Lecture 5: Recursion
Administrative Issues

• Where is Lecture 4? See Last slides.

• Labs are to help you learn the materials, so please make full use of them

• Materials for midterm go through March 4th Lecture.
Computational Concepts Toolbox

- Data type: values, literals, operations,
  - e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
  - indexing
- Data structures
- Tuple assignment
- Call Expressions
- Function Definition Statement
- Conditional Statement
- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
Today: Recursion

re·cur·sion

/nərˈkɜːrzhən/ 〈noun〉

noun  MATHEMATICS  LINGUISTICS

the repeated application of a recursive procedure or definition.

• a recursive definition.
  plural noun: recursions

re·cur·sive

/nərˈkɜərsiv/ 〈adjective〉

adjective

characterized by recurrence or repetition, in particular.

• MATHEMATICS  LINGUISTICS
  relating to or involving the repeated application of a rule, definition, or procedure to successive results.

• COMPUTING
  relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions.

• Recursive function calls itself, directly or indirectly
Review: Functions

- Generalizes an expression or set of statements to apply to lots of instances of the problem
- A function should *do one thing well*

```python
def concat(str1, str2):
    return str1 + str2;

concat("Hello","World")
```
Review: Higher Order Functions

• Functions that operate on functions
• A function

```python
def odd(x):
    return x%2

>>> odd(3)
1
```

• A function that takes a function arg

```python
def filter(fun, s):
    return [x for x in s if fun(x)]

>>> filter(odd, [0,1,2,3,4,5,6,7])
[1, 3, 5, 7]
```

Why is this not ‘odd’?
Review Higher Order Functions (cont)

• A function that returns (makes) a function

```python
def leq_maker(c):
    def leq(val):
        return val <= c
    return leq

>>> leq_maker(3)
<function leq_maker.<locals>.leq at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0,1,2,3,4,5,6,7])
[0, 1, 2, 3]
```

2/25/19
Review: One more example

- What does this function do?

def split_fun(p, s):
    """Returns <you fill this in>."""
    return [i for i in s if p(i)], [i for i in s if not p(i)]

>>> split_fun(leq_maker(3), [0,1,2,3,4,5,6])
([0, 1, 2, 3], [4, 5, 6])
Function Review

• A function cannot...

A) have a function as argument
B) define a function within itself
C) return a function
D) call itself
E) None of the above.

Solution:
E) A, B, C, D are all possible!
Function Review

• A Python function can...

A) not return a value
B) return different values for the same input
C) halt the entire program
D) change global variables
E) All of the above.

Solution:
E) A, B, C, D are all possible!
Recall: Iteration

```python
def sum_of_squares(n):
    accum = 0
    for i in range(1, n+1):
        accum = accum + i*i
    return accum
```

1. Initialize the “base” case of no iterations
2. Starting value
3. Ending value
4. New loop variable value
Recursion Key concepts – by example

1. Test for simple “base” case
2. Solution in simple “base” case
3. Assume recursive solution to simpler problem
4. Transform solution of simpler problem into full solution

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```
In words

- The sum of no numbers is zero
- The sum of $1^2$ through $n^2$ is the
  - sum of $1^2$ through $(n-1)^2$
  - plus $n^2$

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```
Why does it work

sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
# => sum_of_squares(1) + 2**2 + 3**2
# => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#
# => 0 + 1**2 + 2**2 + 3**2 = 14
How does it work?

• Each recursive call gets its own local variables
  – Just like any other function call

• Computes its result (possibly using additional calls)
  – Just like any other function call

• Returns its result and returns control to its caller
  – Just like any other function call

• The function that is called happens to be itself
  – Called on a simpler problem
  – Eventually bottoms out on the simple base case

• Reason about correctness “by induction”
  – Solve a base case
  – Assuming a solution to a smaller problem, extend it
Questions

• In what order do we sum the squares?
• How does this compare to iterative approach?

```python
def sum_of_squares(n):
    accum = 0
    for i in range(1, n+1):
        accum = accum + i*i
    return accum
```

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return n**2 + sum_of_squares(n-1)
```
Tail Recursion

• All the work happens on the way down the recursion
• On the way back up, just return

```python
def sum_up_squares(i, n, accum):
    """Sum the squares from i to n in incr. order""
    if i > n:
        Base Case
    else:
        Tail Recursive Case

>>> sum_up_squares(1,3,0)
14
```
Local variables

- Each call has its own “frame” of local variables
- What about globals?
- Let’s see the environment diagrams (next lecture)

```python
def sum_of_squares(n):
    n_squared = n**2
    if n < 1:
        return 0
    else:
        return n_squared + sum_of_squares(n-1)
```

https://goo.gl/CiFaUJ
**Iteration vs Recursion**

For loop:

```python
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```

UCB CS88 Sp19 L5
While loop:

def sum(n):
    s=0
    i=0
    while i<n:
        i=i+1
        s=s+i
    return s
Recursion:

def sum(n):
    if n==0:
        return 0
    return n+sum(n-1)
For Homework

\[ \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \]
where \( \text{fibonacci}(1) = \text{fibonacci}(0) = 1 \)
Another Example

```python
def first(s):
    """Return the first element in a sequence."""
    return s[0]
def rest(s):
    """Return all elements in a sequence after the first"""
    return s[1:]
def min_r(s):
    """Return minimum value in a sequence."""
    if len(s) == 1:
        return first(s)
    else:
        return min(first(s), min_r(rest(s))
```

- Recursion over sequence length, rather than number magnitude

indexing an element of a sequence

Slicing a sequence of elements

Base Case

Recursive Case
Visualize its behavior (print)

```python
In [104]:
def min_r(s):
    print('min_r:', s)
    if len(s) == 1:
        return first(s)
    else:
        result = min(first(s), min_r(rest(s)))
    print('min_r:', s, " => ", result)
    return result

In [105]:
min_r([3,4,2,5,11])
```

- What about sum?
- Don’t confuse print with return value
• The recursive “leap of faith” works as long as we hit the base case eventually

What happens if we don’t?
Recursion

• Recursion is...

A) Less powerful than a for loop
B) As powerful as a for loop
C) As powerful as a while loop
D) More powerful than a while loop
E) Just different all together

Solution:
C) Any recursion can be formulated as a while loop and any while loop can be formulated as a recursion (with a global variable).
Why Recursion?

• “After Abstraction, Recursion is probably the 2\textsuperscript{nd} biggest idea in this course”
• “It’s tremendously useful when the problem is self-similar”
• “It’s no more powerful than iteration, but often leads to more concise & better code”
• “It’s more ‘mathematical’”
• “It embodies the beauty and joy of computing”
• …
Why Recursion? More Reasons

• Recursive structures exist (sometimes hidden) in nature and therefore in data!
• It’s mentally and sometimes computationally more efficient to process recursive structures using recursion.
Recursion (unwanted)
Example I

List all items on your hard disk

- Files
- Folders contain
  - Files
  - Folders

Recursion!
def listfiles(directory):
    content = [os.path.join(directory, x) for x in os.listdir(directory)]

    dirs = sorted([x for x in content if os.path.isdir(x)])
    files = sorted([x for x in content if os.path.isfile(x)])

    for d in dirs:
        print d
        listfiles(d)

    for f in files:
        print f

Iterative version about twice as much code and much harder to think about.
Example II

Sort the numbers in a list.

Hidden recursive structure: Decision tree!
Computational Concepts Toolbox

- Data type: values, literals, operations,
  - e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
  - indexing
- Data structures
- Tuple assignment
- Call Expressions
  - Function Definition
  - Statement
- Conditional Statement
- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Recursion
• How many answers can be maximally responded to by 20 questions (how much data do I need on my game device)?

Assume a number of answer possibilities $b$. This gives $b^{20}$ possible answer paths.

In below device: $b=4$ ("unknown", "no", "yes", "sometimes") and $4^{20}=1,099,511,627,776$.

Even if each questions and each answer was only 1 byte long, the device would have to have **peta bytes** of memory.
• How can a 20-questions game get away with less?

Different answers lead to the same path (redundancy). For example, making $b$ effectively 2 \(\text{(instead of 4)}\) results in only $2^{20} = 1,048,576$ concepts. The 20-volume Oxford English Dictionary only describes 171,476 \(<4^9\) words. Typically, in our every-day life we deal with about 2000-4000 concepts \(<4^6\).

• How can you make a 20 questions game fail?

Pick a new concept “data science” or chose a random(!) one from the dictionary!
Thoughts for the Wandering Mind

The computer choses a random element $x$ of the list generated by `range(0,n)`. What is the smallest amount of iteration/recursion steps the best ever algorithms needs to guess $x$?

How would the algorithm look like?