Recursion

Instructor: Nihshanka Debroy
(Several examples borrowed from Tammy Bailey)
A Russian nesting doll is a sequence of similar dolls inside each other that can be opened.

Each time you open a doll a smaller version of the doll will be inside, until you reach the innermost doll which cannot be opened.
Sierpinski Tetrahedron

Image taken from: http://www.palmyria.co.uk/illusions/geometry/sierpinskiitetrahedron.gif
Solving Problems Recursively

Recursion is an indispensable tool in a programmer’s toolkit.

- Allows many complex problems to be solved simply
- Elegance and understanding in code often leads to better programs: easier to modify, extend, verify
- Sometimes recursion is not appropriate, when it is bad it can be very bad --- every tool requires knowledge and experience in how to use it!
Recursive Definition

A statement in which something is defined in terms of smaller versions of itself. Consists of:

**Base case (also called exit case)**
- Simple case that can be solved with no further computation
- Does not make a recursive call
- Cannot be expressed in terms of smaller versions of itself

**Recursive case**
- Can be reduced to smaller versions of the same problem
- Makes a recursive call, with some parameter or other measure that decreases or moves towards the base case
- Ensure that sequence of calls eventually reaches the base case
Classic Examples of Recursion

- Factorial

- **Fibonacci numbers**: 1, 1, 2, 3, 5, 8, 13, 21, ...
  -- \( F(n) = F(n-1) + F(n-2) \)

- **Towers of Hanoi**
  -- \( N \) disks on one of three pegs, transfer all disks to another peg
  -- Never put a disk on a smaller one, only on larger
  -- Every solution takes "forever" when \( N \), number of disks, is large
Example: Factorial

- n! is the **factorial function**: for a positive integer n,
  
  \[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1 = n \times (n-1)! \]

- Know how to compute factorial using a for-loop
- We can also compute n! recursively:

  ```
  int factorial(int n)
  {
      if(n==1)
          return 1;
      else
          return n*factorial(n-1);
  }
  ```
Recursion trace

factorial(6) = 6*factorial(5)
  factorial(5) = 5*factorial(4)
    factorial(4) = 4*factorial(3)
      factorial(3) = 3*factorial(2)
        factorial(2) = 2*factorial(1)
          factorial(1) = 1

factorial(2) = 2*factorial(1) = 2*1 = 2
factorial(3) = 3*factorial(2) = 3*2 = 6
factorial(4) = 4*factorial(3) = 4*6 = 24
factorial(5) = 5*factorial(4) = 5*24 = 120
factorial(6) = 6*factorial(5) = 6*120 = 720
double Power(double x, int n)
// returns x^n
{
    if (n == 0)
    {
        return 1.0;
    }
    return x * Power(x, n-1);
}
Example: Fibonacci Numbers

- Fibonacci numbers: sequence of integers such that
  - the first and second terms are 1
  - each successive term is given by the sum of the two preceding terms
    
      1 1 2 3 5 8 13 21 34 55 ...

- Recursive definition:
  - let $F(n)$ be the $n^{\text{th}}$ Fibonacci number, then

$$F(n) = \begin{cases} 
1 & \text{if } n=1 \text{ or } n=2 \\
F(n-1)+F(n-2) & \text{if } n>2
\end{cases}$$
Recursive subroutine for Fibonacci

```c
int fibonacci(int n)
{
    if( n==1 || n==2 )
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2);
}
```
Recursion vs. Iteration

- Every recursive algorithm can be written non-recursively.
- If recursion does not reduce the amount of computation, use a loop instead:
  - why? … recursion can be inefficient at runtime
- Just as efficient to compute n! by iteration (loop)

```c
int factorial(int n)
{
    int fact = 1;
    for(int k=2; k<=n; k++)
        fact = fact*k;
    return fact;
}
```
Number Sequences

- The ancient Greeks were very interested in sequences resulting from geometric shapes such as the **square numbers** and the **triangular numbers**
- Each set of shapes represents a number sequence
- The first three terms in each sequence are given
- What are the next three terms in each sequence?
- How can you determine in general the successive terms in each sequence?
Square Numbers

The 4th square number is $9 + 2 \times 4 - 1 = 16$

The 5th square number is $16 + 2 \times 5 - 1 = 25$

The 6th square number is $25 + 2 \times 6 - 1 = 36$

1, 4, 9, 16, 25, 36, ...

• Write a recursive Java subroutine to compute the $n^{th}$ square number
Square Numbers

- Base case
  - the first square number is 1
- Recursive case
  - the n\textsuperscript{th} square number is equal to $2n-1 + \text{the (n-1)}^{st}$ square number
Recursive Solution

```c
int SquareNumber( int n )
{
    if( n==1 )
        return 1;
    else
        return 2*n-1 + SquareNumber( n-1 );
}
```
Triangular Numbers

1, 3, 6, 10, 15, 21, ...

- Write a recursive Java subroutine to compute the $n^{th}$ triangular number

The 4th triangular number is $6 + 4 = 10$

The 5th triangular number is $10 + 5 = 15$

The 6th triangular number is $15 + 6 = 21$
Triangular Numbers

- Base case
  - the first triangular number is 1
- Recursive case
  - the $n^{th}$ triangular number is equal to $n + \text{the } (n-1)^{st} \text{ triangular number}$
Recursive Solution

```c
int TriNumber( int n )
{
    if( n==1 )
        return 1;
    else
        return n + TriNumber( n-1 );
}
```
```c
int Something(int a, int b)
{
    if (a < b)
        return a + b;
    else
        return 1 + Something(a - b, b);
}
```

- What is the value of `x` after executing the following statement?
  
  ```
  x = Something(2, 7);
  ```

- What is the value of `y` after executing the following statement?

  ```
  y = Something(8, 2);
  ```
What is the value of the expression `Strange(5)`?

- a. 5
- b. 9
- c. 11
- d. 15
- e. 20

What is the value of the expression `Strange(6)`?

- a. 6
- b. 10
- c. 12
- d. 15
- e. 21

```c
int Strange( int x )
{
    if( x <= 0 )
        return 0;
    else if( x%2 == 0 )
        return x + Strange(x-1);
    else
        return x + Strange(x-2);
}
```
int Weirdo(int n)
{
    if(n == 1)
        return 1;
    else
        return 2 * Weirdo(n-1) * Weirdo(n-1);
}

What is the value of the expression `Weirdo(4)`?

a. 16  
b. 32  
c. 64  
d. 128 
e. 256

If n is a positive integer, how many times will `Weirdo` be called to evaluate `Weirdo(n)` (including the initial call)?

a. 2n  
b. 2n-1  
c. 2^n  
d. 2^(n-1)  
e. 2^(n-1)