122COM: Performance and Scalability

David Croft

Coventry University

david.croft@coventry.ac.uk

2015

Overview

1 Profiling

2 *O*() notation

- Simple algorithms
- Good algorithms
- Bad algorithms

3 Recap

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
- Test the performance.
  - It may be fine.
- Profile your code to get the baseline performance.
  - So that you know if you are making things better or worse.
- Focus your efforts on the code that is consuming all the time.
  - E.g. small pieces of code that get called multiple times.

Statistical

- Regularly checks the system state.
- Will slow down running speed.
    - But equally throughout the code.

Instrumental

- bb

Profiler types

Flat Profiler

- a

Call Profiler

- b

O() notation

Profiling is very useful in determining the actual performance of your code.

- Not so good at measuring how code will scale.
- Algorithmic complexity.
- Certain algorithms are known to be better than other algorithms.

$O()$ notation

Used to describe complexity in terms of time and/or space.

- Commonly encountered examples...
  - $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$ and $O(n!)$
- n refers to the number of values.
  - E.g. n values to be sorted.
  - E.g. n values to be searched.

$O(n)$

Linear time.

- n is directly related to time/space required.
- E.g. linear search.

$O()$ notation describes the worst case scenario.

- Usually otherwise stated.

	0	1	2	3	4	5	6	7
`array[n]` =	A	B	C	D	E	F	G	H

$O(1)$

Constant time.

- *n* doesn't matter.
- Always takes same time/space.
- E.g. getting first item in an array.

$O(n^2)$

Square of the elements.

- A lot of sorting algorithms are $O(n^2)$.
- Nested `for` loops

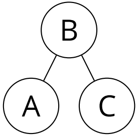
```
for i in range(n):
    for j in range(n):
        pass
```

- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.

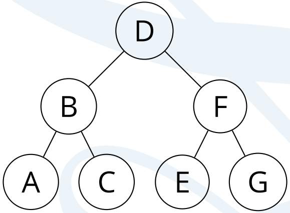
$O(\log n)$

Logarithmic time.

- Bit more complicated.
- Binary search.



$$n = 3$$
$$O(\log n) = 1.58 \Rightarrow 1$$

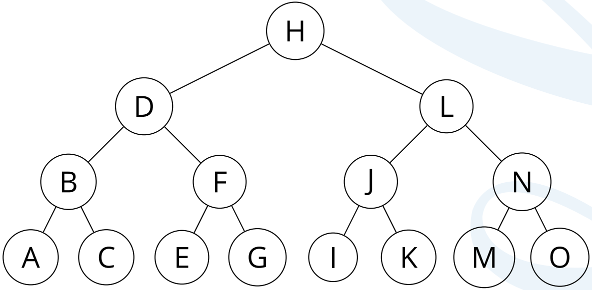


$$n = 7$$
$$O() = 2.81 \Rightarrow 2$$

$O(\log n)$ cont.

$O(\log n)$ complexity increases very slowly.

- $O()$ for a hundred items in only 6.
- $O()$ for a trillion item is only 39!



$$n = 15$$
$$O() = 3.91 \Rightarrow 3$$

$O(n \log n)$

More logarithmic time.

- Looks more difficult than it is.
- $O(n \log n)$ means, do $O(\log n)$ n times.
- E.g. binary search for n items.
  - Binary search is $O(\log n)$.
  - Doing n binary searches.
  - So $O(n \log n)$.

$O(2^n)$

Exponential time.

- Very, very bad.
- Each additional value doubles the time/space.
- Doesn't scale.

$O(n!)$

Factorial time.

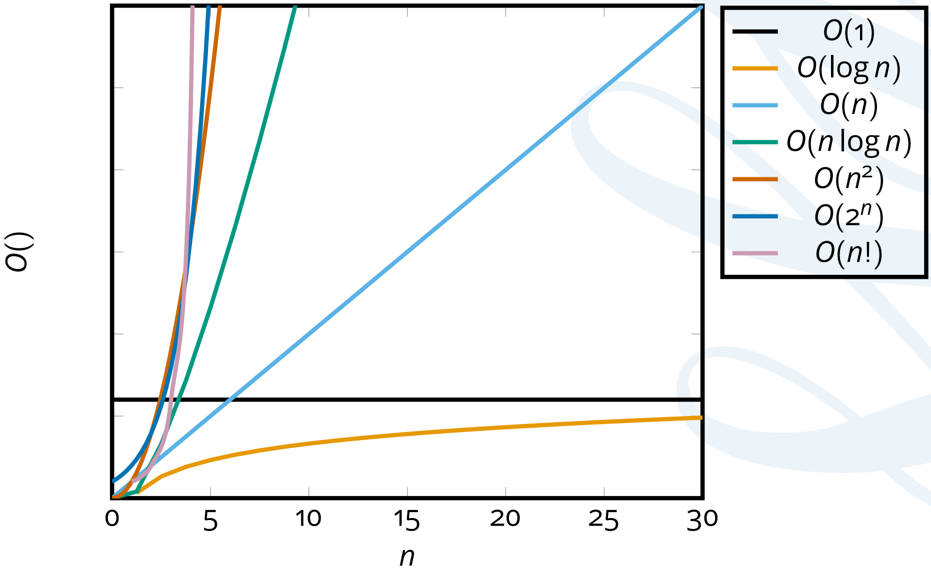
- The worst.
- Every possible combination of n items.
- Brute force travelling salesman is $O(n!)$.
- Totally impractical even for small values of n.

Relative performance

Different $O()$ == wildly different complexity.

		n = 2	10	100
Best	$O(1)$	1	1	1
	$O(\log n)$	1	3	6
$\uparrow$	$O(n)$	2	10	100
$\downarrow$	$O(n \log n)$	2	33	664
	$O(n^2)$	4	100	10000
	$O(2^n)$	4	1024	$1.27 \cdot 10^{30}$
Worst	$O(n!)$	2	3628800	$9.33 \cdot 10^{157}$

Comparison



Scalability

Scalability isn't efficiency.

- A good $O(n^2)$ implementation can be better than a bad $O(n)$.
  - For a while.
- Eventually, as n increases, $O(n)$ will always outperform $O(n^2)$ etc.

Recap

Profiling determines the actual performance of your code.

- .

O() notation is how your code should scale.

- Theoretically.
- Lots of real world issues can mess it up.

The End