CS106X
Winter 2017

Handout
Feb 17th, 2017

Recursion

Definition

In order to understand recursion, you first need to understand recursion! By definition, recursion is any solution technique in which large problems are solved by reducing them to smaller, virtually identical sub-problems. The structure of recursion follows a common template and you can apply recursion as long as your problem meets the following conditions:

1. You can identify **simple cases** for which the answer is easily determined
2. You can identify a **recursive decomposition** that allows you to break any complex instance of the problem into simpler problems of the same form

The classic introductory example employing recursion is an implementation of the **factorial** function:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

The three "musts" of recursion are that :

1. Your code must have a case for all valid inputs.

2. You must have a base case (makes no recursive calls).

3. When you make a recursive call it should be to a simpler instance (forward progress towards base case)

In some cases, the problem to be solved is so simple that we can return or terminate execution without any further recursive calls. The first of the two lines in **factorial** is an example of such a base case—**factorial(0)** is always **1**. However, whenever the specified integer **n** is larger than **0**, it helps to calculate **factorial(n - 1)** and multiply the result of that computation by **n** itself. That's

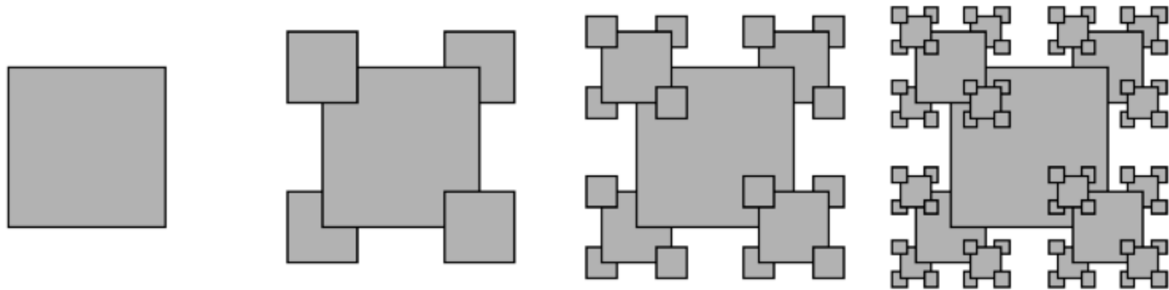
precisely what the recursive call is doing. Now, we can look at some examples of how recursion can create spectacular fractals!

Fractals: Boxy Snowflakes

Assume you're given the following function, which draws a shaded square of the specified dimension with a solid border, centered at (cx, cy):

```
void drawFilledBox(GWindow& window, double cx, double cy, double side,
                   const string& fillcolor, const string& bordercolor);
```

Presented below is the recursive implementation of **`drawBoxyFractal`**, which is capable of drawing the following order-0, order-1, order-2, and order-3 fractals:



Note our implementation is sensitive to the way the centered squares are layered—clearly the sub-fractals drawn in the southwest and northeast corners are drawn before the large center square, which is drawn before the sub-fractals at 4:30 and 10:30. The same layering scheme is respected at all recursive levels:

```
void drawBoxyFractal(GWindow& window, double cx, double cy,
      double dimension, int order) {
      if (order >= 0) {
            drawBoxyFractal(window, cx - dimension/2, cy + dimension/2,
                            kScale * dimension, order - 1);
            drawBoxyFractal(window, cx + dimension/2, cy - dimension/2,
                            kScale * dimension, order - 1);
            drawFilledBox(window, cx, cy, dimension, "Gray", "Black");
            drawBoxyFractal(window, cx - dimension/2, cy - dimension/2,
                            kScale * dimension, order - 1);
            drawBoxyFractal(window, cx + dimension/2, cy + dimension/2,
                            kScale * dimension, order - 1);
      }
}
```