DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, October 19, 2020

Announcements

- Questions about the midterm grading or the fifth assignment?
- The sixth assignment is due Thursday, October 29th – questions?

Recursion

A **recursive method** is a method that solves a problem by making one or more calls to itself.

Recursion is a particularly helpful tool for certain problems.

Some programming languages are even primarily recursive in nature (e.g. Lisp and Scheme), where loops are almost completely absent from programs written in them.

Any recursive method consists of:

- **One or more base cases**: these are the portions of the problem for which an immediate solution is known
- **One or more recursive calls**: to avoid infinite recursion these subproblems must be in some way smaller than the original problem

The best way to learn recursion is to practice, a LOT. So we will see many examples during this part of the course.

Exercises

Problem 3 redux: Write a recursive function **makeCheer**() that on an integer parameter n will return a string of n-1 "Hip" strings followed by "Hurrah!".

```
>>> s = makeCheer(0)
>>> s
'Hurrah!'
>>> s = makeCheer(1)
>>> s
'Hurrah!'
>>> s = makeCheer(2)
>>> s
'Hip Hurrah!'
>>> s = makeCheer(3)
>>> s
'Hip Hip Hurrah!'
```

```
>>> s = makeCheer(6)
>>> s
'Hip Hip Hip Hip Hip Hurrah!'
>>>
```

Problem 4: Write a recursive function **pattern**() that prints a number pattern as shown below:

```
>>> pattern(1)
1
>>> pattern(2)
1 2 1
>>> pattern(3)
1 2 1 3 1 2 1
>>> pattern(4)
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
>>> pattern(5)
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2
1
```

Problem 4 redux: Write a recursive function **makePattern**() that **returns** a string consisting of the same numbers as are printed in pattern().

```
>>> s = makePattern(1)
>>> s
'1'
>>> s = makePattern(2)
>>> s
'1 2 1'
>>> s = makePattern(3)
>>> s
'1 2 1 3 1 2 1'
>>> s = makePattern(4)
>>> s
'1 2 1 3 1 2 1 4 1 2 1 3 1 2 1'
>>>
```

See the solutions in the file **exercises.py**.

The factorial function

For a given non-negative integer n, n! (read “n factorial”) is defined as the product of all the positive integers up to n: n (n-1) (n-2) (n-3) … 1.

It’s easy to write a loop that solves this. What about a recursive definition? How can we write n! in terms of a smaller factorial?

Exercise: Find a recursive definition and write it in Python.

How many calls to factorial are made for a parameter n?

One for n
One for n-1
One for n-2
…
One for 1 → A total of n calls

The exponent function

a to the power n is just a * a * a *… * a, i.e. a multiplied by itself n times.

This can be easily implemented iteratively using a loop:

```
def loop(a, n):
      ans = 1
      for i in range(n):
            ans = ans * a
      return ans
```

How many multiplications are done?

Using recursion and the addition rule for exponents we can reduce the number of multiplications.

What is **the addition rule for exponents**?

$a^{2n} = a^n * a^n = a^{n+n}$

$a^{2n+1} = a^n * a^n * a = a^{n + n + 1}$

See the solution in **exponent.py**.

How many multiplications are done? How does it compare with the iterative definition?

List printing, redux

What if we want to print a list that contains sublists? How can we modify the solution we developed to work in that case?

We need to add several bases cases to handle various cases for the first element.

What values can the first element be? Which of those values is relevant for making recursive calls?

Exercise: Modify the solution produced previously to work on multi-dimensional lists. It should work as follows:

```
>>> listPrint([1, 2, 3, 4])
1
2
3
4
>>> listPrint([[[1, [2], [3], [[[4]]]], [5], [[6, 7],
8]], 9])
1
```

```
2
3
4
5
6
7
8
9
>>> listPrint([])
>>>
```

List processing

Functions that process arbitrarily nested lists can also return values.

Exercise 1: Write a function that returns the number of integers in an arbitrarily nested list. It should work as follows:

```
>>> val = countInts([])
>>> val
0
>>> val = countInts([1, 2, 3.5, 'test', 9.1, 10])
>>> val
3
>>> val = countInts([3.5, [[[[1]]], 2], 'five', [[[3,
(4, 6)]]]])
>>> val
3
>>> val = countInts([[1, 2], [3, 4], [[[[[[[5]]]]]]],
{6, 7}])
>>> val
5
>>>
```

What are the relevant cases? What could the first element be? What do we do in each case?

Exercise 2: Write a function that returns the largest in an arbitrarily nested list that contains only non-negative numbers. It should work as follows:

```
>>> val = findMax([])
>>> val
0
>>> val = findMax([1, 2, 3.5, 100, 9.1, 10])
>>> val
100
>>> val = findMax([3.5, [[[[10]]], 2], 5, [[[3, (4,
6)]]]])
>>> val
10
>>> val = findMax([[1, 20], [3.5, 40],
[[[[[[[5]]]]]]]])
>>> val
40
>>>
```

What are the relevant cases? What could the first element be? What do we do in each case?