Last Class We Covered

- Recursion
  - Recursion
  - Recursion
- Stacks
- 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
  – 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()

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

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

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

num = 4

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

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

def main():
    summ(4)

main()```
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)

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

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

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

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

main()
```
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)
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 = 2
return 1
def main():
    summ(4)
    num = 4
num = 3
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)
num = 3
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 = 1
return 1

return 1 + 0 (= 1)

STACK
main()
summ(4)
summ(1)
summ(2)
summ(3)
summ(4)
main()
```python
def main():
    summ(4)

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

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

### Stack

- `summ(4)`
- `summ(3)`
- `summ(2)`

### Execution Flow

1. `main()` calls `summ(4)`
2. `summ(4)` evaluates to `3 + 1` (stack trace)
3. `main()` returns `2 + 1` (stack trace)

**Final Result:**

```
return 2 + 1 (= 3)
```
```python
def summ(num):
    if num == 0:
        return 0
    else:
        return num + summ(num - 1)

def main():
    summ(4)
    print(3 + 3)  # (= 6)

main()
```

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

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

return 4 + 6 (=10)
def main():
    summ(4)

return None
The stack is empty!

return control
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 \texttt{main()}
  – In most cases, the base case should return as well

• Must pay attention to what happens at the “end” of a function.
def main():
    summ(4)

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

def main():
    summ(4)

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

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

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

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

Does this work? What's wrong?
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 the letter “J” using binary search
Binary Search Example

• Find the letter “J” using binary search
Binary Search Example

- Find the letter “J” using binary search
Binary Search Example

• Find the letter “J” using binary search
Binary Search Example

- Find the letter “G” using binary search
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
  – Iteration doesn’t push things onto the stack

• 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...)
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 ...
Time for...

LIVECODING!!!
Recursively Implement Fibonacci

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

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

• Final is Thursday, December 15th (3:30 – 5:30)
  – If you have a conflict, fill out the Google Form
  – https://goo.gl/forms/We4cMotoNdbmCAoh2

• Homework 8 is out now
  – Last homework of the semester
  – Due this Wednesday – plan ahead!

• Project 2 will come out after Thanksgiving break