Recursion

- Russian nesting dolls
  - 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

Recursive definition

- A statement in which something is defined in terms of smaller versions of itself
- A recursive definition consists of:
  - a base case: the part of the definition that cannot be expressed in terms of smaller versions of itself
  - a recursive case: the part of the definition that can be expressed in terms of smaller versions of itself

Binary trees

- Tree where all nodes have at most two children
- Recursive definition: A binary tree is a structure defined on a finite set of nodes that either
  - contains no nodes, or
  - is composed of a root node, a right subtree which is a binary tree, and a left subtree which is a binary tree

<table>
<thead>
<tr>
<th>Base case: zero nodes</th>
<th>Recursive case:</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
</tr>
<tr>
<td>binary tree as right subtree</td>
<td>binary tree as right subtree</td>
</tr>
<tr>
<td>binary tree as left subtree</td>
<td>binary tree as left subtree</td>
</tr>
</tbody>
</table>

Mergesort

- Divide and conquer sorting algorithm
- Defined recursively

- Divide: Divide n element array to be sorted into two subarrays with n/2 items each
- Conquer: Sort the two subarrays recursively using mergesort
- Combine: Merge the two sorted subarrays to produce the sorted solution
Fundamentals of recursion

• **Base case**
  - at least one simple case that can be solved without recursion

• **Reduction step**
  - reduce problem to a *smaller* one of same structure
  - problem must have some measure that moves towards the base case in each recursive call
  - ensures that sequence of calls eventually reaches the base case

• **The Leap of Faith!**
  - always assume the recursive case works!

---

Classic example - factorial

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

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

```c
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*2 = 6
factorial(4) = 4*6 = 24
factorial(5) = 5*24 = 120
factorial(6) = 6*120 = 720
```

---

Recursion ⇔ 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)
{
    if( n==1 || n==2 )
        return 1;
    else
        return fact*factorial(n-1);
    fact = fact*k;
    return fact;
}
```

---

Classic bad 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

  \[ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ 34 \ 55 \ldots \]

• Recursive definition:
  - let \( F(n) \) be the n\textsuperscript{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);
}
```
Fibonacci recursion

```
fib(7)  
  |    
  v    v
fib(5)  fib(4)
  |    
  v    v
fib(3)  fib(2)
  |    
  v    v
fib(1)  fib(2)
```

Redundant computation

```
int fibonacci(int n)
{
    if ( n==1 || n==2 )
       return 1;
    else
       return fibonacci(n-1)+fibonacci(n-2);
}
```

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

- Write a recursive Java subroutine to compute the n
th square number.

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

- Write a recursive Java subroutine to compute the n-th triangular number

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

- Base case
  - the first triangular number is 1
- Recursive case
  - the n-th triangular number is equal to n + the (n-1)-th triangular number

Recursive solution

```java
int TriNumber( int n )
{
    if( n==1 )
        return 1;
    else
        return n + TriNumber( n-1 );
}
```

Something

```java
int Something( int a, int b )
{
    if( a < b )
        return a+b;
    else
        return 1 + Something( a-b, b );
}
```

Strange

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

```java
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);
}
```

Weirdo

```java
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