CSC148H Lecture 6

Dan Zingaro
OISE/UT

October 15, 2008

Iterating Through Python Objects

```
>>> l = [1,2,3,4]
>>> for elt in l:
...     print elt
...
1
2
3
4
>>>
>>> d = {"Monday":"CSC148", "Wednesday":"CSC148"}
>>> for elt in d:
...     print elt
...
Wednesday
Monday
```

Iterating Through Our Objects?

- So far, we cannot iterate over the elements of our binary search trees
- This is unfortunate, because our BST's are just like Python maps!

```
>>> from bst import BinarySearchTree
>>> bst = BinarySearchTree ()
>>> bst.put (1, 'val1')
>>> bst.put (2, 'val2')
>>> for elt in bst:
...   print elt
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iteration over non-sequence
```

Iterators

- The problem is our binary search trees don't implement the iterator protocol
- An iterator is an object that allows you to iterate through the elements of a container
- Iterator objects define a **next** method for retrieving the next element of the container
- Whenever you iterate through the elements of a container, a lot of the time you are implicitly using an Iterator object.

Iterators...

- For a list `l`, when we say `for elt in l:`:
  - First, Python calls `iter(l)` to retrieve an iterator object
  - On each iteration, `next()` is called on the iterator object and its result is assigned to `elt`
  - When there are no more elements in the container, a `StopIterationException` is raised by the implicit call of `next()`
- For objects of user-defined classes, all we have to do is define a `__iter__(self)` method that returns an object that has a `next()` method

Iterator Example

```
''' example of an iterator '''

class MyClass:
    def __init__(self):
        self.data = [1,2,3]

    def __iter__(self):
        return MyClassIterator(self.data[:])


class MyClassIterator:
    ''' an iterator that iterates through its elements in reverse '''
    def __init__(self, data):
        self.data = data
        self.pos = len(self.data)

    def next(self):
        if self.pos > 0:
            self.pos = self.pos-1
            return self.data[self.pos]
        else:
            raise StopIteration
```

Generators

- When normal functions "return" their execution is complete
- A generator is a function that returns from its call by using a yield statement instead of a return statement
- The difference: generator functions can resume their execution following a yield
- Whenever python sees a function that uses a yield statement, it returns a generator object'
- This object is just an iterator (i.e., it has the method `next()` defined).
- Thus a generator can be used in any context where an iterator is required (such as in a loop).

Generators...

- The benefit of using a generator function (instead of defining your own Iterator) is that the current position in your container can be maintained implicitly by the state of the generator.
- We'll define the `__iter__` method in the BST as a generator function (like listing 5.29 in your text).

Generator Example

The iterator for a **TreeNode**:

```
def __iter__(self):
  if self.left:
    for elt in self.left:
      yield elt
  yield (self.key, self.val)
  if self.right:
    for elt in self.right:
      yield elt
```

Algorithm Analysis

- Algorithm analysis is about determining the computing resources required by an algorithm
- Since there's often more than one way to solve a problem, evaluating the computing resources required by an algorithm allows us to determine its efficiency compared to other algorithms
- Computing resources typically refers to the execution "time" an algorithm requires, but sometimes may also refer to the amount of memory an algorithm requires.
- We'll go through several approaches for solving the same problem, and compare them

Largest Segment Sum

- Input: Python list s of n numbers (positive or negative)
- Output: maximum sum found in any segment (contiguous portion) of the input
- The maximum segment sum of a list of all positive numbers is the list itself, so it's only interesting when we have some negative numbers
- If all numbers are negative, we will say the maximum sum is 0
- It's not obvious which negative numbers we should include, and which ones we should exclude
- Example
  - List: $[4, -3, 9, -5]$
  - What's the largest segment sum?

Solution (A)ful

Try every possible segment:

```
def maxSeg (s):
  maxSoFar = 0
  for l in range (len(s)):
    for u in range (l, len(s)):
      sum = 0
      for i in range(l, u+1):
        sum = sum + s[i]
      maxSoFar = max(maxSoFar, sum)
  return maxSoFar
```

Improving the Solution

- The above (awful) solution is really awful
- For example, when it sums the elements between bounds l and u, it will repeat all of this work when finding the sum between bounds l and $u + 1$
- Let's figure out how we can do better ...

Solution (B)ad

Instead of directly computing the sum of elements between `l` and `u`, we can just add `s[u]` to the sum we obtained from the segment `s[l..u-1]`

```
def maxSeg (s):
  maxSoFar = 0
  for l in range (len(s)):
    sum = 0
    for u in range (l, len(s)):
      sum = sum + s[u]
      maxSoFar = max(maxSoFar, sum)
  return maxSoFar
```

Solution (C)ool

To find the maximum over all segments in the array `s[0..i]`, we can find the maximum segment in `s[0..i-1]` and the maximum segment ending at `s[i]`, and compare them

```
def maxSeg (s):
  maxSoFar = 0
  maxEndingHere = 0
  for i in range (len(s)):
    maxEndingHere = max(maxEndingHere+s[i], 0)
    maxSoFar = max(maxSoFar, maxEndingHere)
  return maxSoFar
```

Timing the Algorithms

Size	**Awful**	**Bad**	**Cool**
200	0.46	0.03	0.0
300	1.47	0.04	0.0
400	3.49	0.08	0.0
500	6.76	0.13	0.0
600	11.53	0.20	0.0
700	18.06	0.26	0.0
800	27.24	0.35	0.0
900	38.11	0.43	0.0
100000	YAWN	yawn	0.17

Algorithm Analysis

- The above timing tells us something about which algorithm is the fastest
- We can't rely on the wallclock execution time, though, because
  - The time required for a program to execute may vary from computer to computer. (A program will probably be a lot slower on a PC from the 90's than a brand-new PC that has a wicked-cool processor.)
  - A "fast algorithm" on a slow computer may be slower than a "slow algorithm" on a fast computer on certain inputs

Algorithm Analysis...

- Our way of characterizing the time efficiency of an algorithm should:
  - be independent of the machine where it may execute
  - be able to distinguish the big differences between algorithms and not concern itself so much with "little differences"

Algorithm Analysis...

- A step is a basic unit of computation that can be done in a fixed amount of time by a computer.
- We want to determine the number of steps an algorithm takes as a function of its input size.
- How we define input size depends a lot on the problem.
- For the segment-sum problem, the input size is n (the size of the list)
- Typically the input size is the number of elements in the input
- For a given algorithm, we'll use the function $T(n)$ to denote the number of steps the algorithm takes on input size n

Analyzing Awful

- Question: how many times is `sum = sum + s[i]` executed? (This is the same as asking for the number of iterations of the inner loop)
- The outer loop executes n times
- The middle loop is executed at most n times for each iteration of the outer loop
- So, the middle loop executes at most n^2 times
- The inner loop is executed at most n times for each iteration of the middle loop
- So, the inner loop executes at most n^3 times
- Similarly, `sum = 0` and `maxSoFar = ...` are executed at most n^2 times
- Conclusion: we can say that an upper bound on the number of steps we execute is $n^3 + 2n^2$

Analyzing Awful...

- Observation: as n increases, the n^3 term in $n^3 + 2n^2$ comes to dominate, and $2n^2$ doesn't contribute much to $T(n)$

n	n^3	$2n^2$
10	1000	200
20	8000	800
30	27000	1800
40	64000	3200
50	125000	5000
100	1000000	20000
200	8000000	80000
300	27000000	180000
400	64000000	320000
500	125000000	500000
1000	1000000000	2000000

Big Oh

- Even if we had $T(n) = n^3 + 4n^2$, or $T(n) = n^3 + 50n^2$, when n becomes large enough, the n^2 term is peanuts compared to the n^3 term
- To measure the efficiency of an algorithm, we focus only on the approximate number of steps it takes
- We are not concerned with deriving an exact value for $T(n)$, so we ignore its constant factors and nondominant terms
- Big Oh notation makes this idea precise

Big Oh...

- Say we have an algorithm whose running time on input of size n is $f(n)$
- Three requirements:
  - We want to bound $f(n)$ from above by $g(n)$
  - We want $g(n)$ to be a reasonable estimate; that is, it only "overestimates" by a constant c
  - We only require $g(n)$ to be such an estimate for sufficiently large values of n (since we don't care about small instances)
- This all amounts to requiring that $f(n) <= cg(n)$ for all $n >= n_0$ and positive constant c
- We then say that $f(n) = O(g(n))$

Properties of Big Oh

- Constant factors disappear
- Example: $6n$ and $n/2$ are $O(n)$
- Lower-order terms disappear
- Example: $n^5 + n^3 + 6n^2$ is $O(n^5)$
- We can often just look at the loop structure of a program to determine its growth rate
- e.g. Bad is $O(n^2)$ because of the two nested loops (dependent on $n!$), and Cool is $O(n)$

Big Oh Proof

- TPS: prove that Awful is $O(n^3)$
- You have to show that $n^3 + 2n^2 <= cn^3$ for all $n >= n_0$ by finding positive constants c and n_0 that satisfy the claim

Big Oh Proof

- TPS: prove that Awful is $O(n^3)$
- You have to show that $n^3 + 2n^2 <= cn^3$ for all $n >= n_0$ by finding positive constants c and n_0 that satisfy the claim
- Dividing by n^3, we get $1 + \frac{2}{n} <= c$
- We can make this true (and complete the proof) if we set $c = 3$ and $n_0 = 1$

Big Oh Approximations

- TPS: is Awful $O(n^6)$?

Big Oh Approximations

- TPS: is Awful $O(n^6)$?
- Big oh gives us an upper bound on the time our algorithm takes to execute
- It gives no guarantee that the bound is "close" to what actually happens
- It's equally valid to say that Awful is $O(n^3)$, $O(n^6)$, $O(2^n)$, etc.
- However, saying that Awful is $O(n^3)$ gives us the most useful information
- $O(n^3)$ is a "tight bound": there is no smaller function q for which Awful is still $O(q)$

Exponential Cliff

- An important divide is the one between polynomial algorithms and exponential algorithms
- If there is no polynomial-time algorithm to solve a problem, we can solve it only for very small instances
- Even if computer speeds keep doubling, exponential algorithms explode too quickly with problem size to be feasible
- Problems that take more than $O(k^n)$ for $k > 1$ are considered intractable to solve

What is the Time Efficiency? (1)

```
def bigoh1(n):
    sum = 0
    for i in range(100, n):
        sum = sum+1

    print sum
```

What is the Time Efficiency? (2)

```
def bigoh2(n):
    sum = 0
    for i in range(1, n/2):
        sum = sum + 1
    for j in range(1, n*n):
        sum = sum + 1
    print sum
```

What is the Time Efficiency? (3)

```
def bigoh3(n):
    sum = 0
    if n%2 == 0:
        for j in range(1, n*n):
            sum = sum + 1
    else:
        for k in range(5, n+1):
            sum = sum + k

    print sum
```

What is the Time Efficiency? (4)

```
def bigoh4(m, n):
    sum = 0
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            sum = sum + 1
    print sum
```