the recursive case? Perhaps more than one
  - What are the sub-problems in the recursive case?
  - How are the answers to the sub-problems combined?

What questions to ask?

- Is a helper method needed?
  - With arrays, you may need extra parameter(s) to track the index
  - If you have to return an array, it may be easier to pass a result array of the required size and fill it recursively.

Tail recursion

- Tail recursion is a recursive call that occurs as the last action in a method.
- This is not tail recursion:

```
public int factorial(int n){
    if (n==0)
        return 1;
    return n* factorial(n-1);
}
```

Tail recursion

- This is tail recursion:

```
public int factorial(int n) {
   return factorialTail(n, 1);
}
int factorialTail(int n, int product) {
   if(n == 0)
      return product;
   return factorialTail(n-1, product*n);
}
```

Tail recursion

- This is tail recursion:

```
public int factorial(int n) {
   return factorialTail(n, 1);
}
int factorialTail(int n, int product) {
   if(n == 0)
      return product;
   return factorialTail(n-1, product*n);
}
```

- But why would you care? Turns out that compilers can optimize memory usage when they detect that this is the case.

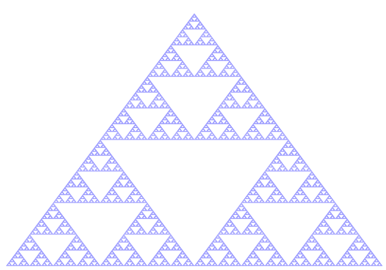
Tail recursion

- This is tail recursion:

```
public int factorial(int n) {
   return factorialTail(n, 1);
}
int factorialTail(int n, int product) {
   if(n == 0)
      return product;
   return factorialTail(n-1, product*n);
}
```

- When making a recursive call, you no longer need to save the information about the local variables within the calling method.

Fractals – the Koch curve and Sierpinski Triangle



Dictionary lookup

- Suppose you're looking up a word in the dictionary (paper one, not online!)
- You probably won't scan linearly thru the pages – inefficient.
- What would be your strategy?

Binary search

```
binarySearch(dictionary, word){

   if (dictionary has one page) {// base case
       scan the page for word
   }

   else {// recursive case

       open the dictionary to a point near the middle
       determine which half of the dictionary contains word

       if (word is in first half of the dictionary) {
           binarySearch(first half of dictionary, word)
       }
       else {
           binarySearch(second half of dictionary, word)
       }
   }
```

Binary search

- Let's write a method called `binarySearch` that accepts a sorted array of integers and a target integer and returns the index of an occurrence of that value in the array.
  - If the target value is not found, return -1

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

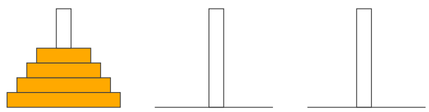
```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -1
```

Binary search

```
// Returns the index of an occurrence of the given
// value in the given array, or -1 if not found.
// Precondition: a is sorted
public int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}
// Recursive helper to implement search.
private int binarySearch(int[] a, int target,
                         int first, int last) {
    if (first > last) {
        return -1;   // not found
    } else {
        int mid = (first + last) / 2;
        if (a[mid] == target) {
            return mid;   // found it!
        } else if (a[mid] < target) {
            // middle element too small; search right half
            return binarySearch(a, target, mid+1, last);
        } else {  // a[mid] > target
            // middle element too large; search left half
            return binarySearch(a, target, first, mid-1);
        }
    }
}
```

Towers of Hanoi

Example: Towers of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.



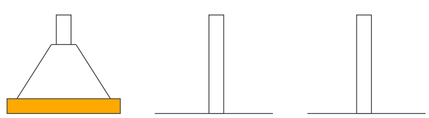
41

Try to find the pattern by cases

- One disk is easy
- Two disks...
- Three disks...
- Four disk...

Towers of Hanoi

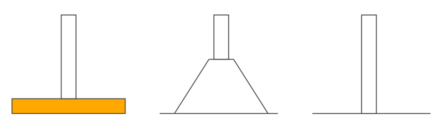
Example: Towers of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.



43

Towers of Hanoi

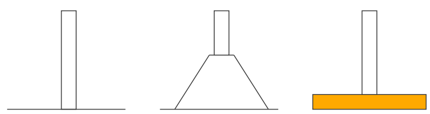
Example: Towers of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.



44

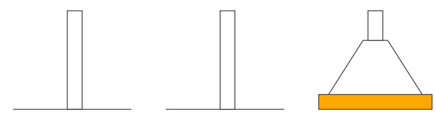
Towers of Hanoi

Example: Towers of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.



Towers of Hanoi

Example: Towers of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.



Let's go play with it at: http://www.mazeworks.com/hanoi/index.htm
Or http://www.mathsisfun.com/games/towerofhanoi.html

Fibonacci's Rabbits

- Suppose a newly-born pair of rabbits, one male, one female, are put on an island.
  - A pair of rabbits doesn't breed until 2 months old.
  - Thereafter each pair produces another pair each month
  - Rabbits never die.
- How many pairs will there be after n months?

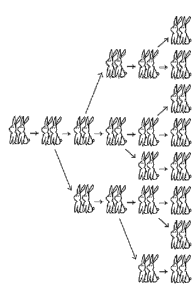


image from: http://www.jimloy.com/algebra/fibo.htm

Fibonacci numbers

- The *Fibonacci numbers* are a sequence of numbers $F_0, F_1, \ldots F_n$ defined by:

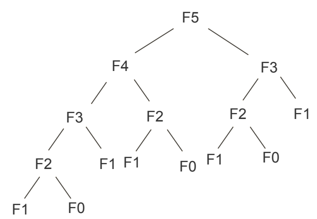
$$F_0 = F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \text{ for any } i > 1$$

- Write a method that, when given an integer *i*, computes the *nth* Fibonacci number.

Fibonacci numbers

- Let's run it for n = 1,2,3,... 10, ... , 20,...
- If n is large the computation takes a long time! Why?



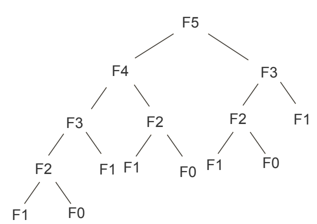
Fibonacci numbers

- recursive Fibonacci was expensive because it made many, recursive calls
  - fibonacci(n) recomputed fibonacci(n-1),…,fibonacci(1) many times in finding its answer!
  - this is a case, where the sub-tasks handled by the recursion are redundant with each other and get recomputed

50

Fibonacci numbers

- Every time n is incremented by 2, the call tree more than doubles.



Growth of rabbit population

1 1 2 3 5 8 13 21 34 ...

The fibonacci numbers themselves also grow rapidly: every 2 months the population at least **DOUBLES**