CSC148 *Intro. to Computer Science*

Lecture 6: *Recursion*

Amir H. Chinaei, Summer 2016

Office Hours: R 10-12 BA4222

ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/

Course page:
http://www.cs.toronto.edu/~ahchinaei/teaching/20165/csc148/

Test #1 Average

- 66%
- Sample solution: available in the course page
- Remark requests are accepted until June 20
- Some of you may not have had the best day
  - 50% vs 150%

Test #2 Preparation

- Carefully reading previous terms solution?
- Carefully reading other problems solutions?
- Watching tutorials, videos, online lessons?
- Nothing helps as much as

getting involved in solving problems prior to see their solution

- Take most advantage of **Peer Instructions**
  - Its optional
  - No re-mark option
  - Provides bonus points, and most importantly opportunity to grasp

Review

- Last lectures
  - Linked lists
  - Wrappers and helpers
- Today
  - Quick review of linked lists
  - Introduction to recursion

- Recall
  - Utilize office hours, forum, **CS help centre**
    - *in addition to lectures and labs*

Example 1: sum of a list

```
>>> L1 = [1, 9, 8, 15]
>>> sum(L1)
?
>>> L2 = [[1, 5], [9, 8], [1, 2, 3, 4]]
>>> sum(L2)
?
>>> sum([sum(row) for row in L2])
?
>>> L3 = [[1, 5], 9, [8, [1, 2], 3, 4]]
```

How can we sum L3?

In general, how can we sum any list?

sum_list()

```
def sum_list(L):
    ''' (list or int) -> int
    Return L if it's an int, or sum of the numbers
    in possibly nested list L
    >>> sum_list(17)
    17
    >>> sum_list([1, 2, 3])
    6
    >>> sum_list([1, [2, 3, [4]], 5])
    15
    '''
    # reuse: isinstance, sum, sum_list !
    if isinstance(L, list):
        return sum([sum_list(x) for x in L])
    else: # L is an int
        return L
```

Tracing sum_list()

- To understand recursion, trace from simple to complex:
- Trace sum_list(17)

Tracing sum_list()

- To understand recursion, trace from simple to complex:
- Trace sum_list([1, 2, 3])
  - Remember how the built-in sum works

Tracing sum_list()

- To understand recursion, trace from simple to complex:
- Trace sum_list([1, [2, 3], 4, [2, 3]])
  - Immediately replace calls you've already traced (or traced something equivalent) by their value

Tracing sum_list()

- To understand recursion, trace from simple to complex:
- Trace sum_list([1, [2, [3 ,4], 5], 6, [2, 7, 5]])
  - Immediately replace calls you've already traced by their value.

Example 2: depth of a list

Define the depth of L as follows.

If L is a list, 1 plus the maximum depth of L's elements, otherwise 0.

Example 2: depth of a list

```
>>> L1 = [1, 9, 8, 15]
>>> depth(L1)
?
>>> L2 = [[1, 5], [9, 8], [1, 2, 3, 4]]
>>> depth(L2)
?
>>> depth (12)
?
>>> L3 = [[1, 5], 9, [8, [1, 2], 3, 4]]
```

How can we calculate depth of L3?

How can we calculate depth of any list?

depth()

```python
def depth(L):
    ''' (list or int) -> int
    Return 0 if it's empty or an int,
    otherwise 1 + max of L's elements
    >>> depth(17)
    0
    >>> depth([17])
    1
    >>> depth([1, [2, 3, [4]], 5])
    3
    '''
    # reuse: isinstance, max, depth !
    if isinstance(L, list):
        if len(L) == 0:
            return 0
        else:
            return 1 + max([depth(x) for x in L])
    else: # L is an int
        return 0
```

Tracing depth()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace depth([])

Tracing depth()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace depth(17)

Tracing depth()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace depth([3, 17, 1])

Tracing depth()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace depth([5, [3, 17, 1], [2, 4], 6])

Tracing depth()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace depth([14, 7, [5, [3, 17, 1], [2, 4], 6], 9])

Example 3: find **maximum** in nested list

- how would you find the max of non-nested list?

  `>>> max(...)`

- how would you build that list using a comprehension?

  `>>> max([...])`

- what should you do with list items that were themselves lists?

  `>>> max([max_list(x) ...])`

- get some intuition by tracing through at lists, lists nested one deep, then two deep. . .

max_list()

```
def max_list(L):

    …

    if isinstance(L, list):
      return max([max_list(x) for x in L])
    else: # L is an int
        return L
```

Tracing max_list()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace max_list([3, 5, 1, 3, 4, 7])

Tracing max_list()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace max_list([4, 2, [3, 5, 1, 3, 4, 7], 8])

Tracing max_list()

- Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced
- Trace max_list([6, [4, 2, [3, 5, 1, 3, 4, 7], 8], 5])

Example 4: get some turtles to draw

- Spawn some turtles, point them in different directions, get them to draw a little and then spawn again…

- Try out tree_burst.py from the course page

- Notice that tree_burst returns **NoneType**: we use it for its side-effect (drawing on a canvas) rather than returning some value.