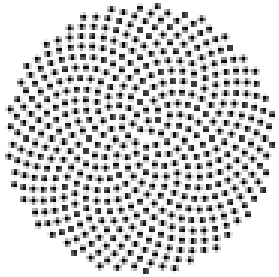
# RECURSION

# Today

- This lecture looks at
  - The basics of recursion.
  - Some examples of recursive functions.
- The textbook doesn't cover recursion in any detail (the only material is on pages 96 and 97 in my copy)..

# Recursion

- Recursion is defining something in terms of itself
- There are many examples in nature:
  - Seeds in a sunflower

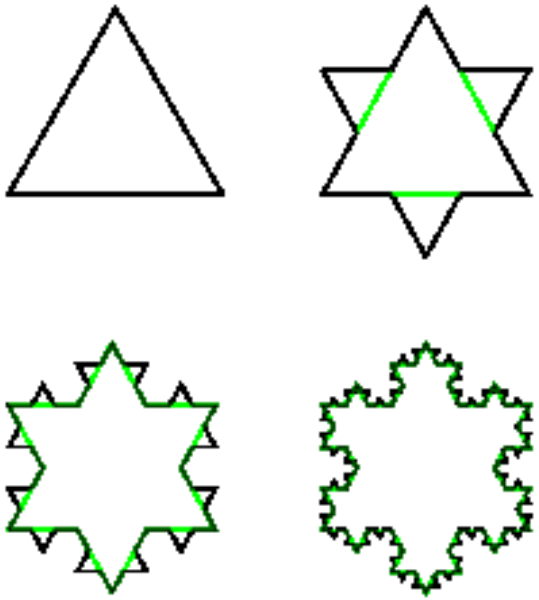


- . . . in mathematics:
  - Factorial
  - Induction
- . . . and in computer graphics:
  - Koch snowflake

# Koch snowflake

- Starting with a line, then:
  1. Divide each line into three segments of equal length.
  2. Draw an equilateral triangle that has the middle segment from step 1 as its base and points outward.
  3. Remove the line segment that is the base of the triangle from step 2.
- Repeat as often as you like.

- Here are the first four iterations of the Koch snowflake.



- The more iterations, the more snowflaky it looks.

# Power function

- *Power* is defined recursively:

$$x^y = \begin{cases} \text{if } y == 0, & x^y = 1 \\ \text{otherwise}, & x^y = x * x^{y-1} \end{cases}$$

- There are two parts to the definition:
  - The *base case*, what we do when $y$ is zero.
  - The *recursive case*, what we do when $y$ is not zero.
- This is the common pattern for all recursive definitions.

## Here it is in C++

```
// r1.cpp
#include <iostream>
using namespace std;

int power( int x, int y ) {
  if ( y == 0 )
    return( 1 );
  else
    return( x * power( x, y-1 ));
} // end of power()

int main() {
  cout << "2^3 = " << power( 2,3 ) << endl;
}
```

- Notice that `power()` calls itself!
- This seems to be magic, but we'll see how it is done in a moment.
- You can make recursive calls with any method *except main()*
- BUT beware of infinite loops!!!
- You have to know when and how to stop the recursion — what is the *stopping* condition.

# Walking through `power( 2,4 )`

- Initial call is `power(2, 4)`

| | call | x | y | return value |
|---|---|---|---|---|
| 1 | power( 2,4 ) | 2 | 4 | 2 * power( 2,3 ) |
| 2 | power( 2,3 ) | 2 | 3 | 2 * power( 2,2 ) |
| 3 | power( 2,2 ) | 2 | 2 | 2 * power( 2,1 ) |
| 4 | power( 2,1 ) | 2 | 1 | 2 * power(2, 0 ) |
| 4 | power( 2,0 ) | 2 | 0 | 1 |

- The first is the *original call*
- Followed by four *recursive calls*

# Stacks

- The computer uses a data structure called a *stack* to keep track of what is going on
- Think of a *stack* like a stack of plates
- You can only take off the top one
- You can only add more plates to the top
- This corresponds to the two basic *stack operations*:
  - *push* — putting something onto the stack
  - *pop* — taking something off of the stack
- When each recursive call is made, `power()` is pushed onto the stack
- When each return is made, the corresponding `power()` is popped off of the stack

## Another example: factorial

- *factorial* is defined recursively:

$$N! = \begin{cases} \text{if } N == 1, & N! = 1 \\ \text{otherwise}, & N! = N * (N-1)! \end{cases}$$

(for $N > 0$)

## Here it is in C++

```
// r2.cpp
#include <iostream>
using namespace std;

int factorial ( int N ) {
  if ( N == 1 )
    return( 1 );
  else
    return( N * factorial( N-1 ));
} // end of factorial()


int main() {
  cout << "5! = " << factorial( 5 ) << endl;
}
```

- Walk through `factorial(4)`

# Another example

```
//r3.cpp
#include <iostream>
using namespace std;

void countDown (int n) {
  if ( n <= 0 )
    cout << "Blastoff!" << endl;
  else {
    cout << "Time to launch is " << n << " seconds" << endl;
    countDown(n - 1);
  }
} // end of factorial()


int main() {
  countDown(5);
}
```

- What does this do?

- Again `countDown` has the general structure:

```
// base case part

if (<base-case condition>)
   return <base-case-value>

// general case

else
   return <recursively computed expression>
```

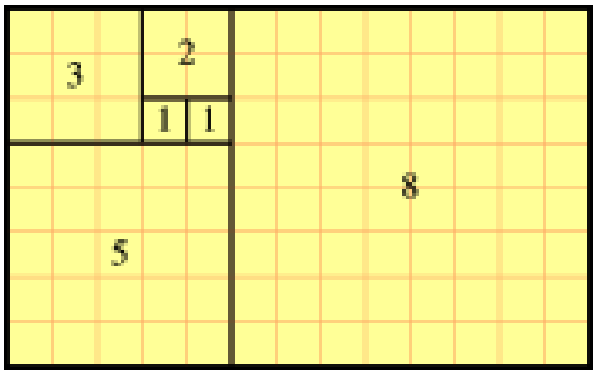
- This is common to all recursive functions — the only difference you'll see is that some functions have two base cases.

# Fibonacci

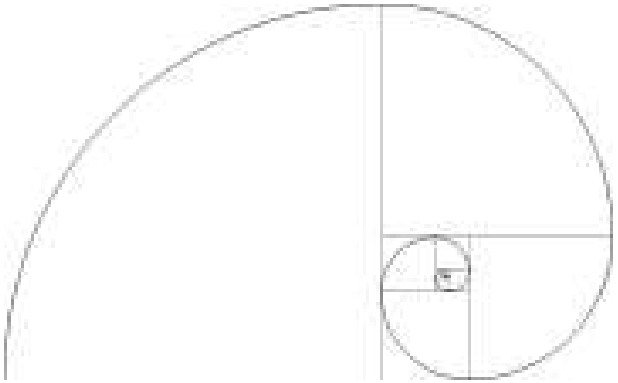
```
// in r4.cpp

int fibonacci (int n) {
  if (n == 0){
    return 0;
  }
  else
    if (n == 1){
      return 1;
    }
    else {
      return(fibonacci(n - 1) + fibonacci(n -2));
    }
} // end of fibonacci()
```

- – A tiling where tile sides are successive members of the Fibonacci sequence.




– A spiral constructed from the above tiling.



# Summary

- This lecture has looked at
  - The basic idea of recursion
  - A bunch of different examples of recursion
- We will look at recursion more in the next lecture.