CMSC201
Computer Science I for Majors
Lecture 19 – Recursion

Prof. Jeremy Dixon

Based on slides from the book author, and previous iterations of the course
Last Class We Covered

• Project 1 Details
• Classes
• Inheritance
Any Questions from Last Time?
Today’s Objectives

• To introduce recursion
• To begin to learn how to “think” recursively
• To better understand the concept of stacks
Introduction to Recursion
M.C. Escher: "Drawing Hands" (1948)
What is Recursion?

• In computer science, recursion is a way of thinking about and solving problems

• It’s actually one of the central ideas of CS

• Solving a problem using recursion means the solution depends on solutions to smaller instances of the same problem
Recursive Procedures

• When creating a recursive procedure, there are a few things we want to keep in mind:
  – We need to break the problem into smaller pieces of itself
  – We need to define a “base case” to stop at
  – The smaller problems we break down into need to eventually reach the base case
Normal vs Recursive Functions

• So far, we’ve had functions call other functions
  – For example, `main()` calls the `square()` function

• A recursive function, however, calls itself
Why Would We Use Recursion?

• In computer science, some problems are more easily solved by using recursive methods
• For example:
  – Traversing through a directory or file system
  – Traversing through a tree of search results
  – Some sorting algorithms recursively sort data
• For today, we will focus on the basic structure of using recursive methods
Simple Recursion Example

```python
def compute(intInput):
    print(intInput)
    if (intInput > 2):
        compute(intInput-1)

def main():
    compute(50)

main()
```

This program simply computes from 50 down to 2.

This is where the recursion occurs.

You can see that the `compute()` function calls itself.
Visualizing Recursion

• To understand how recursion works, it helps to visualize what’s going on.

• To help visualize, we will use a common concept called the *Stack*.

• A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  – Push: you can push something onto the stack.
  – Pop: you can pop something off the top of the stack.

• Let’s see an example stack in action.
Stacks
Stacks

• The diagram below shows a stack over time.
• We perform two pushes and two pops.
Stacks

• In computer science, a stack is a last in, first out (LIFO) abstract data type and data structure.

• A stack can have any abstract data type as an element, but is characterized by only two fundamental operations, the push and the pop.

• The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty.
Stacks

- The nature of the pop and push operations also means that stack elements have a natural order.
- Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.
Stacks and Functions

• When you run a program, the computer creates a stack for you.
• Each time you invoke a function, the function is placed on top of the stack.
• When the function returns or exits, the function is popped off the stack.
Stacks and Functions

Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: square()

Time 3:
Pop: square() returns a value. Method exits.

Time 4:
Pop: main() returns a value. Method exits.

This is called an activation record or stack frame.

Usually, this actually grows downward.
Stacks and Recursion

• Each time a function is called, you *push* the function on the stack.
• Each time the function returns or exits, you *pop* the function off the stack.
• If a function calls itself recursively, you just push another copy of the function onto the stack.
• We therefore have a simple way to visualize how recursion really works.
def compute(intInput):
    print(intInput)
    if (intInput > 2):
        compute(intInput-1)

def main():
    compute(50)

main()

Here’s the code again. Now, that we understand stacks, we can visualize the recursion.
Stack and Recursion in Action

Inside compute(9):
print (intInput);
if (intInput < 2)
  compute(intInput-1);

Inside compute(8):
print (intInput);
if (intInput < 2)
  compute(intInput-1);

Inside compute(7):
print (intInput);
if (intInput < 2)
  compute(intInput-1);

After returning from compute(2)
pop everything
Defining Recursion
def f(n):
    if n == 1:
        return 1
    else:
        return f(n - 1)

"Useful" recursive functions have:
• at least one recursive case
• at least one base case
so that the computation terminates
Recursion

def f(n):
    if n == 1:
        return 1
    else:
        return f(n + 1)

Find $f(5)$

We have a base case and a recursive case. What's wrong?
Recursion

The recursive case should call the function on a *simpler input*, bringing us closer and closer to the base case.
Recursion

def f(n):
    if n == 0:
        return 0
    else:
        return 1 + f(n - 1)

Find f(0)
Find f(1)
Find f(2)
Find f(100)
Recursion

```python
def f(n):
    if n == 0:
        return 0
    else:
        return n + f(n - 1)

f(3)
3 + f(2)
3 + 2 + f(1)
3 + 2 + 1 + f(0)
3 + 2 + 1 + 0
6
```
Factorial

• $4! = 4 \times 3 \times 2 \times 1 = 24$
Factorial

• Does anyone know the value of 9?
  • 362,880

• Does anyone know the value of 10?
  • How did you know?
Factorial

• $9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$

• $10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$

• $10! = 10 \times 9!$

• $n! = n \times (n - 1)!$

• That's a recursive definition!
def fact(n):
    return n * fact(n - 1)

fact(3)
3 × fact(2)
3 × 2 × fact(1)
3 × 2 × 1 × fact(0)
3 × 2 × 1 × 0 × fact(-1)
...

Factorial
Factorial

• What did we do wrong?

• What is the base case for factorial?
Any Other Questions?
Announcements

• Lab has been cancelled this week!
  – Work on your project instead

• Project 1 is out
  – Due by Tuesday, November 17th at 8:59:59 PM
  – Do NOT procrastinate!

• Next Class: Recursion