Lecture #4: Recursion II and Higher Order Functions

Hackers steal medical data of US Olympic stars

This was: September 16, 2016

February 9th, 2018

http://inst.eecs.berkeley.edu/~cs88
Administrative issues

• Waitlist should be cleared.

• Based on your feedback: I will record optional lectures going deeper on data, code, algorithms, information, recursion, decision trees, run”time” complexity and other stuff.

• Speaking of recording: ETS will start capturing.
Computational Concepts today

- More on Recursion
- Runtime (preliminary)

- 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
Recursion is **not** iteration (i.e., loops)

- a) more powerful than
- b) just as powerful as
- c) less powerful than

Remember: Sanity Check…
Why Recursion?

- “After Abstraction, Recursion is probably the 2nd 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!
Tree Recursion makes Sorting Efficient

Break the problem into multiple smaller sub-problems, and solve them recursively

```python
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def qsort(s):
    """Sort a sequence - split it by the first element, sort both parts and put them back together."""
    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split(pivot, rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]```
QuickSort Example

[3, 3, 1, 4, 5, 4, 3, 2, 1, 17]

[3, 1, 3, 2, 1]  [4, 5, 4, 17]

[1, 3, 2, 1]  []  [4]  [5, 17]


[1]  [ ]  [2]  [ ]  [4]  [17]

[2, 3]  [ ]  [4]  [ ]  [5, 17]

[1, 1, 2, 3]  [4, 4, 5, 17]

[1, 1, 2, 3, 3]  [4, 4, 5, 17]

[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
More on Recursion

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

- The sum of no numbers is zero
- The sum of $1^2$ through $n^2$ is $n^2$ plus the sum of $1^2$ through $(n-1)^2$
Recap: 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:
        return accum  # Base Case
    else:
        return sum_up_squares(i+1, n, accum+i**2)  # Tail Recursive Case

>>> sum_up_squares(1,3,0)
14
```
How much ???

• “Time” is required to compute \( \text{sum\_of\_squares}(n) \)?
  – Recursively?
  – Iteratively?

• “Space” is required to compute \( \text{sum\_of\_squares}(n) \)?
  – Recursively?
  – Iteratively?

• Count the steps: Recursive is linear, iterative is constant! What about the order of evaluation?

• Careful: As taught traditionally, Computer Science has no measurement units convertible to physical time \((s)\) and space \((m^3)\)!
Recap: Defining Functions

• Generalizes an expression or set of statements to apply to lots of instances of the problem
• A function should do one thing well
Recap: Data or Code?
Higher Order Functions

• Functions that operate on functions
• A function

```python
def odd(x):
    return (x%2==1)

>>> odd(3)
True
```

• 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'?
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]
>>>
One more example

- **What does this function do?**

```python
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)]
```

```python
>>> split_fun(leq_maker(3), [0,1,2,3,4,5,6])
([0, 1, 2, 3], [4, 5, 6])
```
Three super important HOFS

map(function_to_apply, list_of_inputs)
Applies function to each element of the list

filter(condition, list_of_inputs)
Returns a list of elements for which the condition is true

reduce(function, list_of_inputs)
Reduces the list to a result, given the function
Recursion with Higher Order Fun

```python
def map(f, s):
    if not s:
        return []
    else:
        return [f(first(s))] + map(f, rest(s))

def square(x):
    return x**2

>>> map(square, [2, 4, 6])
[4, 16, 36]
```

- Divide and conquer
Using HOF to preserve interface

```python
def sum_of_squares(n):
    def sum_upper(i, accum):
        if i > n:
            return accum
        else:
            return sum_upper(i+1, accum + i*i)
    return sum_upper(1,0)
```

- What are the globals and locals in a call to `sum_upper`?
  - Try [python tutor](https://www.python.org)
- Lexical (static) nesting of function def within def - vs
- Dynamic nesting of function call within call
Recap: Quicksort

• Break the problem into multiple smaller sub-problems, and Solve them recursively

```python
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def qsort(s):
    """Sort a sequence - split it by the first element, sort both parts and put them back together."""
    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split(pivot, rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3, 3, 1, 4, 5, 4, 3, 2, 1, 17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```
def qsort(s):
    """Sort a sequence - split it by the first element, sort both parts and put them back together."""
    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split_fun(leq_maker(pivot), rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
How much ???

• “Time” is required to compute quicksort(s)?

• “Space” is required?

• Name of this recursion scheme?
  – Tree recursion

Logarithmic to len(s)
c*\log(len(s)) for some c