DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 242 NOTES
Fall 2024-2025: Anthony Zoko

Week of October 28th, 2024

Traversal using recursion

One of the most natural uses of recursion is to traverse computer directories and web pages.

We'll start locally by considering how to recursively traverse computer directories.

An anti-virus scanner

In the file **antiVirus.py** is the implementation of a simulation of an anti-virus scanner.

It stores the signatures of several (fake) viruses as a dictionary.

The name of the virus is the key and the signature of the virus is the value in each pair. (The names and signatures are fabricated).

The **high-level idea** of the program is as follows:

- Starting from the current directory (the folder in which antiVirus.py is stored), we **recursively visit all files and subdirectories** of the current directory.
- All the files discovered are **opened and checked** for the signature of the viruses in the dictionary. When a file with one of the signatures is found, a message is printed indicating which file was detected and where it was found.

The program uses several new Python features from the os module.

The **os module** contains functions that make available to the programmer operating systems features. To find what functions are available, type help(os) after importing the os module.

One of the functions the **antiVirus.py** program uses is os.listdir().

os.listdir() takes a string representing a folder pathname as a parameter and returns a list containing the names of entries in the folder. The names in the list are in arbitrary order.

Note: os.listdir() does not return a list of full pathnames, just a list of names of files in the folder described by the parameter.

Recursive calls on those files should be made on the files' pathnames, not the files' names.

The **pathname of the file/subfolder** is obtained by concentrating the folder with the name of the file or subfolder, for example on a Windows machine:

```
path = folder + “\” + file/subfolder
```

Explicitly doing the concatenation means that the program has to be changed if the type of machine on which it is being run changes. Macs and Linux/Unix use “/” not “\”.

A more portable solution is to use the **path.join()** method of the os module.

os.path.join(pathname, item) joins the two parameters to create a pathname that works on the machine on which the program is being executed.

Let's examine the **recursion in the anti-virus program** more carefully.

The **recursive call is on a subdirectory**.

If the item passed as a subdirectory is in fact a file, then the call to os.listdir() will throw an exception.

```
>>> os.listdir("test/directory.py")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    os.listdir("test/directory.py")
WindowsError: [Error 267] The directory name is
invalid: 'test/directory.py\\*.*'
```

If we put the call to os.listdir() into a try block, then we can put the base case in an associated except block.

In the **base case** we open the file, read its contents, and then check for all of the file signatures in our dictionary.

Modifications to the anti-virus scanner

There are several modifications we can make on the anti-virus scanner:

1. Check for hidden files and avoid them (see **antiVirusForMacs.py**)

Explicitly check for file and directories and do the right thing (see **antiVirusRedux.py**).

Fibonacci sequence

The **Fibonacci sequence** of integers is defined recursively as follows:

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

The sequence starts with 1, 1, 2, 3, 5, 8, 13, 21, 34, …

The obvious recursive algorithm can be found in **fib.py**.

The problem is that it can't compute even small(ish) Fibonacci numbers in a reasonable amount of time!

What happened?

There is a **huge duplication of effort** when the obvious recursive definition is implemented. It slows down the computation to point where large values become infeasible. Draw the picture!

Recursion is not always the right solution. See an iterative version that works efficiently in the file **fibonacci.py**.

Searching

In a searching problem you are looking for a specified value in a collection, like a list. You typically want to return the index of the item if it is present or -1 if it is not present.

See a simple first searching example in the file **linearSearch.py**.

Binary search

Binary search is used to search a sorted list in time that is logarithmic in the size of the list.

Binary search can be implemented very intuitively using recursion.

The recursive definition is as follows:

1. If the lower index is larger than the higher index, return -1.
2. Otherwise, if the element we want to find is less than the element at the midpoint, search in the first half of the list.
3. Otherwise, search in the second half of the list.

See the recursive version in the file **recBinarySearch.py**. Start with the **recBinSearchTemplate.py** file.

Now try writing an iterative version of binary search.

The solution is in the file **iterBinarySearch.py**.

Towers of Hanoi

As a last example of recursion we will consider the classic Towers of Hanoi problem. It is a problem that is much easier to understand recursively.

> **Problem**: Given three pegs (1, 2, and 3) and $n \geq 0$ disks of different diameters that sit on peg 1, with disks of increasing diameter stacked from top to bottom, determine a way to move the disks to peg 3.
>
> **Constraints**:
> - The disks can only be moved one at a time
> - A disk must be released before another disk is picked up
> - A disk cannot be placed on top of a smaller disk

See: http://www.algomation.com/algorithm/towers-hanoi-recursive-visualization for a visual representation of the problem

Goal: Develop a recursive function hanoi(n) that moves n disks from peg 1 to peg 3 using legal single-disk moves.

> **Base cases**:
>
> n = 0: No disks to move. Done!
>
> n = 1: Move the disk from peg 1 to peg 3 immediately. Done!
>
> n = 2: Move the top disk from peg 1 onto peg 2. Then move the lower disk from peg 1 onto peg 3. Finally move the disk from peg 2 onto peg 3.
>
> **Recursive case**:
>
> To move n disks from peg 1 to peg 3, move n-1 disks from peg 1 to peg 2, move the lowest disk from peg 1 to peg 3, and then move n-1 disks from peg 2 to peg 3.

Intermediate function: moveTower(n, source, dest, temp) – move n disks from source to dest using temp as temporary storage

> **Recursive case**:
>
> moveTower(n-1, peg1, peg2, peg3)
> moveTower(1, peg1, peg3, peg2)
> moveTower(n-1, peg2, peg3, peg1)
>
> **Base case**: moveTower(1, source, dest, irrelevant)

See the implementation in **hanoi.py**. It also prints directions to be used for the game.