Competition Programming and Problem Solving using (of course) Python

Dhruv Baldawa

About Me

- B.E. Computer Science from University of Mumbai
- GSoCer 2011 and 2012
- National Cyber Olympiad Gold medallist 2007
- Currently working at Enthought, Inc.

Why is solving difficult ?

- the dataset is too large to be iterate even once
- the complexity is huge or very bad
- or BOTH !!

What this talk covers

- Memory Management
  - Common problems
  - List forms v/s Iterator forms
  - itertools
- Time Management
  - Choosing the correct data structures
  - Approach
- Profiling tips

Memory Management

Why is it important ?

- Most of the common competitions require you store and compute large amounts of data.
- Running out of memory limit is pretty common.
- There is “always” a way to improve memory consumption of your program.

Common Problems

- MemoryError[1]
  - when you run out of memory, but the situation can still be rescued (by deleting certain objects)
- OverflowError[1]
  - when the result of an arithmetic operation is too large to be represented

List forms v/s Iterator forms

- List forms
  - better to store and re-use results of computations
  - use if you want to perform list operations, where you need to store entire lists
  - consumes much more memory
  - only prefer when a real list cannot be avoided
  - all the elements are generated/initialized at once

List forms v/s Iterator forms

- Iterator forms
  - lazy and on-demand generation of values
  - very low memory consumption
  - very useful when you only need to work with the value at hand
  - **Cons:**
    - cannot step backwards
    - cannot skip or jump forwards
  - they are scalable and memory-friendly and used where real lists are not required

List forms v/s Iterator forms

- range v/s xrange[2]
- list comprehensions v/s generator expressions
- functions v/s generators

import itertools

- itertools module provides a set of fast, memory efficient iterators
- provides fast implementations for common jobs like product, permutations, combinations
- itertools documentation

import itertools

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, ...	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, ... endlessly or up to n times	`repeat(10, 3) --> 10 10 10`
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`groupby()`	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
`ifilter()`	pred, seq	elements of seq where pred(elem) is True	`ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9`
`ifilterfalse()`	pred, seq	elements of seq where pred(elem) is False	`ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
`islice()`	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
`imap()`	func, p, q, ...	func(p0, q0), func(p1, q1), ...	`imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000`
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
`tee()`	it, n	it1, it2 , ... itn splits one iterator into n	

import itertools

`takewhile()`	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
`izip()`	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`izip('ABCD', 'xy') --> Ax By`
`izip_longest()`	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Iterator	Arguments	Results
`product()`	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

@dhruvbaldawa

Time Optimizations

- choosing the right data structure
- choosing the right approach

choosing the right data structure

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

--Linus Torvalds

Data Structures in Python

- don't re-invent the wheel. Use tuple, list, dict, sets, as they are coded in C and hence are FAST
- for membership tests use dict/set [O(1)] instead of lists/tuple [O(n)]
- use collections
- Queue operations like pop(), insert() are better in collections.deque [O(1)] than lists [O(n)]
- use bisect, heapq for sorted lists

import collections[4]

- deque
- Counter
- OrderedDict
- defaultdict

Memoization[5]

- caching results from previous procedure calls, and using it directly
- if a function call returns the same value when the same set of arguments are passed, then it can be memoized !

```
def memoized_function(value):
    if value in cache:
        return cache[value]
    else:
        cache[value] = compute(value)
        return cache[value]
```

Collatz Conjecture[6]

$f(n) = n/2$; if n is even,

$= 3n+1$; if n is odd

For example:

f(4) = 4 -> 2 -> 1

f(3) = 3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

Multiplication problem

- Multiply two numbers assuming there is no multiplication operator available

Profiling

- On Unix machines, you can simply:
  time python my_script.py
- python –m cProfile my_script.py
- On iPython, you can use:
  %timeit my_expensive_function()

References

1. http://docs.python.org/library/exceptions.html
2. http://justindailey.blogspot.in/2011/09/python-range-vs-xrange.html
3. http://en.wikipedia.org/wiki/List_comprehension
4. http://docs.python.org/library/collections.html
5. http://en.wikipedia.org/wiki/Memoization
6. http://en.wikipedia.org/wiki/Collatz_conjecture
7. http://wiki.python.org/moin/PythonSpeed/PerformanceTips
8. http://wiki.python.org/moin/PythonSpeed
9. http://www.udacity.com/wiki/CS215%20Unit%201%20Code?course=cs215