CSC148

Lecture 6

Iterators and Generators

Algorithm Analysis
Iterators

- Iteration is not a new concept
- We've talked frequently about iterating through the elements of a container (such as a list)
- We also know the difference between an iterative algorithm and a recursive algorithm.
- So what exactly is an Iterator?
Iterators

• An Iterator is an object that allows you to iterate through the elements in a container
  – In particular, an iterator object defines the `next()` method, which returns the next element in the container.

• Whenever you iterate through the elements of a container, a lot of the time you are implicitly using an Iterator object.
Iterators

- Consider the following code fragment:

```python
for elt in L:
    # do something with elt
```

- Under the covers:
  - Python calls `iter(L)` to retrieve the iterator object
  - Each iteration of the loop calls the `next()` method on the iterator and assigns it to the loop variable `elt`
  - When there are no more elements in the container, a call to `next()` raises the `StopIteration` exception.
Iterators

- Python can iterate over instances of user-defined classes if you define a special method
- Simply define the `__iter__(self)` method in your class, and have it return an object that defines the `next()` method
Iterators

- Examples
Generators

- A generator is a function that returns from its call by using a `yield` statement.
- 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).
Algorithm Analysis

• Algorithm analysis is about determining the computing resources required by an algorithm.

• 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.

• “Time” isn't wall-clock time.
Algorithm Analysis

• We can't just use the wall-clock execution time for the following reasons:
  – The time required for a program to execute may vary from computer to computer. (A program will probably be a lot slower on PC from the 90's than a brand-new PC that has a multicore 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

• How can we do this?
Algorithm Analysis

- Let's try answering this by way of an example:
- We want to find the sum of $N$ integers from 1 to $N$. That is, $1+2+3+\ldots+N$. 
Algorithm Analysis

# one way of solving it

sum = 0
for i in range (1,N+1):
    sum = sum + i

# another way of solving it

sum = n*(n+1)/2
Algorithm Analysis

- Observations:
  - The first way will take longer for larger \( N \) than smaller \( N \).
  - Moreover, the first way will always take longer than the second way (except possibly for very small \( N \)).
Algorithm Analysis

• We can make this more precise by looking at the number of “steps” performed by each algorithm.

• We need to define exactly what we mean by a “step”.
  – A step is a basic unit of computation and can be done in a fixed amount of time by a computer.
Algorithm Analysis

- 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 summation problem, the input size is N.
- Typically the input size is the number of elements in the input. For example, the input size can be the number of elements in a list that is to be sorted.
Algorithm Analysis

- For a given algorithm, we'll use the function $T(n)$ to denote the number of steps the algorithm takes on input size $n$.
- $T(n) = n + 1$ for the first solution, and $T(n) = 1$ for the second solution.
- But what if we modified the second solution to look like it does on the following slide?
Algorithm Analysis

• Now $T(n) = 3$.

• Does this really make a difference in how efficient the algorithm is? Not really.

• The first algorithm (where we loop through all integers from 1 to $N$) takes *approximately* $N$ steps, and the second algorithm takes *approximately* 1 step.

```plaintext
sum = N+1
sum = sum * N
sum = sum / 2
```
Algorithm Analysis

- Suppose $T(n) = n^2 + 5n + 100$ for some algorithm $A$.
- As the input size increases (i.e., as $n$ increases), the $n^2$ term is going to dominate the expression. (That is, $5n + 100$ doesn't really contribute much to the overall value of $T(n)$.)
Algorithm Analysis

- Suppose $T'(n) = 2n^2 + 10n + 200$ for some algorithm $A'$

- That is, $T'(n) = 2T(n)$ (where $T(n)$ is from the previous slide)

- We can interpret this as follows: For every step taken by algorithm A, algorithm $A'$ takes 1 additional step.

- Is the efficiency of algorithms A and $A'$ really all that different? No.
Algorithm Analysis

- To measure the efficiency of an algorithm, we only care about approximately how many steps it takes.
- We don't care about deriving an exact value for $T(n)$ – its constant factors and non-dominant terms can be ignored.
- How can we make this idea more precise?
- Big-Oh notation!!!
Big-Oh Notation

• We say that a function $f(n)$ is $O(g(n))$ if there exists positive constants $c$ and $B$ such that
  - $f(n) \leq c \cdot g(n)$ for all $n \geq B$

• Whenever you see “$f(n)$ is $O(g(n))$” this can be read as “$f$ has order $g$”, or “$f$ is big-oh of $g$”.

• To estimate $T(n)$, the number of steps an algorithm takes, it suffices to find a function $g(n)$ such that $T(n)$ is $O(g(n))$. 
Key properties of Big-Oh Notation

- Constant factors disappear
  - examples on board
- Low-order terms disappear
  - examples on board
Big-Oh Notation

- Big-Oh notation gives us a convenient way to approximate the number of steps an algorithm requires in the worst-case, ignoring constant factors and lower order terms.
- The two summation algorithms that we looked at before are $O(n)$ and $O(1)$, respectively.
Things to Keep in Mind

• Just because the number of steps an algorithm takes is $O(f(n))$ does not mean that the algorithm takes anywhere near $f(n)$ steps.
  – Technically, both summation algorithms we studied take $O(n^2)$ steps.
  – In future courses, you'll see other notations related to Big-Oh that allow you to deal more precisely with this issue.
Things to Keep in Mind

- Sometimes constant factors have practical significance.

- Sometimes even though a big-oh analysis may indicate one algorithm is more efficient than another, this may only occur after the input size is impractically large.
Things to Keep in Mind

- Technically $O(f(n))$ defines a set of functions
- see the big-oh hierarchy drawn on board.
Examples

- Examples of big-oh notation
- Examples of analyzing the time efficiency of algorithms