Eric Roberts
CSCI 121

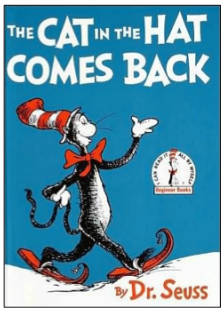
Handout #31
October 8, 2018

# Introduction to Recursion

## Introduction to Recursion

Eric Roberts
CSCI 121
October 8, 2018

## Recursion in Familiar Contexts




## Recursion in Familiar Contexts


## Recursion

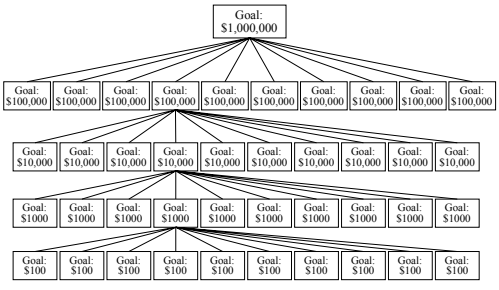
- ***Recursion*** is the process of solving a problem by dividing it into smaller subproblems *of the same form*. The italicized phrase is the essential characteristic of recursion; without it, all you have is a description of the more familiar strategy of decomposition or stepwise refinement.
- The fact that recursive decomposition generates subproblems that have the same form as the original problem means that recursive programs will use the same function or method to solve subproblems at different levels of the solution. In terms of the structure of the code, the defining characteristic of recursion is having functions that call themselves, directly or indirectly, as the decomposition process proceeds.

## A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a national campaign and need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the $100 range.
- Another strategy would be to ask 10,000 friends for $100 each. Unfortunately, most of us don't have 10,000 friends.
- There are, however, more promising strategies. You could, for example, find ten regional coordinators and ask each one to raise $100,000. Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of $10,000, continuing until the process reached the $100 level.

## A Simple Illustration of Recursion

The following diagram illustrates the recursive strategy for raising $1,000,000 described on the previous slide:



## A Pseudocode Fundraising Strategy

If you were to implement the fundraising strategy in the form of a Python function, it would look something like this:

```
def collectContributions(n):
    if n <= 100:
        Collect the money from a single donor.
    else:
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
```

What makes this strategy recursive is that the line

*Get each volunteer to collect n/10 dollars.*

will be implemented using the following recursive call:

```
collectContributions(n / 10);
```

## Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

## The Towers of Hanoi

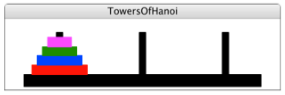
In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

—Henri de Parville, *La Nature,* 1883

## Solving the Towers of Hanoi

```
def TowersOfHanoi():
    def moveTower(n, start, finish, tmp):
        if (n == 1):
            moveSingleDisk(start, finish)
        else:
            moveTower(n - 1, start, tmp, finish)
            moveSingleDisk(start, finish)
            moveTower(n - 1, tmp, finish, start)
```

| n | start | finish | tmp |
|---|---|---|---|
| | | | |




## The Recursive "Leap of Faith"

- The purpose of going through the complete decomposition of the Towers of Hanoi problem is to convince you that the process works and that recursive calls are in fact no different from other method calls, at least in their internal operation.
- The danger with going through these details is that it might encourage you to do the same when you write your own recursive programs. As it happens, tracing through the details of a recursive program almost always makes such programs harder to write. Writing recursive programs becomes natural only after you have confidence in the process.
- As you write a recursive program, it is important to believe that any recursive call will return the correct answer as long as the arguments define a simpler subproblem. Believing that to be true—even before you have completed the code—is called the ***recursive leap of faith.***

## The Recursive Paradigm

- Most recursive functions you encounter in an introductory course have bodies that fit the following general pattern:

```
if test for a simple case:
    Compute and return the simple solution without using recursion.
else:
    Divide the problem into one or more subproblems that have the same form.
    Solve each of the problems by calling this method recursively.
    Return the solution from the results of the various subproblems.
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify ***simple cases*** that can be solved without recursion.
  2. Find a ***recursive decomposition*** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

## Recursive Checklist

- Does your recursive implementation begin by checking for simple cases?
- Have you solved the simple cases correctly?
- Does your recursive decomposition make the problem simpler?
- Does the simplification process eventually reach the simple cases, or have you left out some of the possibilities?
- Do the recursive calls in your method represent subproblems that are truly identical in form to the original?
- Do the solutions to the recursive subproblems provide a complete solution to the original problem?

## Generating Mondrian-Style Paintings

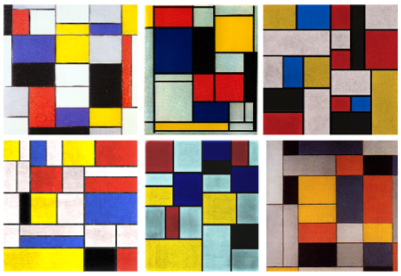
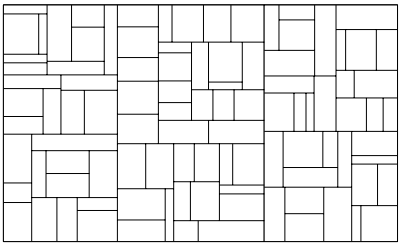


**Fig. 11:** Three real Mondrian paintings, and three samples from our targeting function. Can you tell which is which?

Source: Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun, "Metropolis Procedural Modeling," *ACM Transactions on Graphics*, April 2019.

## Mondrian Decomposition



## Code for the Mondrian Program

```
# File: Mondrian.java

"""
This program draws a recursive Mondrian style picture by recursively
subdividing the plane.
"""

from pgl import GWindow, GLine
import random

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
MIN_AREA = 10000
MIN_EDGE = 20

def Mondrian():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    subdivide(gw, 0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

## Code for the Mondrian Program

```
# At each level, subdivide first checks for the simple case, which
# is when the size of the rectangular canvas is too small to subdivide
# (i.e., when the area is less than MIN_AREA).  In the simple case,
# the method does nothing.  In the recursive case, the method splits the
# canvas along its longest dimension by choosing a random dividing line
# that leaves at least MIN_EDGE on each side.  The program then uses
# a divide-and-conquer strategy to subdivide the two rectangles.

def subdivide(gw, x, y, width, height):
    if width * height >= MIN_AREA:
        if width > height:
            dx = random.uniform(MIN_EDGE, width - MIN_EDGE)
            gw.add(GLine(x + dx, y, x + dx, y + height))
            subdivide(gw, x, y, dx, height)
            subdivide(gw, x + dx, y, width - dx, height)
        else:
            dy = random.uniform(MIN_EDGE, height - MIN_EDGE)
            gw.add(GLine(x, y + dy, x + width, y + dy))
            subdivide(gw, x, y, width, dy)
            subdivide(gw, x, y + dy, width, height - dy)
```

## ColorMondrian Decomposition