Optimisation and Operations Research

Lecture 8: Algorithm analysis and Big-O notation

Matthew Roughan
<matthew.roughan@adelaide.edu.au>
http://www.maths.adelaide.edu.au/matthew.roughan/notes/OORII/

School of Mathematical Sciences,
University of Adelaide

August 1, 2019

Section 1

Algorithm analysis

Algorithm Analysis

- We would like to estimate how long our program will take to run
- More generally, how many resources will it take?
  - time
  - memory (space)
  - unicorns
- And often, we would like to make it better

Empirical Measures

- Run the program and test it
  - see how long it takes
    - in MATLAB use `tic` and `toc`
    - there are lots of tools, *e.g.*, see *profilers*
  - see how much memory it uses
- This is simple, but
  - how do we *anticipate* performance for a new problem?
  - how do we determine the practical limits of our program?
  - how will our program run on a different computer?

Memory Analysis

- What takes up memory?
  - the program itself
    - this is usually fairly constrained, so we mostly ignore it
  - the variables
    - in MATLAB, most variables are `double precision floating point`
    - effectively they take *8 bytes* each
- Memory analysis is just a matter of counting the number of variables
  - a vector of length n, takes $8n$ bytes
  - an $n \times m$ matrix takes $8nm$ bytes
- In general, there are many other issues to consider, but simple counting is the starting point

Time Analysis

- Time analysis is more complicated
- What takes time?
  - each *operation* takes time, but they aren't all the same
    - $+$ might be faster than $\times$
    - $\times 2$ is often very fast in binary
  - the time an operation takes depends on the particular computer
    - so often we don't work out actual time, we look directly at the number of operations
- So once again it is just counting, but
  - there are different types of operations to count
  - there are some extra complexities we will consider below

Time Analysis: simple examples

calculation	operations +	×	notes
$(x_1 + x_2) \times x_3$	1	1	
$(x_1 \times x_3) + (x_2 \times x_3)$	1	2	same output as previous calc
x^6		5	assumes naïve multiplication
$A + B$	nm		for $n \times m$ matrices

It's just counting, but details matter!

Time Analysis: operations

There are lots of operations you could count

- *arithmetic:* *e.g.*, $\times$, $+$, $-$, $/$, ...
- *relational:* *e.g.*, comparisons $x > 0$, $y == 2$
- *logic:* `if true ...`
- *bitwise:* we don't use these much in MATLAB
- *set:* *e.g.*, union, intersection, ...
- *memory access:* *e.g.*, creating a variable (memory allocation), setting a variable, reading from an array, ...
- functions you call contain multiple operations: *e.g.*, $\sin(x)$
- in MATLAB vector operations are actually made up of lots of smaller operations, *e.g.*, $A + B$ would need to add all of the elements
- Input/Output (to screen or disk) – be aware this is *SLOW*

Time Analysis: operations

- There are lots of operations you need to consider
- We aim to break it down to *primitive operations*
  1. *e.g.*, arithmetic, relational, and logic
  2. Separate I/O from the algorithm
     - ⋆ make sure MATLAB lines end with a ';'
- Take *uniform-cost model*
  1. assume all primitive operations take the same time

Time Analysis: loops

```
for i=1:n
   % do some stuff that takes k operations
   ...
end
```

- The cost of a loop is the internal cost (assume k) multiplied by the number of times the loop runs (here n)
- So the cost here is nk operations

Time Analysis: loops example

```
for i=1:m
  for j=1:n
    x = x + (i*j)
  end
end
```

- The inner loop does 2 primitive ops
  - Note that in *this* example, it doesn't depend on the values of x, i or j
- The inner "j" loop performs this n times, so $2n$ ops
- The outer "i" loop repeates this m times, so the final count is

$$2nm$$

operations.

Strategy

- Break the program into blocks
  - we can just add up the cost of each block
- Look for loops
  - the cost of the "stuff" inside the block is *multiplied* by the number of times the loop runs
- Functions like $\sin(x)$ can often be given a constant cost
  - its hard to know exactly what it is
  - we'll fix that: see Big-O notation below

Example: pivot

```
function [Mout] = pivot(M, i, j, epsilon);
%
% pivot.m, (c) Matthew Roughan, 2015
%
% created:     Wed Jul 1 2015
% author:      Matthew Roughan
% email:       matthew.roughan@adelaide.edu.au
%
% Perform a pivot at position (i,j) of matrix M
%
% INPUTS:
%        M     = Tableau on which we operate
%        (i,j) = pivot location
%    optional inputs
%        epsilon = small number so that we don't test "exactly" zero, b
%
% OUTPUTS:
%        M_out
%
```

Example: pivot

```
if nargin < 4
  epsilon = 1.0e-12;
end

% check inputs
assert(i>=1 && i<=size(M,1) && i == round(i), 'invalid value of i');
assert(j>=1 && j<=size(M,2) && j == round(j), 'invalid value of j');
assert(abs(M(i,j)) > epsilon, 'M(i,j) close to zero');

if abs(M(i,j)) < epsilon
  error('M(i,j) close to zero');
end
```

Example: pivot

```
% create the output array
Mout = zeros(size(M));

% divide row i by M(i,j)
Mout(i,:) = M(i,:) / M(i,j);

% subtract enough of the new row from each other row to make other colu
for k=1:size(M,1)
  if k ~= i
    Mout(k,:) = M(k,:) - Mout(i,:)*M(k,j);
  end
end
```

Time Analysis: indeterminacy

The big problem for analysing cost is indeterminacy, *i.e.*, sometimes the code's behaviour changes depending on the values

```
if (x > 0)
  y = x + 1
else
  y = x + z + w
end
```

Often we don't know what value of x to expect (that's might be the whole point of the program) so how should we analyse this?

We'll look at this a little later.

Time Analysis: extra complexities

1. Modern computers really mess all this up
    1. CPU can perform multiple operations per clock cycle, under certain (complex) conditions
    2. Multiple levels of cache change speed to access (and hence operate) on variables
    3. ...
2. So what do we do?
    1. Big-O notation (see next section) abstracts away some details
    2. Complexity analysis looks at the *class* of the algorithm rather than the details, but we will look at this later

Section 2

Big-O Notation (and its friends)

Computational complexity

- Often, we don't care about the time for a particular problem, we care about the practical bounds for problems we might consider in the future
- We would like to estimate how long our program will take to run
  - as a function of the *size* of the problem
    - *e.g.*, n equals the number of variables
    - *e.g.*, m equals the number of constraints
  - could also include the size of the variables in memory
    - *e.g.*, k bit floating point numbers
- often interested in BIG problems, so look at asymptotic behaviour
  - *e.g.*, large m and n
  - use big-O notation

Big-O notation

Definition

$$f(\mathbf{x}) = O\big(g(\mathbf{x})\big)$$

means (i.e., iff) there exists constant c and $\mathbf{x}_0$ such that

$$|f(\mathbf{x})| \leq c|g(\mathbf{x})|$$

for all $\mathbf{x}$ such that $x_i \geq x_0$.

Usage:

- describes asymptotic limiting behaviour: implicit that $x \to \infty$
- the function $g(x)$ is chosen to be as simple as possible
- a common *mistake* is to think that it means $f(x)/g(x) \to k$

Big-O notation properties

- Multiplication: $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then

$$f_1 \times f_2 = O(g_1 \times g_2)$$

- Multiplication by a constant: $f = O(g)$

$$kf = O(g)$$

- Summation: $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then we can write a general expression, but usually either $g_1 = g_2$, or WLOG g_1 grows faster than g_2 and in these cases

$$f_1 + f_2 = O(g_1)$$

These properties mean that we can simplify using a simple set of rules

Big-O notation rules

When we use Big-O notation, we use the following rules:

1. if $f(x)$ is a sum drop everything except the term with the largest growth rate
2. if $f(x)$ is a product any constants are ignored

Assume these rules have been applied, when you see Big-O.

Example of RULE-1

Example

$f(x) = x^7 - 200x^4 + 10$ is dominated (for large x) by the x^7 term, so

$$f(x) = O(x^7)$$

We dropped the terms $-200x^4 + 10$ because they grow slower than x^7.

Example

We can reduce $O(n^2 + \log n)$ to $O(n^2)$.

The log() function grows more slowly than n (or any polynomial).

Example of RULE-2

Example

$f(n) = 3n^2$, which is a product, so we ignore constants, and

$$f(n) = O\left(n^2\right)$$

We ignored the constant 3.

Example

If k is a constant, we can rewrite $O(kn \log n)$ as $O(n \log n)$.

Whether k is a constant depends on the context.

Stirling's approximation

Stirling's approximation is both an example of use of the notation, and also a useful tool in some analysis:

$$\ln n! = n \ln n - n + O(\ln n)$$

We use Big-O notation here

We will use Big-O notation to count operations in an algorithm

Classic examples

problem	complexity	notes
$\sum_{i=1}^{n} x_i$	$O(n)$	
$A \times B$	$O(n^3)$	naïve algorithm
	$O(n^{2.373})$	clever algorithm
A^{-1}	$O(n^3)$	naïve algorithm
	$O(n^{2.373})$	clever algorithm
$det(A)$	$O(n!)$	naïve algorithm
	$O(n^3)$	clever algorithm

Where A and B are $n \times n$ matrices

Example of a more complicated function

Example

Calculate the complexity of computing $f(x) = \exp(x)$.

- This depends on how you compute $\exp(x)$.
- A simple approach is Taylor series
  - assume you want n digits of precision
  - that determines how many terms you need in the Taylor series
  - so computation is $O(nM(n))$, where $M(n)$ is the cost of a multiplication with n digits
- Assuming fixed precision (*e.g.*, in MATLAB, double precision)

$$\exp(x) = O(1)$$

  That is, its computational time doesn't depend on how big x is
- There are faster approaches, but this suffices for today
- Other elementary functions, *e.g.*, sin, cos, arctan, log, are similar

Nomenclature

In order, we describe classes of algorithms as ???-time (*e.g.*, constant-time)

complexity	name	example algorithms
$O(1)$	constant	calculate simple functions
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	adding arrays of length n
$O(n \log n)$	log linear	Fast Fourier Transform (FFT)
$O(n^2)$	quadratic	adding up all elements of a matrix
$O(n^d)$	polynomial	naïve matrix multiplication
$O(c^n)$	exponential	Simplex
$O(n!)$	factorial	brute force search for TSP

Weirdness

Example

$$x = O(x^2) \quad \text{but} \quad x^2 \neq O(x)$$

so using $=$ is slightly weird, as there is an asymmetry.
Sometimes we use $\in$ instead.
e.g.,

$$x \in O(x^2)$$

Often the symbols are used more generally

Sometimes we use these symbols in a type of algebra

Example

$$\left(n + O(n^{1/2})\right)\left(n + O(\log n)\right)^2 = n^3 + O(n^{5/2})$$

Meaning: for any functions which satisfy each $O(\ldots)$ on the LHS, there are some functions satisfying each $O(\ldots)$ on the RHS, such that substituting all these functions into the equation makes the two sides equal.

Variables

It can get confusing, as variables and constants sometimes are inferred from context.

For instance

$$\begin{aligned} f(n) &= O(n^m) \\ g(m) &= O(n^m) \end{aligned}$$

mean quite different things, even though the RHSs are the same.

Big-O limitations

Big-O has advantages:

- it gets to the nub of the question – what is the *shape* of the performance of our algorithm for large problems

However it has limitations

- it doesn't tell us about constants, and lower-order terms
  - these are important, particular for small to moderate sized problems
  - Big-O is only for asymptotic performance
- it doesn't tell us actual computation times
- *it's only an upper bound*

Big-Ω

Two forms of Big-Omega notation

- Hardy-Littlewood (used in math)
- Knuth (used in computational complexity)

Definition (Big Omega)

$$f(x) = \Omega\big(g(x)\big) \Leftrightarrow g(x) = O\big(f(x)\big)$$

More succinctly: $f(x) \geq kg(x)$ for some k

- Similar to Big-O, but gives a lower bound

Big Theta

Definition (Big Theta)

$$f(x) = \Theta\big(g(x)\big)$$

means that $f(\cdot)$ is bounded above and below by $g(\cdot)$, *i.e.*,

$$k_1 g(x) \leq f(x) \leq k_2 g(x)$$

for positive constants k_1 and k_2, for all $x > x_0$.

So Big-Θ notation means the function $f(x)$ grows as fast as $g(x)$.

Takeaways

- How to count (operations)
- Big-? notations
  - Big-O notation means the function grows no faster than
  - Big-Ω means the function grows faster than, and
  - Big-Θ notation means the function grows as fast as.

Used to provide asymptotic descriptions of algorithm performance

Further reading I