CSC148 winter 2015

big-oh, big ideas

week 12

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

April 1, 2015

Outline

big-Oh on paper

big-Oh examples

abstraction and big-oh

big data

$\mathcal{O}(n)$

The stakes are very high when two algorithms solve the same problem but scale so differently with the size of the problem (we'll call that n). We want to express this scaling in a way that:

- is simple
- ignores the differences between different hardware, other processes on computer
- ignores special behaviour for small n

big-O definition

Suppose the number of "steps" (operations that don't depend on n, the input size) can be expressed as $t(n)$. We say that $t \in \mathcal{O}(g)$ if:

> *there are positive constants c and B so that for every natural number n no smaller than B,*
> $t(n) \leq cg(n)$

use graphing software on:

$$t(n) = 7n^2 \qquad t(n) = n^2 + 396 \qquad t(n) = 3960n + 4000$$

to see that the constant c, and the slower-growing terms don't change the scaling behaviour as n gets large

if $t \in \mathcal{O}(n)$, then it's also the case that $t \in \mathcal{O}(n^2)$, and all larger bounds

$$\mathcal{O}(1) \subseteq \mathcal{O}(\lg(n)) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(2^n) \subseteq \mathcal{O}(n^n) \ldots$$

sequences

```
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print('n is: {}.'.format(n))

    if n > 97:
        print('big!')
    else:
        print('not so big!')
```

How does the running time of **silly** depend on **n**?

loops

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n):
    sum += i
```

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n//2):
    for j in range(n**2):
        sum += i * j
```

conditions

How does the running time of this code fragment depend on **n**?

```
sum = 0
if n % 2 == 0:
    for i in range(n*n):
        sum += 1
else:
    for i in range(5, n+3):
        sum += i
```

halving

How does the running time of **twoness** depend on **n**?

```
def twoness(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count
```

working with lg

$\lg(n)$: this is the number of times you can divide n in half before reaching 1.

- refresher: $a^b = c$ means $\log_a c = b$.
- this runtime behaviour often occurs when we "divide and conquer" a problem (e.g. binary search)
- we usually assume $\lg n$ (log base 2), but the difference is only a constant:

$$2^{\log_2 n} = n = 10^{\log_{10} n} n \implies \log_2 n = \log_2 10 \times \log_{10} n$$

- so we just say $\mathcal{O}(\lg n)$.

miscellaneous

How does the running time of this code fragment depend on **n**?

```
for k in range(5000):
    if L[k] % 2 == 0:
        even += 1
    else:
        odd += 1
```

more miscellaneous

How does the running time of this code fragment depend on **n** and **m**?

```
sum = 0
for i in range(n):
    for j in range(m):
        sum += (i + j)
```

summary

sequences:

loops:

conditions:

abstraction and performance

Weeks ago we discussed abstract data types (ADTs) as a technique to suppress implementation details while highlighting the data and behaviour. Do ADTs talk about efficiency?

It depends. So far our public interface has avoided performance guarantees. Our original implementation of **Queue** had $\mathcal{O}(n)$ performance for at least one of **enqueue** or **dequeue**.

A different implementation could guarantee $\mathcal{O}(1)$ performance for both operations.

abstraction and good models

A surprising number of problems can be thought of as graphs: **nodes** (entities) connected by **edges** (pairwise relationships):

- **courses** related by sharing **students** need an exam schedule without (or with few) conflicts
- **scientists** related by scholarly **citations** need a way to quantify scientific influence
- **locations** related by **roads** need an efficient route that visits them all at least cost

graph ADT

very powerful to use the same ADT, and often the same algorithms, on these seemingly unrelated domains

data: nodes (usually containing some value) and edges (trees are a special case), occasionally edge values (weights)

common operations: add or delete edges or nodes, determine "neighbours" of a node, determine value of a node

abstraction and parallelism

you have used list comprehension to **map** an operation over a list:

```
[x**2 for x in num_list]
```

you've also used built-in functions such as **sum** and **max** to **reduce** a list to a single value

```
sum([x**2 for x in num_list])
```

skillful combination of **map** and **reduce** are related to important techniques in distributing processing over many processors.

abstracting memoization

naive recursive algorithms suffer from redundancy, which may be solved via memoization

memoization is a standard enough pattern that it can be programmed, see fib.py

Notice! you may **not** use automated memoization in A3!