Introduction to: Computers & Programming:
Recursion

Adam Meyers
New York University
Summary

• What is recursion?
• Some Simple Examples
• Some Rules about Recursion
• The equivalence between recursion and while loops
• Complex problems/puzzles that are easier to understand through recursion
What is Recursion?

• A recursive function is a function that calls itself to solve successively smaller versions of the same problem.

• Recursive functions typically divide problems into:
  – One or more base cases which have simple solutions.
  – All other cases, for which the function must call itself on smaller instances of the problem
  – Without a base case, recursion is endless (like while loops)

• Recursive functions reduce all complex problems to instances of the base case + additional stuff + a recursive step

• Any instance of recursion can be converted to iteration (loop-based solution).

• Some problems are more naturally implemented recursively and others are more naturally implemented as iteration.
Recursion vs. Iteration

• Examples from recursion-functions.py
  – Counting
  – Counting down
  – Factorial
  – The fibonacci Sequence
  – Endless timer

• For each of these, which solution is more intuitive?

• Notes about recursion:
  – It is provably equivalent to iteration (e.g., with `while`)
    • I will not do a formal proof, but the examples are suggestive
  – Not supported for large problems in some languages (e.g., Python)
  – Recursion attractive because it closely models logic of mathematical proofs by induction.
The Strategy for Solving a Problem with Recursion

• Given a big problem, find a problem that is exactly one step smaller.

• Assume that you can handle the smaller problem and figure out what more you would need to do to solve the whole problem.

• Identify the smallest problem of this type that has a very simple answer and does not require any further breaking down – this is the base case.

• If all problems of this type larger than the base case can be broken down this way, recursion should work.
Class Problem 1

• Write a recursive function for computing compound interest.
• The function should take three arguments:
  – The principle
  – Interest per interval
  – The number of intervals
• It should return the total after adding all the interest
• For example, 10% on 1000 compounded daily
  – Principle = 1000
  – Interest = .1 / 360
  – Intervals = 360
• The total should be about: 1105.16
  – Don't use the shortcut \(( (1 + \text{interest}) ^ {\text{intervals}})\)
Class Problem 2

• Given a list of random numbers, find the highest number

• Base case 1: If there are two numbers in the list, use $\geq$ to find the highest number

• Base case 2: If there is only one number in the list, that number is the highest number

• For all other cases:
  – Divide the list into 2 parts that are equal in size or one away from being equal
  – Find the highest value from each list:
  – Compare them (base case 2)
Recursive Definitions of Family Relations

• Defining family relations in terms of
  – `parent_of`
  – `child_of`
  – recursion

• I have encoded the British royal family tree from:
  as a text file:
  /Users/adam/Desktop/python/Class Scripts/british-royal.txt

• Each line is a list of names divided by spaces
  – The first two names indicate parents of the remaining names on the lines:
    • Prince Charles : Lady Diana : Prince William of Wales : Prince Henry of Wales

• Simplifications:
  – Ignore multiple marriages, divorces and marriages that do not yield children
  – Ignore gender (do not distinguish between mother/father sister/brother, etc.)
Creating Dictionaries

- **is_parent_of_table**
  - Key → name
  - Value → list of 2 parents

- **children_table**
  - Key → names of parents
  - Value → list of children

- **mate_table**
  - Key → name
  - Value → mate
Some Definitions Implemented as functions in recursive-functions.py

- parent, child, mate – defined by dictionary lookup
- siblings(person)
  - Remove person from the set: children_of(parents_of(person))
- ancestors(person) – recursively apply parent_of function (forever)
- descendents(person) – recursively apply child_of function (forever)
Levels of Ancestors/Descendants

• get_nth_ancestors(name, N)
  – gets one generation of ancestors
  – get_nth_ancestors(name, N) is the set of parents of the ancestors of get_nth_ancestors(name, N-1)
  – 1 – parents, 2 – grand parents, 3 – great grand parents, etc.

• get_nth_descendants(name, N)
  – gets one generation of descendants
  – get_nth_descendants(name, N) is the set of children of get_nth_descendants(name, N-1)
  – 1 – children, 2 – grand children, 3 – great grand children, etc.
Aunt/Uncles & Niece/Nephews

• Siblings of nth ancestors are nth aunt/uncles
  – 1 – aunt/uncle, 2 – grand aunt/uncle, 3 – great grand aunt/uncle, etc.

• Nth niece nephews are Nth descendants of siblings
  – 1 – niece/nephew, 2 – grand niece/nephew, 3 – great grand niece/nephew
Nth Cousins, K times Removed

- Cousins are descendants of Aunt/Uncles
- N refers to the minimum number of generations away the pair of cousins are from the pair of siblings from which they descended
- K refers to the difference in generations between the cousins
- Examples
  - 1st cousins, 0 removed – children of siblings
  - 2nd cousins, 0 removed – grandchildren of siblings
  - 2nd cousins, 1 time removed – grandchild of one sibling and great grandchild of the other
Calculating Nth Cousins, K times Removed

- Find Nth Aunt/Uncles of PERSON
  - Their N+K descendents will be PERSON's Nth cousins K times removed
- If K does not equal zero, Find the N+K aunt/uncle of PERSON,
  - Their K descendents will (also) be PERSON's Nth cousins K times removed
- This function has also been implemented
The Tower of Hanoi Problem

• French Mathematician Édouard Lucas invented this problem in 1883.
• The story is that there is a temple with three posts in front of it and 64 different sized disks, which the priests must move from the first post to the last post without ever placing a larger disk on a smaller one.
• The question that has interested mathematicians since is: given an N such disks what is the algorithm for moving the disks?
There is a Turtle Implementation

- tdemo_minimal_hanoi.py
- The key to the whole problem is in the function: \textit{hanoi}, with 4 arguments:
  - Number of disks
  - Start stack
  - Middle stack
  - End stack
- It uses the functions:
  - Pop (which removes the top item from a stack)
  - Push (which pushes an item onto a stack)
The hanoi function II

• The base case: if number of disks is 1:
  – recursive calls don't do anything (called with N-1=0)
  – Moves disk from 1\textsuperscript{st} stack to 3\textsuperscript{rd} stack

• Two recursive calls one before and one after movement of disk from 1\textsuperscript{st} stack to 3\textsuperscript{rd} stack
  – The first call moves n-1 items from stack 1 to stack 2.
  – The second call moves n-1 items from stack 2 to stack 3.

• Let's watch the turtle function a few times
  – Try to break down each movement of n items into smaller subordinate calls doing the same thing
Other Recursive Problems

• Fractiles: design based on recursive application of some design principles
  – Fractiles are designed to look (mostly) the same whether you look at a small portion of the piece of work or the whole thing.

• Other sequences (besides Fibonacci)
  – Pascale's Triangle
    http://daugerresearch.com/vault/parallelpascaltriangle.shtml
  – Triangle number/Factorials
Summary

• Recursion is an effective programming technique which closely reflects the way mathematical ideas are formalized.

• For finite recursion, it is crucial to have a way of exiting recursion (a base case), just as it is essential to make sure that while loops have a way of exiting the loop, unless the loop is supposed to be endless.

• There will be no explicit example on the final that requires you to use recursion.

• It is worthwhile to have a number of strategies (types of loops, recursion, etc.) for the final exam. We will not have homework in recursion, simply because there is very little time left in the semester.
Optional Homework – Will Not be Graded

1. Write a recursive program that calculates triangle numbers. A triangle number of a positive integer is the sum of all the numbers less than or equal to that number. For example, triangle(5) = 1 + 2 + 3 + 4 + 5 = 15; triangle(9) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45, etc.

2. Write a recursive program that counts the number of times an element appears in a list.
3. Write a recursive program that requests an answer to the question “Are we there yet?” using an input statement and terminates if the user answers 'yes', but executes a recursive call if the user answers anything else.

4. (Hard Example – would have been extra credit): Write a recursive function that sorts a list of numbers using the following strategy (this is an implementation of the Quicksort algorithm):

   ● If the list contains one item, return that list (it is sorted)
     ● This is one basic step
   ● Else If the list contains 2 items, compare the two items with < and return a list with the smaller item first
     ● This is also a basic step
   ● Else (this is the recursive step):
     ● Choose the first item in the list (the pivot) and create two smaller lists: one list of all the items larger than the first item and one list of all items smaller than that item
     ● (Recursively) sort the higher list
     ● (Recursively) sort the lower list
     ● Create a new list by adding the pivot to the end of the lower list (append) and then adding the higher list to the end as well (extend)