CS 88 Fall 2018 John Yang
Prof. David Culler Friday 2-4 PM

Recursion

Link to Discussion Slides: https://goo.gl/dCX1Yx
Link to Relevant Reading: http://composingprograms.com/pages/17-recursive-functions.html

1 Preface

Before diving into this worksheet, I highly recommend reading the text, going over the discussion slides, and making sure you understanding the foundations of recursion. This worksheet assumes you know the fundamental concepts and build on top of that in the form of problem solving strategies.

2 Problem Solving Approach

Here are some general guidelines that I've found to be helpful over the years when it comes to solving recursive problems.
1. Think about the problem inductively AKA break down the problem into a singular step.
2. What are your base case(s)? How many ways can your program terminate, and in what fashion?
3. Trust the recursion aka recursive leap of faith! This is related to tip #1. Assume your solution to a simpler problem works for a larger problem that repeats the same step a couple more times.
4. For loops and iteration are easier to visualize. If you're feeling stumped, write the iterative equivalent, then convert it to a recursive context!

3 Problems

3.1 Factorial

This first problem is a gentle beginning meant to reinforce the **similarities** between iterative and recursive solutions. If a problem has an iterative solution, it nearly *always* has a recursive solution!

```
def factorial(n):
    factorial = 1
    while n > 1:
        factorial = factorial * n
        n = n - 1
    return factorial
```

We're all familiar with good old Fibonacci! The iterative solution is given above as reference. Your task is to **write the recursive form of Factorial**. If you're stuck, here are a couple hints:
1. Infer the base case from the while loop's condition.
2. The code within the while loop forms your recursive case.

```
def factorial(n):
    if n <= 1:
        return return 1
    else:
        return return n * factorial(n-1)
```

3.2 Leap of Faith

In this next section, I'll be asking you to evaluate the output of different recursive functions. The premise for this exercise is that visualizing recursive processes can be difficult. If you breeze this section, good for you! The hope is that after completing these problems, you have a good grasp on the anatomy of recursive functions and are able to identify base cases + recursive cases effectively.

```
def fun1(x):
    if (x < 1):
        return "Done!"
    else:
        return str(x) + " " + fun1(x - 1)
```

Output of fun1(3):
3 2 1 Done!

```
def fun2(x, y):
    if (y == 2):
        return x
    else:
        return fun2(x, y - 1) + x
```

Output of fun2(3, 6):
15

```
def fun3(x):
    if (x < 1):
        return 1
    else:
        return x + fun3(x-4) + fun3(x-1)
```

Output of fun3(5):

23

3.3 Merge Baby Merge

Time for you to implement your own recursive function! Given two lists, 'a' and 'b', which are already in sorted order, write a recursive function that merges the two lists into one, maintaining its sorted order!

Feeling nervous? Take a deep breath, you got this! The problem description is short, and it can be hard to find a place to start. Try creating an example and walking through the merge process yourself. What is the step that's being repeated? That's your recursive case! When do you stop? That's your base case!

```
def mergeLists(a, b):
    if a == [ ]:
        return b
    elif b == [ ]:
        return a
    else:
        if a[0] < b[0]:
            return [a[0]] + mergeLists(a[1:], b)
        else:
            return [b[0]] + mergeLists(a, b[1:])
```

Hint: Given [1, 3, 5] and [2, 4, 6]. This sort takes you a millisecond to process, but inductively, what did you do? Laid out step by step, it'd probably look something like this.
1. Compare the first elements of the list.
2. Add whichever element is smaller to the new list.
3. Keep going until one of the lists becomes empty.
How do you convert this into recursion?

3.4 Step by Step

Last one! You've made it so far. Given a positive integer 'n' and a step size 'm', print out rows of 'x' characters corresponding to the values between 1 and 'n' in m intervals.

For example, if n = 9 and m = 3, the output should look like this:

```
x
xxxx
xxxxxxx
```

Another example, if n = 5 and m = 1, the output should look like this:

```
x
xx
xxx
xxxx
xxxxx
```

```
def steps(n, stepSize):
    # Write answer below
    helper(1, n, stepSize)

def helper(size, n, stepSize):
    if (size > n):
        return
    else:
        line = ""
        for x in range(size):
            line += "x"
        print(line)
        helper(size + stepSize, n, stepSize)
```