CMSC201
Computer Science I for Majors
Lecture 20 – Recursion (Continued)

Prof. Jeremy Dixon

Based on slides from UPenn’s CIS 110, and from previous iterations of the course
Last Class We Covered

• Stacks
• Recursion
  – Recursion
    • Recursion
• Parts of a recursive function:
  – Base case: when to stop
  – Recursive case: when to go (again)
Any Questions from Last Time?
Today’s Objectives

• To gain a more solid understanding of recursion
• To explore what goes on “behind the scenes”
• To examine individual examples of recursion
  – Binary Search
  – Hailstone problem (Collatz)
  – Fibonacci Sequence
• To better understand when it is best to use recursion, and when it is best to use iteration
Review of Recursion
What is Recursion?

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

• In other words, to define a function or calculate a number by the repeated application of an algorithm
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
“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)”
Code Tracing: Recursion
Stacks and Tracing

• Stacks will help us track what we are doing when tracing through recursive code

• Remember, stacks are LIFO data structures
  – Last In, First Out

• We’ll be doing a recursive trace of the summation function
Summation Function

• The addition of a sequence of numbers
• The summation of a number is that number plus all of the numbers less than it (down to 0)
  – Summation of 5: $5 + 4 + 3 + 2 + 1$
  – Summation of 6: $6 + 5 + 4 + 3 + 2 + 1$
• What would a recursive implementation look like? What’s the base case? Recursive case?
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

Base case:
Don’t want to go below 0
Summation of 0 is 0

Recursive case:
Otherwise, summation is
num + summation(num-1)
main()

```python
def main():
    summ(4)
```

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)
main()

def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)
```python
def main():
    summ(4)
    num = 4

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)
```

```
main()
```
```python
def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)
```

This is a local variable. Each time the `summ()` function is called, the new instance gets its own unique local variables.
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

main()
def main():
    summ(4)
    num = 4

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)
    num = 3
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def main():
    summ(4)
def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

main()
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

main()
def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 4

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 2
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 3
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 1
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 0
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 1
```python
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def main():
    summ(4)

main()
```

**STACK**

- summ(4)
- summ(3)
- summ(2)
- summ(1)
- summ(0)
- main()
```python
# Define the summ function
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

# Main function
def main():
    summ(4)

# Call the main function
main()
```
```python
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

num = 4
num = 3
num = 2
num = 1

main()

return 1 + 0 (= 1)
```
main()

def main():
    summ(4)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

return 2 + 1 (= 3)
main()

def main():
    summ(4)
    num = 4

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num - 1)

def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num - 1)

num = 4

return 3 + 3 (= 6)
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

main()
def main():
    summ(4)

return 4 + 6 (=10)
```python
def main():
    return None
```

return None return None

POP! POP! POP! POP! POP! POP! POP!
return control

The stack is empty!
Returning and Recursion
Returning Values

• If your goal is to return a final value
  – Every recursive call must return a value
  – You must be able to pass it “back up” to `main()`
  – In most cases, the base case should return as well

• Must pay attention to what happens at the “end” of a function.
```python
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num-1)

def main():
    summ(4)

main()
```

Does this work? What's wrong?
Hailstone Example
The Hailstone Problem

• Included as part of the “Nested and While Loops” homework assignment

• The problem is actually known as the “Collatz Conjecture”
Rules of the Collatz Conjecture

• Three rules to govern how it behaves
  – If the current height is 1, quit the program
  – If the current height is even, cut it in half (divide by 2)
  – If the current height is odd, multiply it by 3, then add 1

• This process has also been called HOTPO
  – Half Or Triple Plus One
Implementation

• In your homework, you implemented this process using a **while** loop

• Can you think of another way to implement it?
  • Recursively!
Designing our Recursive Function

• What is our base case?
  – When the **Height is 1**

• What is our recursive case?
  – We have two! What are they?
    – **Height is even:** divide by 2
    – **Height is odd:** multiply by 3 and add 1
Exercise

• Create a function `hail()` that takes in a number and prints out the height of the hailstone at each point in time

• Important considerations:
  – What do we check first? Base or recursive case?
  – Is this function returning anything? Why or why not?
Exercise Details

• Rules for function behavior
  – If the current height is 1, quit the program
  – If the current height is even, cut it in half (divide by 2)
  – If the current height is odd, multiply it by 3, then add 1

• Create a function `hail()` that
  – Takes in a number
  – Prints out the height of the hailstone each time
Binary Search
Searching

• Given a list of sorted elements (e.g., words), find a specific word as quickly as possible

• We could start from the beginning and iterate through the list until we find it
  — But that could take a long time!
Binary Search

• Uses a “divide and conquer” approach

• Go to the middle, and compare the element there to the one we’re looking for
  – If it’s larger, we know it’s not in the last half
  – If it’s smaller, we know it’s not in the first half

  – If it’s the same, we found it!
Binary Search Example

Find "J"
Binary Search Example

Find “V”
Binary Search

• Can be implemented using a `while` loop
  – But much more common to use recursion

• What is the base case?
• What is the recursive case?
Recursion vs Iteration
Recursion and Iteration

• Both are important
  – All modern programming languages support them
  – Some problems are easy using one and difficult using the other

• How do you decide which to use?
Use Iteration When...

• Speed and efficiency is an issue

• The problem is an obvious fit for iteration
  – Processing every element of a list (or 2D list)
Use Recursion When...

• Speed is not an issue
• The data being processed is recursive
  – A hierarchical data structure
• A recursive algorithm is obvious
• Clarity and simplicity of code is important
Fibonacci Sequences
Fibonacci Sequence

- Number series
- Starts with 0 or 1
- Next number is found by adding the previous two numbers together
- Pattern is repeated over and over (and over...)

www.umbc.edu
Fibonacci Sequence

• Starts with 0, 1, 1
• Next number is ...?

0 1 1 2 3 5 8 13 21 34 55
89 144 233 377 610 987 ...
Recursively Implement Fibonacci

• The formula for a number in the sequence:
  \[ F(n) = F(n-1) + F(n-2) \]

• What is our base case?
• What is our recursive case?

• How would we code this up?
Any Other Questions?
Announcements

• Lab is back in session this week!
  – Lab 11 is on classes

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

• Next Class: Dictionaries