Use what you’ve got

Recursion

Lawrence Snyder
University of Washington
Announcements

- Upcoming events ... it’ll be cool
- Availability survey
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:

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

- 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.
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);
}

..  
..  
.....

........................

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

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
Sierpinski Triangle

```c
// 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 + TOP_Y * sz;
    float cx2 = cx + sz/2;
    float cy2 = cy + TOP_Y * sz;
    triangle(cx0, cy0, cx1, cy1, cx2, cy2);
}
```
Sierpinski Triangle

```java
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);
    }
}
```

```java
triangle(cx0, cy0, cx1, cy1, cx2, cy2);
```
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
\[
\text{fib}(n) = \begin{cases} 
1 & \text{if } n < 2 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]

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

Programmers don’t need to worry about the details if the definition is right and the termination is right; the computer does the rest.
Making It All Work

- 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 \textit{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
Give all ways to arrange \( k \) things in sequences of length \( n \)

- The key idea is to be orderly about how you do it
- Here we have two kinds of hearts and they are in sequences of 4
  - Notice
    - the top half are red
    - bottom half are white ... recursively
void addon(int span, int base, String nextdigit) {
    for (int i = 0; i < span; i++) {
        seq[i+base] = seq[i+base] + nextdigit;
    }
    if (span > 1) {
        addon(span/2, base, binDigits[0]);
        addon(span/2, base+span/2, binDigits[1]);
    }
}

//Prepare for the caption
//Caption, line 1
//Prepare to display UTF-8
//Initialize with empties

//Build the String sequences
//And print them
Watch The Enumeration ...

```java
void addon(int span, int base, String nextdigit) {
    for (int i = 0; i < span; i++) {
        seq[i+base] = seq[i+base] + nextdigit;
    }
    if (span > 1) {
        addon(span/2, base, binDigits[0]);
        addon(span/2, base+span/2, binDigits[1]);
    }
}
```

if (span > 8)

if (span > 4)

if (span > 2)
A Theoretical Fact

- It is a fact of computer science that loops like our for-loop are unnecessary for programming: All computation can be performed using recursion alone.

- It is actually fun and totally cool to program this way!

- In CSP we use recursion when it’s convenient.
Assignment: Enumerate

- Assignment 14 is to enumerate some strings

- Useful advice –
  - Read the whole assignment before starting
  - Study the solution in these slides, since they are similar