Introduction to Computer Science and Programming for Astronomers

Lecture 3.

István Szapudi

Institute for Astronomy
University of Hawaii

January 29, 2014
Outline

1. Reminder
2. Basic Data Structures: Continued
3. Algorithmic Analysis
   - Fibonacci
Reminder
Where were we last time...

- We have looked at testing and debugging.
- Elements of object oriented design.
- We started to look at data structures
- The two basic ones we finished were stacks and queues, next continue with binary trees.
- FAQ: how are lists implemented? A: array of C-pointers in the reference CPython implementation. (There are other implementations, e.g. ironpython, pypy, etc, they all should work the same, we take the “black box” approach)
Binary Trees
This is the most basic and useful tree data structure

- Graphs are networks consisting of nodes connected by edges or arcs.
- Trees are graphs which have no loops.
- A simple linked list can be thought of as a graph or a tree (sometimes called “snake topology” in tree graphs language).
- Binary trees are trees with two branches.
- Root: the top of the tree. Leaves: tips with null references. (Sometimes parent/childre/siblings is used).
Binary Tree
This tree is upside down!

- Note the similarity of construction with linked lists.
- The "leafs" are marked with None.
- The binary nature would be easy to generalize.
class Tree:
    def __init__(self, cargo, left=None, right=None):
        self.cargo = cargo
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.cargo)

tree = Tree(1, Tree(2), Tree(3))
Traversing a tree
Simply take left and right

```python
def totalTree(tree):
    if tree == None: return 0
    return totalTree(tree.left) +
    totalTree(tree.right) +
    tree.cargo
```
Traversing a tree
Simply take left and right

Python 2.3.3 (#1, May 7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import lecture2
>>> Tree = lecture2.Tree
>>> tree = Tree(1,Tree(2),Tree(3))
>>> lecture2.totalTree(tree)
6
Traversing a tree
E.g. preorder

```python
def printTreePreorder(tree):
    if tree == None: return
    print tree.cargo,
    printTreePreorder(tree.left)
    printTreePreorder(tree.right)
```
An Expression Tree
A simple example of a common application.

The following represents $1 + 2 \times 3$
Traversing an expression tree
Preorder (prefix), postorder (postfix), inorder (infix)

```python
>>> tree = Tree('+', Tree(1),
           Tree('*', Tree(2), Tree(3)))
>>> lecture2.printTreePreorder(tree)
+ 1 * 2 3
>>> lecture2.printTreePostorder(tree)
1 2 3 * +
>>> lecture2.printTreeInorder(tree)
1 + 2 * 3
```
I only covered the three most basic data structures. Some of the important structures you might encounter are:

- General Graphs
- Ordered Lists and Sorted Lists
- Hashing, Hash Tables, and Scatter Tables (use Python dictionaries)
- General Trees/Search Trees
- Heaps and Priority Queues
- Sets, Multisets, and Partitions (Python 2.4 > has now direct support for sets)
Analysis of Algorithms

- The algorithm is the basic idea or outline behind a program.
- We express algorithms in Python in this class but you should be able to translate any algorithm, to code in any language via easy (and mechanical) steps (or you could write a cross compiler :-)
- We will talk about both the design and the analysis of algorithms
Why Analysing of Algorithms?

- Analysis is more reliable than experimentation.
- It helps choosing among different solutions to problems.
- Predict the performance of a program before coding (especially important for large projects).
- By analyzing an algorithm, we gain a better understanding of where the fast and slow parts are, and what to work on or work around in order to speed it up.
The Fibonacci Numbers
A classic problem to introduce algorithms.

Problem: calculate the Fibonacci numbers. They are defined by the following recursion:

\[ F_n = F_{n-1} + F_{n-2} \]  \hspace{1cm} (1)

with initial conditions \( F_0 = F_1 = 1 \).

- The original problem is a toy model for population dynamics.
- This is simple example is rich enough to demonstrate many aspects of algorithmic analysis.
The Fibonacci Numbers
Mathematical Analysis

- This is second order, homogenous, linear recursion.
- Linear recursions (difference equations) are a bit like linear differential equations. For this one, we can look for the solution in the form of $F_n = q^n$. This gives rise to the following characteristic equation:

\[ q^n = q^{n-1} + q^{n-2} \rightarrow (q \neq 0) \]
\[ q^2 = q + 1 \]  \hspace{1cm} (2)

- The solution: $q_\pm = \frac{1 \pm \sqrt{5}}{2}$ (golden ratio)
The Fibonacci Numbers

Initial Conditions

- Since the equation is linear, superposition of solutions is a solution.
- We need to use the initial conditions, (second order needs two condition), assuming $F_n = aq^n + bq^n$

\[
\begin{align*}
  a + b &= 1 \\
  aq + bq &= 1 
\end{align*}
\]  

(3)

- The solution:

\[
\begin{align*}
  a &= \frac{1 - q}{q + q} , \quad b = \frac{q - 1}{q + q} 
\end{align*}
\]  

(4)

- Note that you can solve all equations of the form $cF_n + dF_{n-1} + F_{n-2} = 0$ exactly the same way.
The Fibonacci Numbers
Let’s write a class!

class Fibonacci:

def __init__(self):
    self.qp = 0.5+math.sqrt(5.0)/2.0
    self.qm = 0.5-math.sqrt(5.0)/2.0
    dq = self.qp-self.qm
    self.a = (1.0-self.qm)/dq
    self.b = (self.qp-1.0)/dq
class Fibonacci:
    ...
    def fibQ(self, n):
        """arithmetic
        ""
        return self.a * self.qp**n + 
               self.b * self.qm**n
>>> import lecture3
>>> f = lecture3.Fibonacci()
>>> f.fibQ(100)
5.7314784401381897e+20
>>> f.fibNonRecS(100)  # we’ll see this later
573147844013817084101L

For many purposes probably...but what if want accurate, integer arithmetic?
An Archetypal Recursive Algorithm
Together with factorial...

def fibRec(self, n):
    """
    simple recursive version
    """
    if n == 0 or n == 1:
        return 1
    else:
        return self.fibRec(n-1) + self.fibRec(n-2)
Is this fast enough?
How do we compare the efficiency of algorithms?

- If $n > 1$ this algorithm executes 2 lines, and then itself twice, symbolically

$$t_n = 2 + t_{n-1} + t_{n-2}$$

- This can be visualized as a binary tree.
- This is almost the same equation (it is inhomogenous, to solve it we would need to find a particular solution, e.g. with the method of “varying the constants”).
- Even without solving it, it is clear that it will give exponential solution $t_n \approx const^n$. This can be very slow. (try to run it for $n = 40$).
Dynamic Programming
This is a useful paradigm to find efficient algorithms

- The problem with the above algorithm is that we keep calculating the same things over and over. E.g. to calculate \( n = 100 \), we have to calculate each \( n \) before, and for each we calculate \( n = 2 \), etc.

- Dynamic programming means breaking the program into smaller problems, calculating and storing the results.

- In this case this is achieved easiest by opening up the recursion.

- There are alternative definitions for “dynamic programming”, but the essence is the same.
An iterative algorithm
Uses loops instead of recursion

def fibNonRec(self, n):
    """
    non-recursive version
    """
    if n == 0 or n == 1:
        return 1
    else:
        f = (n+1)*[0]
        f[0] = f[1] = 1;
        for i in range(2, n+1):
            f[i] = f[i-1] + f[i-2]
        return f[n]
A More Pythonic Solution
Memoization with dictionaries

This is still dynamic programming, but without resolving the recursion.

```python
self.cacheDic = {0:1, 1:1} #__init__
...
def fibCacheDic(self,n):
    if self.cacheDic.has_key(n):
        return self.cacheDic[n]
    else:
        newFib = self.fibCacheDic(n-1) + self.fibCacheDic(n-2)
        self.cacheDic[n] = newFib
    return newFib
```
Comparison

- This is much faster: e.g. for $n = 45$ the (uncached) recursive algorithm takes over a billion steps.
- The iterative algorithm takes about $2n = 90$ steps. It is 10 million times faster.
- Notice that the difference is quite subtle between the two!
Space Complexity

- Speed is not everything: we can also compare algorithms on the basis of how much other resources they are using (e.g. memory, disk, etc.)
- If an algorithm takes too much memory you may not be able to run (even if it was theoretically faster than another algorithm)
- In the iterative algorithm we used an array of length $n$
- It is more complicated to analyse the space complexity of a recursive algorithm: at any one time we have to take into account all the active calls. This form a path from the active call to the leaves of the tree. This is at most $n$.
- The second algorithm can be modified to use less space.
def fibNonRecS(self, n):
    """
    non-recursive version, space saving
    """
    if n == 0 or n == 1:
        return 1
    else:
        a = b = 1; i = 2
        while i < n+1:
            c = a + b
            a = b
            b = c
            i = i+1
        return b;
The O(Big O) notation

- Def: \( f(n) = O(g(n)) \) iff there exists a constant \( c \) s.t. \( f(n) \leq c \cdot g(n) \).
- This notation does not care about fine details, but allows us to compare two algorithms easily.
- E.g. two algorithms with \( 2n \) and \( 10n \) respectively are equivalent according to this.
- \( 2n^2 \) and \( 10n \) are not. The idea is that the algorithm with faster asymptotics will win for large \( n \) anyway.
- Other \( \Omega \): lower bound, \( \theta \), both \( O \) and \( \Omega \), (little) \( o \): \( O \) but not \( \Omega \). In practice, we use \( O \) is if it was \( \theta \).
A Matrix Trick

One can check with direct calculation and prove with induction that the following identity is true:

\[
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^n = 
\begin{pmatrix}
F_n & F_{n-1} \\
F_{n-1} & F_{n-2}
\end{pmatrix}
\]

We can use this to construct an algorithm, which calculates Fibonacci numbers through matrix multiplication (but what good does that do?)
Recursive Powering

- For calculating the Fibonacci numbers with the above trick, we need to calculate $M^n$, where $M$ is a two-by-two matrix.
- If we had $A = M^{n/2}$, we could just do it one multiplication: $M^n = A^2$.
- Use this to calculate the powers in $O(\log(n))$ time.
- Note: $\log$ is always base-2 unless otherwise noted!
The Basic Idea

```python
def fibMat(self, n):
    if n == 0 or n == 1:
        return 1
    else:
        self.m = [[1,1],[1,0]]
        self.matRecPow(n)
        return self.m[0][0]
```

735 Computational Astronomy
The Heart of the Algorithm

def matRecPow(self, n):
    if n > 1:
        self.matRecPow(n/2)
    self.m = self.matSqr2x2(self.m)
    if n % 2 is 1:  # n odd
        self.m = self.matMul2x2(self.m, [[1, 1], [1, 0]])

735 Computational Astronomy
def matMul2x2(self, m1, m2):
    """
    multiply two 2x2 matrices
    """
    return [[m1[0][0]*m2[0][0]+m1[0][1]*m2[1][0],
             m1[0][0]*m2[0][1]+m1[0][1]*m2[1][1]],
            [m1[1][0]*m2[0][0]+m1[1][1]*m2[1][0],
             m1[1][0]*m2[0][1]+m1[1][1]*m2[1][1]]]
def matSqr2x2(self, m):
    """
    square of a 2x2 matrix
    """
    return self.matMul2x2(m, m)
Summary of Fibonacci Algorithms

- The arithmetic algorithm is $O(1)$, but uses floating point.
- The recursive one the most natural algorithm, however it has exponential scaling (the only really slow one)
- The dynamic algorithms are $O(n)$ (linear). The second version is $O(1)$ after a number higher then $n$ is called. We had a variation of the first version which had $O(1)$ space complexity.
- The matrix algorithm is $O(\log(n))$. In principle this would be the fastest integer algorithm for very large $n$. (However, can we store the results, say, for $n = 1000000$?)
Summary

- We finished basic data structures
- Started to study analysis of algorithms
- Demonstrated analysis through an assortment of Fibonacci algorithms
Homework # 3

- Reading: read the rest of the python tutorial at http://http://docs.python.org/2/tutorial/.

- (E1) A full node in a binary tree is a node with two non-empty subtrees. Let \( l \) be the number of leaf nodes in a binary tree. Show that the number of full nodes is \( l-1 \). (hint: induction)

- (E2) Write a program which will create a lookup table for hierarchical tree-indexing of a 2-dimensional \( 2^n \times 2^n \) array. Hierarchical indices have the advantage that two pixels which are close physically also have indices which are close. This can be thought of as a list representation of a (quad)-tree. Optionally, make your program work in arbitrary dimensions.
The recursive definition of the hierarchical index for two-dimensions is the following: The first two bits of the index tells you if the pixel is in the upper/lower or left/right portion of the full array. Once we know these bits, we know the pixel is in one of four quadrants of size $2^{n-1} \times 2^{n-1}$. Then the next two bits carry the same information with respect to this smaller rectangle, etc. untill we arrive at the lowest level. This needs a bit of digesting; also look at the figure http://healpix.jpl.nasa.gov/html/intronode3.htm
(E3) Measure the speed of the algorithms given in lecture3.py, verify the scalings. Plot your measured times on a figure, and fit a formula for each programs running time. Comment on the results.

(E4) Solve the recursions $y_n + y_{n-1} \pm 2y_{n-2} = 2^{-n}$, with initial conditions $y_0 = y_1 = 1$. Derive analytic solutions, and devise an algorithm which calculates the recursion. Compare the two solutions on a figure. (hint: solve the homogenous equation first, and find a particular solution).