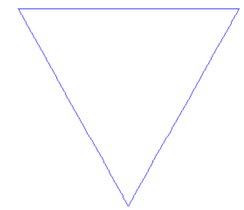
CS 106B, Lecture 10
Recursion and Fractals

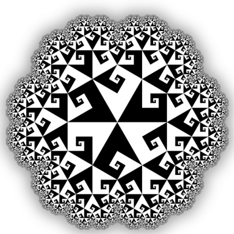
Plan for Today

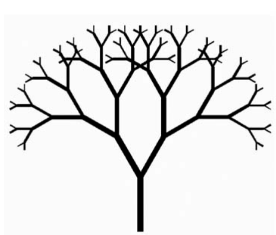
- One more recursive data example
- Introduction to **fractals**, a powerful tool used in graphics

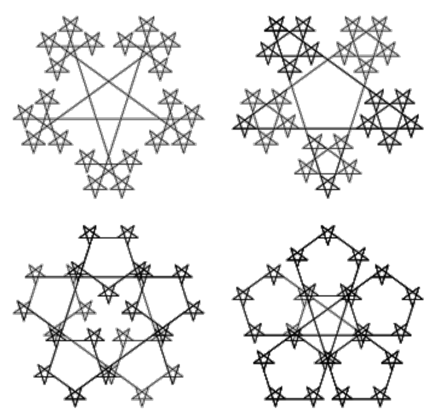
Fractals

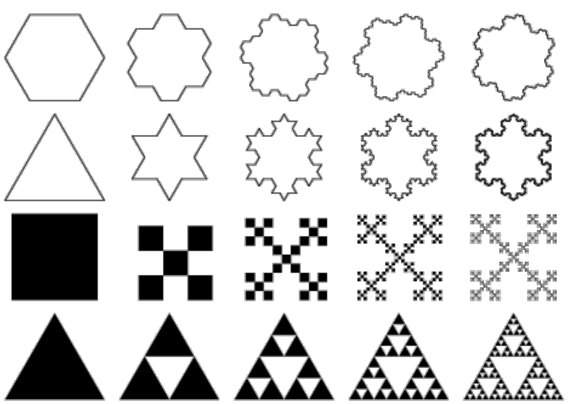
- **fractal**: A self-similar mathematical set that can often be drawn as a recurring graphical pattern.
  - Smaller instances of the same shape or pattern occur within the pattern itself.
  - When displayed on a computer screen, it can be possible to infinitely zoom in/out of a fractal.









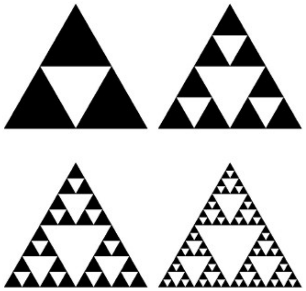


Fractals in nature

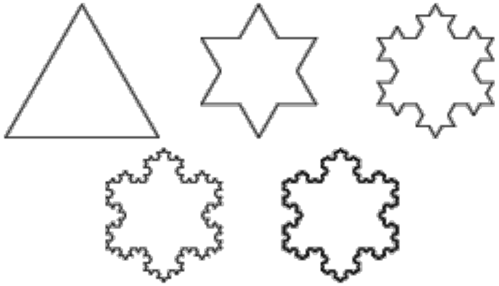
- Many natural phenomena generate fractal patterns:
  - earthquake fault lines
  - animal color patterns
  - clouds
  - mountain ranges
  - snowflakes
  - crystals
  - DNA
  - shells
  - ...

Example fractals

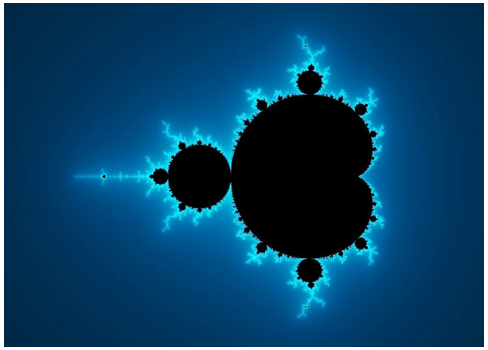
- **Sierpinski triangle**: equilateral triangle contains smaller triangles inside it (your next homework)



- **Koch snowflake**: a triangle with smaller triangles poking out of its sides



- **Mandelbrot set**: circle with smaller circles on its edge

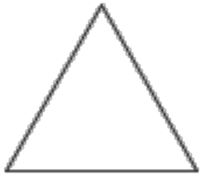


Coding a fractal

- Many fractals are implemented as a function that accepts x/y coordinates, size, and a *level* parameter.
  - The *level* is the number of recurrences of the pattern to draw.
  - The *position* and *size* change in the recursive call; *level* decreases by 1

- Example, Koch snowflake:

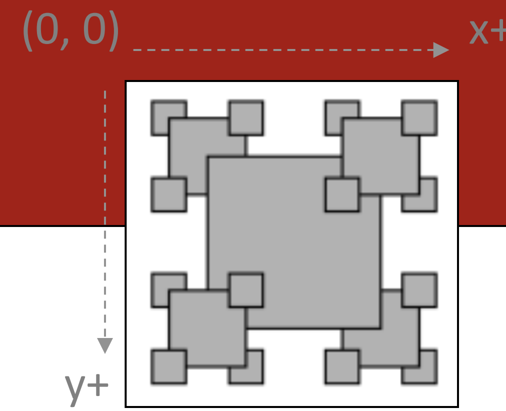
```
snowflake(window, x, y, size, 1);
```



```
snowflake(window, x, y, size, 2);
```

```
snowflake(window, x, y, size, 3);
```

Boxy fractal



- Where should the following line be inserted in order to get the figure at right?

```
gw.fillRect(x - size / 2, y - size / 2, size, size);
```

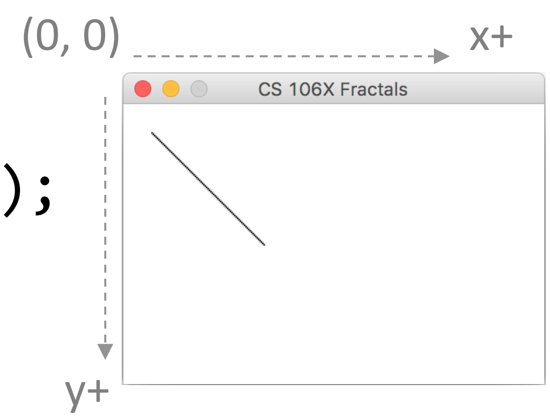
```
void boxyFractal(GWindow& gw, int x, int y, int size, int order) {
  if (order >= 1) {
    // A) here
    boxyFractal(gw, x - size / 2, y - size / 2, size / 2, order - 1);
    // B) here
    boxyFractal(gw, x + size / 2, y + size / 2, size / 2, order - 1);
    // C) here
    boxyFractal(gw, x + size / 2, y - size / 2, size / 2, order - 1);
    // D) here
    boxyFractal(gw, x - size / 2, y + size / 2, size / 2, order - 1);
    // E) here
    }
}
```

Stanford graphics lib

```
#include "gwindow.h"
```

`gw.drawLine(x1, y1, x2, y2);`	draws a line between the given two points
`gw.drawPolarLine(x, y, r, t);`	draws line from (*x*,*y*) at angle *t* of length *r*; **returns the line's end point as a `GPoint`**
`gw.getPixel(x, y)`	returns an RGB int for a single pixel
`gw.setColor("color");`	sets color with a color name string like "red", or #RRGGBB string like `"#ff00cc"`, or RGB int
`gw.setPixel(x, y, rgb);`	sets a single RGB pixel on the window
`gw.drawOval(x, y, w, h);` `gw.fillRect(x, y, w, h); ...`	other shape and line drawing functions (see online docs for complete member list)

```
GWindow gw(300, 200);
gw.setTitle("CS 106B Fractals");
gw.drawLine(20, 20, 100, 100);
```



Cantor Set

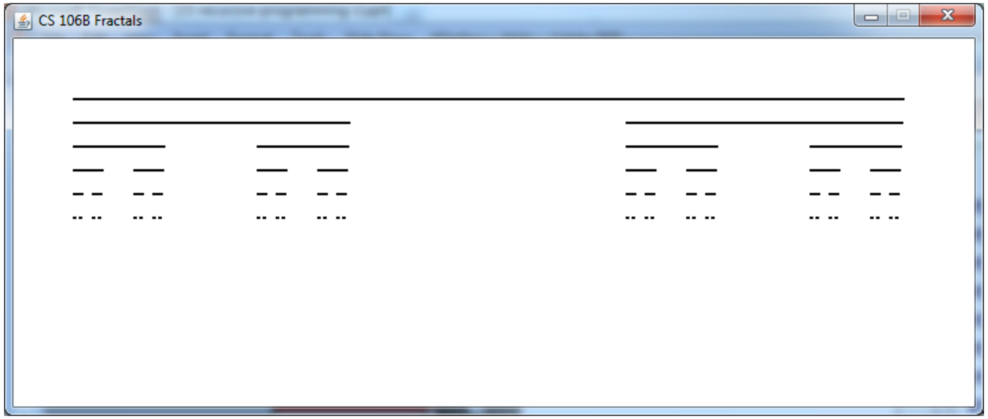
- The **Cantor Set** is a simple fractal that begins with a line segment.
  - At each *level*, the middle third of the segment is removed.
  - In the next *level*, the middle third of each third is removed.

- Write a function **`cantorSet`** that draws a Cantor Set with a given number of levels (lines) at a given position/size.
  - Place CANTOR_SPACING of vertical space between levels.

- How is this fractal *self-similar*?
- What is the *minimum amount of work* to do at each level?
- What's a good stopping point (base case)?

Cantor Set solution

```
void cantorSet(GWindow& window, int x, int y,
               int width, int levels) {
    if (levels > 0) {
        // recursive case: draw line, then repeat by thirds
        window.drawLine(x, y, x + width, y);
        cantorSet(window, x, y + 20, width/3, levels-1);
        cantorSet(window, x + 2*width/3, y + 20, width/3, levels-1);
    }
    // else, base case: 0 levels, do nothing
}
```

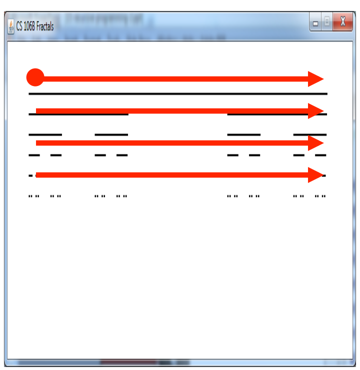


Cantor Set animated

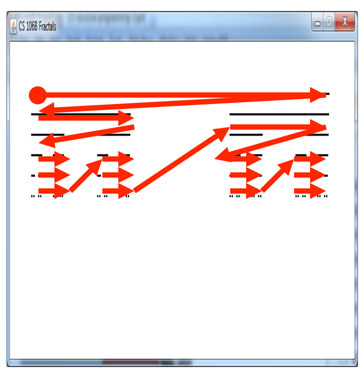
Q: Which way does the drawing animate? *(How could we change it?)*

```
void cantorSet(GWindow& window, int x, int y,
               int width, int levels) {
    if (levels > 0) {
        // recursive case: draw line, then repeat by thirds
        pause(250);
        window.drawLine(x, y, x + width, y);
        cantorSet(window, x, y + 20, width/3, levels-1);
        cantorSet(window, x + 2*width/3, y + 20, width/3, levels-1);
    }
}  //
```

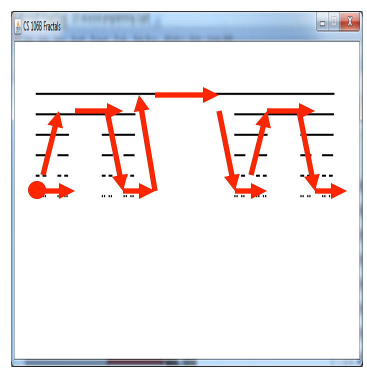
A.



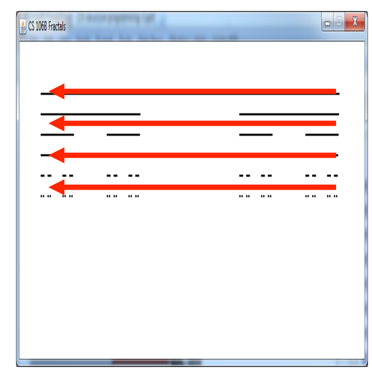
B.



C.



D.

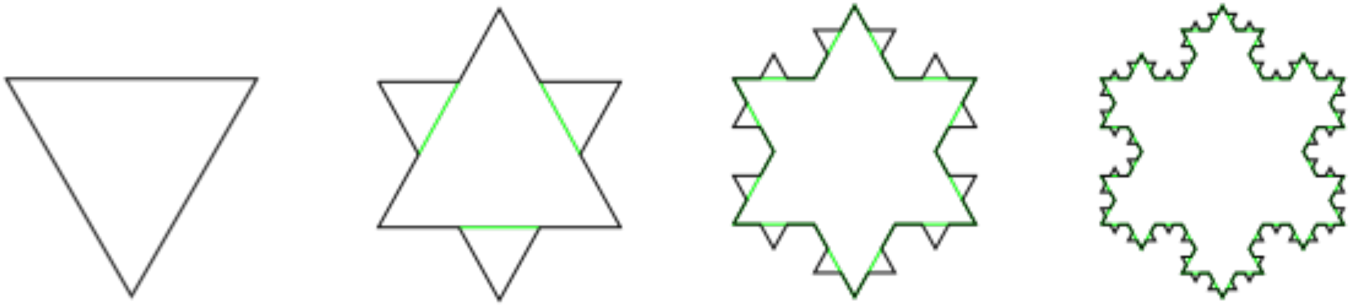


Announcements

- Homework 2 due on today at **5PM**
- Homework 1 grades will be released by your section leader soon!
- Shreya will be guest-lecturing on Monday
  - My office hours will be cancelled that day (still available via email)
- **Midterm Review Session** on Tuesday, July 24, from 7-9PM in Gates B01

Koch snowflake

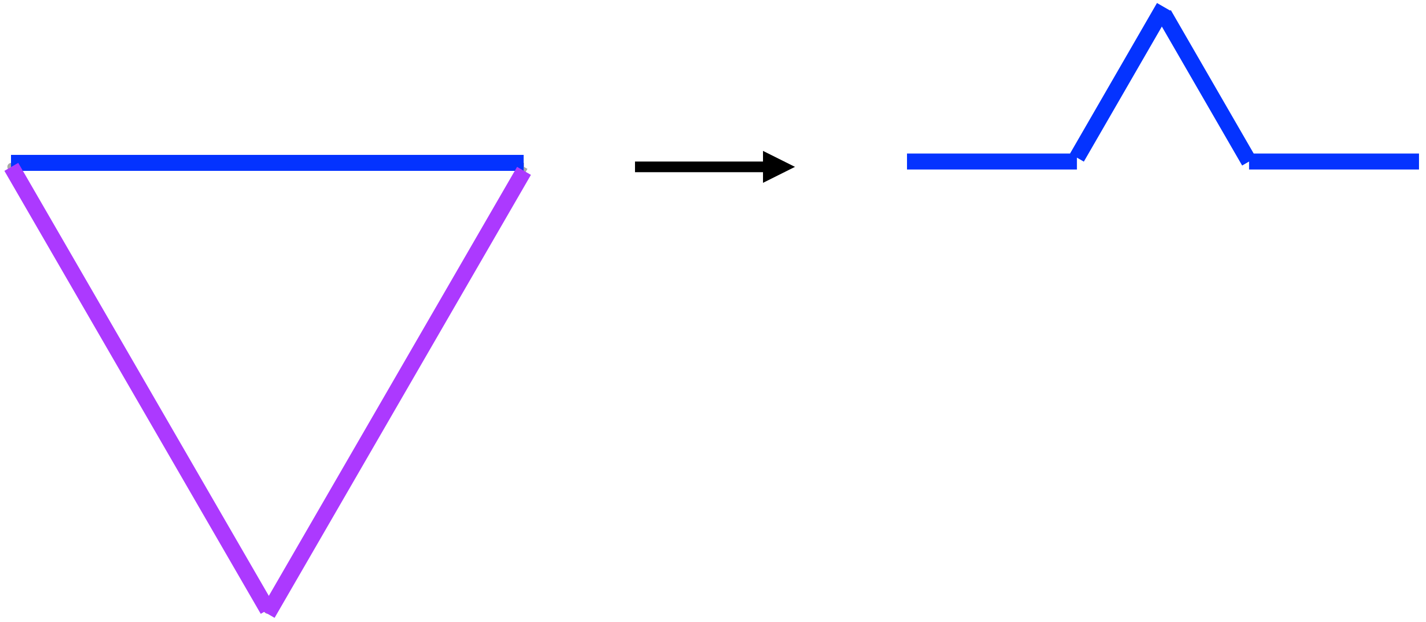
- **Koch snowflake**: A fractal formed by pulling a triangular "bend" out of each side of an existing triangle at each level.



- Start with an equilateral triangle, then:
  - Divide each of its 3 line segments into 3 parts of equal length.
  - Draw an eq.triangle with middle segment as base, pointing outward.
  - Remove the middle line segment.

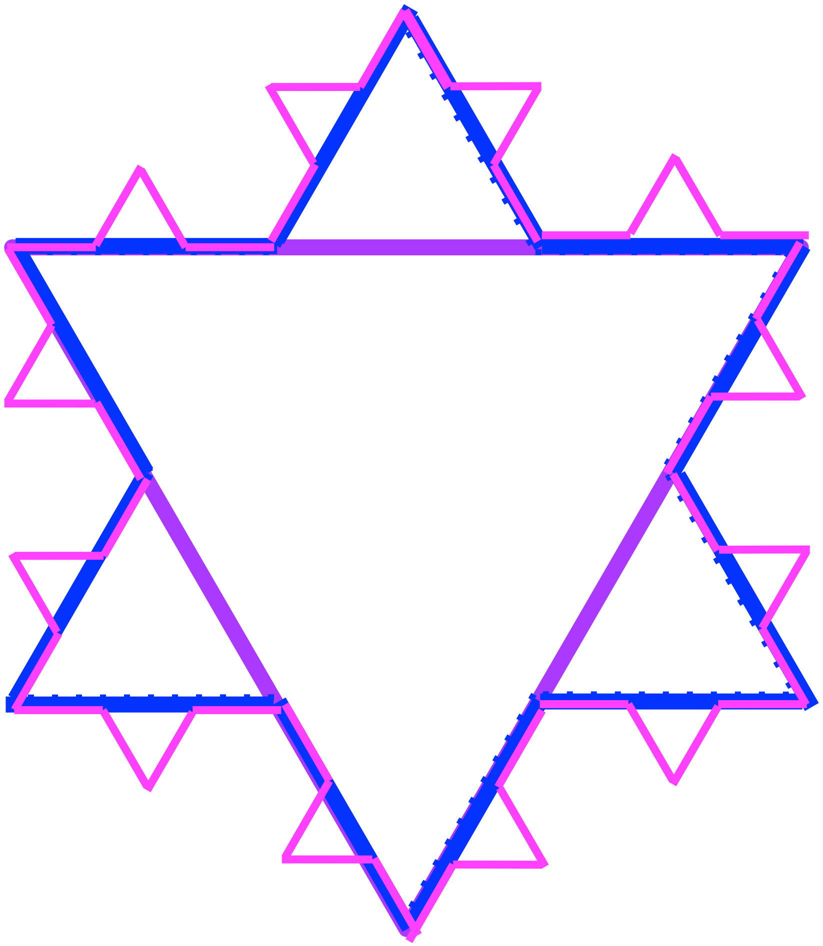
Line segment replace

- Replace each line segment as follows:



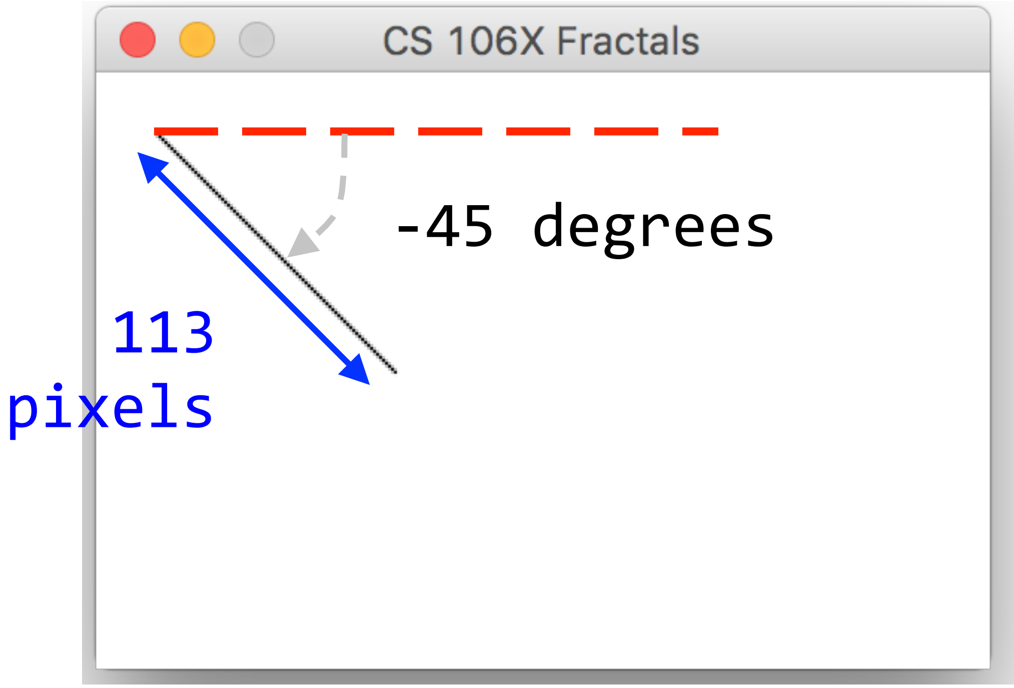
Multiple levels

- How is this fractal self-similar?



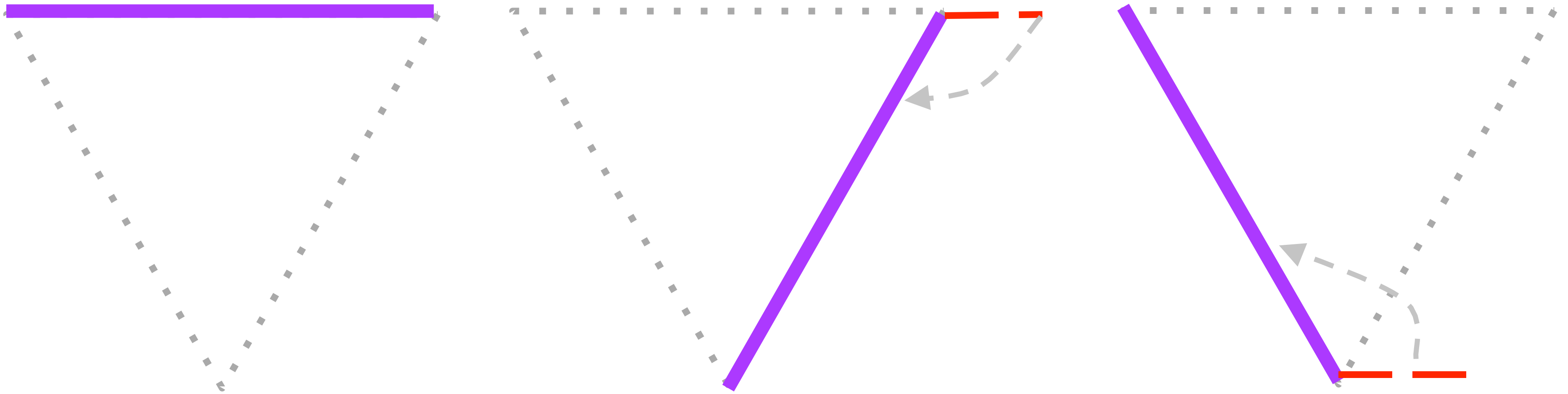
Polar lines

```
//                    x   y    r   theta
window.drawPolarLine(20, 20, 113, -45);
```



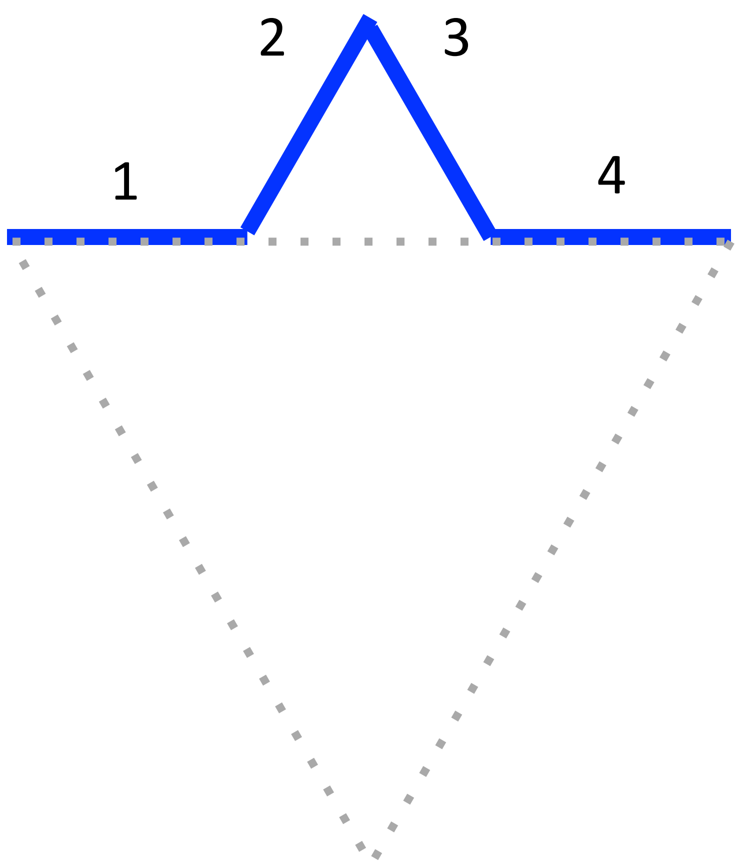
Triangle in polar

- Segment 1: Segment 2: Segment 3:



Segment in polar

- Think of a triangle side as 4 polar line segments, as below.
  - What are their angles, relative to the angle of this triangle side?



Snowflake solution

```
GPoint ksLine(GWindow& gw, GPoint pt, int size, int t, int levels) {
    if (levels == 1) {
        return gw.drawPolarLine(pt, size, t);
    } else {
        pt = ksLine(gw, pt, size/3, t, levels - 1);
        pt = ksLine(gw, pt, size/3, t + 60, levels - 1);
        pt = ksLine(gw, pt, size/3, t - 60, levels - 1);
        return ksLine(gw, pt, size/3, t, levels - 1);
    }
}

void kochSnowflake(GWindow& gw, int x, int y, int size, int levels) {
    GPoint pt(x, y);
    pt = ksLine(gw, pt, size,    0, levels);
    pt = ksLine(gw, pt, size, -120, levels);
    pt = ksLine(gw, pt, size,  120, levels);
}
```

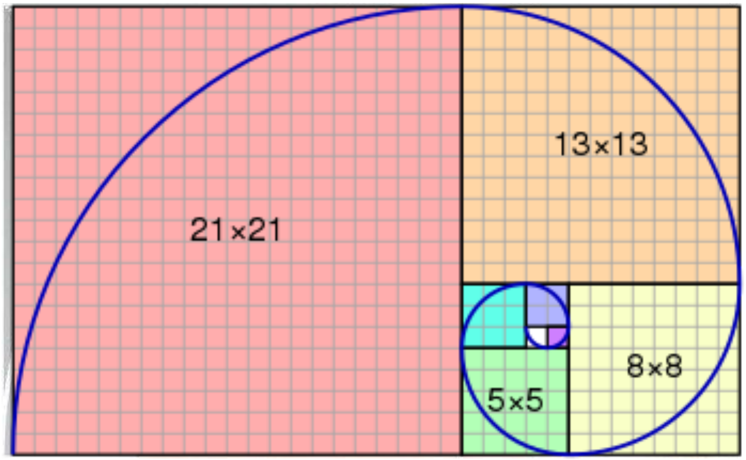
Fibonacci exercise

crawl

- Write a recursive function **`fib`** that accepts an integer *N* and returns the *N*th Fibonacci number.
  - The first two Fibonacci numbers are defined to be 1.
  - Every other Fibonacci number is the sum of the two before it.

    (Don't worry about integer overflow.)

```
fib(1) => 1
fib(2) => 1
fib(3) => 2
fib(4) => 3
fib(5) => 5
fib(6) => 8
fib(7) => 13
fib(8) => 21
fib(9) => 34
...
```



Bad fib solution

```
// Returns the nth Fibonacci number.
int fib(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

// what does the call stack look like?
```

Memoization

- **memoization**: Caching results of previous expensive function calls for speed so that they do not need to be re-computed.
  - Often implemented by storing call results in a collection.

- Pseudocode template:

```
cache = {}.          // empty

function f(args):
    if I have computed f(args) before:
        Look up f(args) result in cache.
    else:
        Actually compute f(args) result.
        Store result in cache.
    Return result.
```

Wrapper Functions

- We don't want the user to have to worry about the cache!
  - Alternative to the default parameters we saw yesterday
- Some recursive functions need extra arguments to implement the recursion
- A **wrapper function** is a function that does some initial prep work, then fires off a recursive call with the right arguments.
  - Might be good to know
- The recursion is done in the **helper** function

Memoized fib solution

```
// Returns the nth Fibonacci number.
// This version uses memoization.
int fib(int n) { // wrapper function
    Map<int, int> cache;
    return fibHelper(n, cache);
}

int fibHelper(int n, Map<int, int> &cache) {
    if (n <= 2) {
        return 1;
    } else if (cache.containsKey(n)) {
        return cache[n];
    } else {
        int result = fib(n - 1) + fib(n - 2);
        cache[n] = result;
        return result;
    }
}
```

Overflow (extra) slides

Tail recursion

- **tail recursion**: When a recursive call is made as the final action of a recursive function.
  - Tail recursion can often be **optimized** by the compiler.
    - Qt Creator: "Release" mode, not "Debug" mode
  - Are these tail recursive?

```
int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

```
int fact(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

Tail-recursive factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.
int factorial(int n, int accum = 1) {
    if (n <= 1) {
        return accum;
    } else {
        return factorial(n - 1, accum * n);
    }
}
```

– Tail recursive solutions often end up passing partial computations as parameters that would otherwise be computed after the recursive call.

Non-recursive factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.
int factorial(int n) {
    int accum = 1;
    for (int i = 1; i <= n; i++) {
        accum *= i;
    }
    return accum;
}
```

- Sometimes looking at the non-recursive version of a function can help you find the tail recursive solution.
  - Often looks more like the non-recursive version, with a variable or parameter keeping track of partial computations.
  - Loop is replaced by recursive call.