Recursion Overview

• In order to solve a problem, solve a smaller version of the same problem
  • In order to solve that problem, solve a smaller version of the same problem
    • In order to solve that problem, solve a smaller version of the same problem
      • In order to solve that problem, solve a smaller version of the same problem
        • In order to solve that problem, solve a smaller version of the same problem
          • In order to solve that problem, solve a smaller version of the same problem
            • …..

• “A function calling itself”
Recursion Overview

• Solving smaller versions of the same problem (recursive case) until we reach a version that is so simple, you can just do it (base case).

• **Factorials:**

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \]

- \( n! \) is just \( n \times (n-1)! \)
- \( (n-1)! \) is just \( (n-1) \times (n-2)! \)
- \( (n-2)! \) is just \( (n-2) \times (n-3)! \)

\[ \ldots \]

- \( 1! \) is just \( 1 \)
Recursion Practice

This week’s section handout, Recursion #2:

Write a recursive function named `sumOfSquares` that takes in an integer \( n \) and returns the sum of squares from 1 to \( n \), inclusive.

For example, \( \text{sumOfSquares}(3) \) should return 14 (\( 1^2 + 2^2 + 3^2 = 14 \)). You can assume \( n \geq 1 \).
Recursion Practice

Base case?
What is the simplest $n$ for which we can find the $\text{sumOfSquares}$?

ANS: $n = 1$
(Remember, we were allowed to assume that $n \geq 1$)

$$\text{sumOfSquares}(1) = 1^2 = 1$$
Recursion Practice

Recursive case?
Given integer \( k \), what’s the input that’s just one step smaller?

ANS: \( n = k - 1 \)

If we have \( \text{sumOfSquares}(k-1) \), how do we get \( \text{sumOfSquares}(k) \)?

ANS: \[
\text{sumOfSquares}(k) = k^2 + \text{sumOfSquares}(k-1)
\]
Recursion Practice

Solution:

```c
int sumOfSquares(int n) {
    if(n == 1) {
        return 1;
    } else {
        return n*n + sumOfSquares(n-1);
    }
}
```
Part A. Fractals
What is a Fractal?

• A figure that displays **self-similarity** on all scales
What is a Fractal?

• Fractals are naturally **recursive** objects

1 big one = 3 smaller ones
Sierpinski Triangle
Sierpinski Triangle

```c
void drawSierpinskiTriangle(Gwindow &gw, double x, double y, double size, int order)
```

Order 1

Order 2

Order 3
Drawing equilateral triangles

```c
void drawLine(double x1, double y1,
              double x2, double y2)
Usage → gw.drawLine(20, 20, 40, 40);
```

Use trig to find h!
Sierpinski Triangle

Approach

• Must write **recursively**
  • No loops, no data structures allowed
  • What’s a good base case? What’s the recursive case?

• Hint: to draw the Order $n$ triangle, you need to draw three smaller Order $n - 1$ triangles.
Sierpinski Triangle

To draw an Order $n$ Triangle here…

…you should draw Order $n-1$ Triangles in these places!
Sierpinski Triangle – tips

• All triangles you draw should **point downwards**
  • If you’re drawing any upward-pointing triangles, double check your approach!
  • Each line in the final drawing should only be traced **once** – don’t redraw any lines!

• Don’t forget edge cases and exceptions!
  • Order 0 triangle?
  • Negative values for x, y, size, or order?
Mandelbrot Set
Complex Numbers

Complex numbers are of the form $a + bi$
where $i$ is the imaginary number $\sqrt{-1}$

Complex numbers can be graphed in the complex plane

$(3, 2) = 3 + 2i$
Complex Numbers

Real part \( a + bi \) Imaginary part

- **Addition** \((a_1 + b_1 i) + (a_2 + b_2 i)\)
  \[= (a_1 + a_2) + (b_1 + b_2)i\]
  Add real and imaginary parts separately

- **Multiplication** \((a_1 + b_1 i) (a_2 + b_2 i)\)
  \[= a_1a_2 + (a_1b_2 + a_2b_1)i - b_1b_2\]
  FOIL

- **Absolute value** \(|(a + bi)|\)
  \[= \sqrt{a^2 + b^2}\]
  Distance from origin of complex plane
**Complex Numbers**

- We provide a **Complex** class to help you work with complex numbers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex(double a, double b)</td>
<td>Constructor to create a complex number a + bi</td>
</tr>
<tr>
<td>c.abs()</td>
<td>Returns absolute value of c</td>
</tr>
<tr>
<td>c.real()</td>
<td>Returns real part of c</td>
</tr>
<tr>
<td>c.imag()</td>
<td>Returns coefficient of imaginary part of c</td>
</tr>
<tr>
<td>c1 + c2</td>
<td>Returns sum of c1 and c2</td>
</tr>
<tr>
<td>c1 * c2</td>
<td>Returns product of c1 and c2</td>
</tr>
</tbody>
</table>
Mandelbrot Set – what is it?

The set of all complex numbers $c$ that satisfy the following property:

• The function $f(z) = z^2 + c$ does not diverge when iterated from $z = 0$.

• i.e. the sequence $f(0), f(f(0), f(f(f(0))))$, ... does not diverge to infinity

When you plot all the values in the Mandelbrot set, it looks like this →

(black = in the set)
Mandelbrot Set – Recursion!

We can use the following recursive definition for the Mandelbrot Set to figure out what numbers are in it:

\[ z_{n+1} = z_n^2 + c \]
\[ z_0 = 0, n \to \infty \]

- \( z_0 = 0 \)
- \( z_1 = z_0^2 + c = 0^2 + c = c \)
- \( z_2 = z_1^2 + c = c^2 + c \)
- \( z_3 = z_2^2 + c = (c^2 + c)^2 + c \)
- Etc.
We can use the following recursive definition for the Mandelbrot Set to figure out what numbers are in it:

\[ z_{n+1} = z_n^2 + c \]
\[ z_0 = 0, n \to \infty \]

- If \( |z_n| \) does not diverge after infinitely many iterations (or for our purposes, some large number like 200), then \( c \) is in the set.
- If \( |z_n| \) does diverge at some point (for our purposes, if it exceeds 4), then \( c \) is not in the set.
void mandelbrotSet(Gwindow &gw, double minX, double incX, double minY, double incY, int maxIterations, int color)

int mandelbrotSetIterations(Complex c, int maxIterations)

int mandelbrotSetIterations(Complex z, Complex c, int remainingIterations)
Mandelbrot Set – overall function

```c
void mandelbrotSet(Gwindow &gw, double minX, double incX, double minY, double incY, int maxIterations, int color)
```

- **minX** and **minY** – the complex number at the upper left of your grid (dictates your “window”)
  - Complex startingCoord = Complex(minX, minY);

- **incX** and **incY** – the increment you should move as you go from square to square in your grid (“resolution”)
  - \((\text{row} = 3, \text{col} = 5) = (\text{minX} + 3 \times \text{incX}, \text{minY} + 5 \times \text{incY})\)

- **maxIterations** – the number of iterations you should try before determining that a number does not diverge
Mandelbrot Set – helpers

int mandelbrotSetIterations(Complex c, int maxIterations)
int mandelbrotSetIterations(Complex z, Complex c, int remainingIterations)

• Compute the number of iterations needed to determine if a particular number \( c \) diverges
• Same name, different parameters (“overloaded”)
• First = wrapper function
  • Returns how many iterations were needed for number \( c \)
• Second = recursive helper function
  • Implements the recursive definition for the Mandelbrot Set
  • Remember: \( z \) starts at 0!
void mandelbrotSet(Gwindow &gw, double minX, double incX, double minY, double incY, int maxIters, int color) {
    //for each pixel
    Complex c = Complex(pixelX, pixelY);
    numIters = mandelbrotSetIterations(c, maxIters);
    //color pixel
}

int mandelbrotSetIterations(Complex c, int maxIterations) {
    //call mandelbrotSetIterations
}
Mandelbrot Set – coloring

```c
void mandelbrotSet(Gwindow &gw, double minX, double incX,
                    double minY, double incY, int maxIterations, int color)
```

- **color** – determines the color of your graph
  - **color != 0** – set the pixel’s color to color if that pixel represents a number in the set

```
pixels[r][c] = color;
```
Mandelbrot Set – coloring

```java
void mandelbrotSet(Gwindow &gw, double minX, double incX, 
                   double minY, double incY, int maxIterations, int color)
```

- **color** – determines the color of your graph
  - `color == 0` – set the pixel’s color based on what `mandelbrotSetIterations` returned

```java
pixels[r][c] = palette[numIterations % palette.size()];
```
Part B. Grammar Solver
What is a Grammar?

• A *formal language* is a set of words/symbols plus a set of rules that dictate how those symbols can be put together
  
  • A *grammar* describes the rules for a particular formal language

**Symbols:**
S, NP, AdjP, V’, green, etc.

**Rules:**
S → NP + VP
NP → N’
N’ → AdjP + N’
What is a Grammar?

- A grammar could reflect how we understand grammar in the English language (or any other spoken language), but it doesn’t have to.
  - You can make a language out of arbitrary symbols and a grammar out of arbitrary relationships between those symbols!

You could have \( S \rightarrow NP + VP \)

(Sentence \( \rightarrow \) Noun Phrase + Verb Phrase)

You could also have \( \text{diamond} \rightarrow \text{square} + \text{ampersand} \)
Backus-Naur Form (BNF)

- A way of formatting the rules of a grammar
  
  non-terminal1::=rule|rule|rule|...
  non-terminal2::=rule|rule|rule|...

- **non-terminal**: a symbol that gets expanded into other symbols (think of this as a part of speech)
  - **terminal**: a symbol that does not expand (i.e. it terminates) (think of this as a word)

- **rule**: a sequence of symbols that a non-terminal can expand to. Different possible rules are separated by “|”
BNF – an example

- Start with <$S>$
- Follow its rule: <$S>$ → <$NP>$<$VP>$
  - Take <$NP>$, choose a rule: <$NP>$ → <$AdjP>$<$NP>$
    - Take <$AdjP>$, choose a rule: <$AdjP>$ → sleepy
    - Take <$NP>$, choose a rule: <$NP>$ → cat
  - Take <$VP>$, choose a rule: <$VP>$ → barks
- Final sentence: sleepy cat barks
Grammar Solver

Vector<string> grammarGenerate(istream &input,
                               string symbol, int times)

- **input** – an input stream containing a grammar in Backus-Naur Form
- **symbol** – a starting symbol for each sentence to be generated
- **times** – the number of sentences to generate
Grammar Solver

Vector<string> grammarGenerate(istream &input, string symbol, int times)

1. Read the input file and store the grammar into some data structure
2. Randomly generate sentences (starting with the given symbol) from the grammar
   • Must be done recursively!
3. Return a vector of the sentences generated
1. Reading the Input File

- Read each line of the file and store grammar in a Map (no recursion needed):

\[
\text{non-terminal1 ::= rule|rule|rule|...}
\]

**Helpful functions from `strlib.h`:**
- `Vector<string> stringSplit(string s, string delimiter)`
- `void trim(String s)`
2. Generating sentences

- **Recursively** generate random expressions given a starting symbol $S$.

1. If $S$ is a terminal, then that’s it! The resulting expression is just $S$ itself.
2. If $S$ is a non-terminal, randomly select one of its rules $R$.
3. For each symbol in $R$, generate a random expression.
2. Generating sentences

<s> ::= <np> <vp>
<np> ::= <dp> <adjp> <n> | <pn>
<dp> ::= the | a
<adjp> ::= <adj> | <adj> <adjp>
<adj> ::= big | fat | green | wonderful | fat
<n> ::= dog | cat | man | university | father
<pn> ::= John | Jane | Sally | Spot | Fred | El
<vp> ::= <tv> <np> | <iv>
<tv> ::= hit | honored | kissed | helped
<iv> ::= died | collapsed | laughed | wept
Questions?
Extension: Fractal Tree
Fractal Tree

• Draw a tree as shown below with (you guessed it) recursion
Fractal Tree

```c
void drawTree(Gwindow &gw, double x, double y, double size, int order)
```
Fractal Tree