Use what you’ve got

Recursion

Lawrence Snyder
University of Washington
Recursion means to call a function in its own definition.
Recursion

- If the “concept applies,” use it
Recursion

- Recursion means to use a function in its own definition ... that is, there is one or more calls to the function in the body

Factorial 

\( n! = n \times (n-1) \times \ldots \times 1 \) is classic example:

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n \text{ is 0 or 1} \\
n \times \text{fact}(n-1) & \text{otherwise}
\end{cases}
\]

- Well-formed recursive functions have two (or more) cases: basis case, and a recursive case

- A recursive function must test to separate the “basis case,” that is, the non-recursive case, from the normal recursive case
Let’s See It In Processing

```java
void setup() {
    size(700, 200);
    background(0);
    frameRate(4);
    fill(255,255,0);
    for (int i=1; i<6; i++) {
        drawSquare(i);
    }
}

void drawSquare( int n ) {
    int reps = fact(n);
    for (int i=0; i < reps; i++) {
        rect( 10+i*5, 20+n*20, 3, 3);
    }
}

int fact (int n) {
    if (n == 1) {
        return 1;
    }
    return n*fact(n-1);
}
```
Recursion is abstraction ...

- Recall that when we abstract a rule for a sequence (like drawing 4 squares) ...

  ```java
  for (int i = 0; i < 4; i = i+1) {
    rect(100+100*i, 20, 50, 50)
  }
  ```

  we try to make each case differ in the same way ... allows work to be done automatically

- Recursion works the same way
Process an “L” corner ...

- From its position, Lightbot processes an “L”
- Returns to the same relative position
Often Many Approaches Work

- Notice that processing a “7” works, too
Having Gone Around and Back

- Ready to complete a final circuit
Same Idea: Sierpinski Triangle
What is it?

Abstracting, we have

- “A Sierpinski Triangle is an equilateral triangle”
- “A Sierpinski Triangle can also be three copies of a Sierpinski Triangle, touching at their corners”
Sierpinski Triangle

- What is it?
- Abstracting, we have
  - “A Sierpinski Triangle is an equilateral triangle”
  - “A Sierpinski Triangle can also be three copies of a Sierpinski Triangle, touching at their corners”

- What’s the base case? What’s the recursive case?
// Sierpinski.pde by Martin Prout
float T_HEIGHT = sqrt(3)/2;
float TOP_Y = 1/sqrt(3);
float BOT_Y = sqrt(3)/6;
float triangleSize = 800;

void setup(){
  size(int(triangleSize), int(T_HEIGHT*triangleSize));
  smooth();
  fill(255);
  background(0);
  noStroke();
  drawSierpinski(width/2, height *(TOP_Y/T_HEIGHT), triangleSize);
}

void drawSierpinski(float cx, float cy, float sz){
  if (sz < 5){ // Limit no of recursions on size
    drawTriangle(cx, cy, sz); // Only draw terminals
    noLoop();
  }
  else{
    float cx0 = cx;
    float cy0 = cy - BOT_Y * sz;
    float cx1 = cx - sz/4;
    float cy1 = cy + (BOT_Y/2) * sz;
    float cx2 = cx + sz/4;
    float cy2 = cy + (BOT_Y/2) * sz;
    drawSierpinski(cx0, cy0, sz/2);
    drawSierpinski(cx1, cy1, sz/2);
    drawSierpinski(cx2, cy2, sz/2);
  }
}

void drawTriangle(float cx, float cy, float sz){
  float cx0 = cx;
  float cy0 = cy - TOP_Y * sz;
  float cx1 = cx - sz/2;
  float cy1 = cy + BOT_Y * sz;
  float cx2 = cx + sz/2;
  float cy2 = cy + BOT_Y * sz;
  triangle(cx0, cy0, cx1, cy1, cx2, cy2);
}
Sierpinski Triangle

void setup(){
  size(int(triangleSize),int(T_HEIGHT*triangleSize));
  smooth();
  fill(255);
  background(0);
  noStroke();
  drawSierpinski(width/2, height * (TOP_Y/T_HEIGHT), triangleSize);
}

void drawSierpinski(float cx, float cy, float sz){
  if (sz < 5){ // Limit no of recursions on size
    drawTriangle(cx, cy, sz); // Only draw terminals
    noLoop();
  } else{
    float cx0 = cx;
    float cy0 = cy - BOT_Y * sz;
    float cx1 = cx - sz/4;
    float cy1 = cy + (BOT_Y/2) * sz;
    float cx2 = cx + sz/4;
    float cy2 = cy + (BOT_Y/2) * sz;
    drawSierpinski(cx0, cy0, sz/2);
    drawSierpinski(cx1, cy1, sz/2);
    drawSierpinski(cx2, cy2, sz/2);
  }
}

triangle(cx0, cy0, cx1, cy1, cx2, cy2);
Why Recursion Is So Beautiful ...

- Often we can solve a problem “top down”
- Finding Fibonacci numbers is classic example – 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  Each item is the sum of the two before it, except the first two which are both 1
- This definition translates directly:

\[
\text{fib}(n) = \begin{cases} 
  1 & \text{if } n < 2 \\
  \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]

- It works like all functions work
Leave The Thinking To The Agent ...

\[ fib(n) = \begin{cases} 
1 & \text{if } n < 2 \\
fib(n-1) + fib(n-2) & \text{otherwise}
\end{cases} \]

- Compute Fibonacci number 4:
  - \( fib(4) = fib(3) + fib(2) \)
    - \( fib(3) = fib(2) + fib(1) \)
      - \( fib(2) = fib(1) + fib(0) = 1 + 1 = 2 \)
      - \( = 2 + 1 = 3 \)
      - \( = 3 + fib(2) \)
    - \( fib(2) = fib(1) + fib(0) = 1 + 1 = 2 \)
    - \( = 3 + 2 = 5 \)

Programmers don’t need to worry about the details if the definition is right and the termination is right; the computer does the rest.
Recall that each parameter is created with a function call, and initialized; when the function is over, it is thrown away ... which means: “inner” params hide “outer” parameters.

Think about \texttt{fact(4)}

```c
int fact (int n) {
    if (n == 1) {
        return 1;
    }
    return n*fact(n-1);
}
```
Recursion is a big deal because it is so elegant; as a result information is everywhere – e.g. see Wikipedia

See Processing Ref for this cute program
Random Circles ...
```cpp
void setup() {
  size(800, 800);
  background(200, random(255), random(255));
  noStroke();
  smooth();
  drawCircle(400, 400, 80, 7);
}

void drawCircle(float x, float y, int radius, int level) {
  fill(random(255), random(255), random(255));
  ellipse(x, y, radius*2, radius*2);
  if (level > 1) {
    level = level - 1;
    int num = int(random(2, 6));
    for (int i=0; i<num; i++) {
      float a = random(0, TWO_PI);
      float nx = x + cos(a) * 20.0 * level;
      float ny = y + sin(a) * 20.0 * level;
      drawCircle(nx, ny, radius/2, level);
    }
  }
}
```
All Circles From One Call Equal