CMSC201
Computer Science I for Majors
Lecture 20 – Recursion
Last Class We Covered

• Python’s standard library
• Importing modules

• “Random” numbers
  – Pseudo randomness
  – Seeding the RNG
  – Generating random numbers/choices
    • Three different methods
Any Questions from Last Time?
Today’s Objectives

• To introduce recursion

• To better understand the concept of “stacks”

• To begin to learn how to “think recursively”
  – To look at examples of recursive code
  – Summation, factorial, etc.
Introduction to Recursion
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

• In recursion, the solution depends on solutions to smaller instances of the same problem
Recursive Solutions

• When creating a recursive solution, there are a few things we want to keep in mind:
  1. We need to break the problem into smaller pieces of itself
  2. We need to define a “base case” to stop at
  3. 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
Toy Example of Recursion

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

def main():
    compute(50)

main()
```

What does this program do?

This program prints the numbers 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.

• Python uses a **stack** to keep track of function calls

• A stack is an important computer science concept
Stacks
Stacks

• A stack is like a bunch of lunch trays in a cafeteria
• It has only two operations:
  – Push
    • You can push something onto the top of the stack
  – Pop
    • You can pop something off the top of the stack

• Let’s see an example stack in action.
Stack Example

- The diagram below shows a stack over time
- We perform two pushes and two pops
Stack Details

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

• It can store any type of data, but has only two operations: push and pop

• Push adds to the top of the stack, hiding anything else on the stack

• Pop removes the top element from the stack
Stack Details

• 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.
  – The lower elements are those that have been in the stack the longest.
Stacks and Functions

• When you run your program, the computer creates a stack for you

• Each time you call a function, the function is pushed onto the top of the stack

• When the function returns or exits, the function is popped off the stack
Stacks and Functions Example

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.
Stacks and Recursion

• If a function calls itself recursively, you push another call to the function onto the stack

• We now have a simple way to visualize how recursion really works
Toy Example of Recursion

```python
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);
→ 9

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

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

After returning from compute(2)
pop everything
Defining Recursion
“Cases” in Recursion

• A recursive function must have two things:

• At least one base case
  – When a result is returned (or the function ends)
  – “When to stop”

• At least one recursive case
  – When the function is called again with new inputs
  – “When to go (again)”
def f(n):
    if n == 1:
        return 1
    else:
        return f(n - 1)
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

```python
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

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

• What did we do wrong?

• What is the base case for factorial?
Announcements

• Project 1 is/was due Wednesday

• Homework 8 is/was released Wednesday night
  – Last homework of the semester
  – Due the Wednesday before Thanksgiving
  • Plan ahead!