

# COMP90038 Algorithms and Complexity

## Growth Rate and Algorithm Efficiency

Michael Kirley

Lecture 3

Semester 1, 2019

# Assessing Algorithm “Efficiency”

Resources consumed: **time** and **space**.

We want to assess efficiency as a function of input size:

- Mathematical vs empirical assessment
- Average case vs worst case

Knowledge about input peculiarities may affect the choice of algorithm.

The right choice of algorithm may also depend on the programming language used for implementation.

# Running Time Dependencies

There are many things that a program's running time depends on:

- 1 The complexity of the algorithms used
- 2 Input to the program
- 3 Underlying machine, including memory architecture
- 4 Language/compiler/operating system

Since we want to compare algorithms, we ignore (3) and (4); just consider units of time.

Use a natural number  $n$  as measure of (2)—size of input.

Express (1) as a function of  $n$ .

# Estimating Time Consumption

If  $c$  is the cost of a **basic operation** and  $g(n)$  is the number of times the operation is performed for input of size  $n$ ,

then running time  $t(n) \approx c \cdot g(n)$ .

# Examples: Input Size and Basic Operation

Problem	Size measure	Basic operation
Search in list of $n$ items	$n$	Key comparison
Multiply two matrices of floats	Matrix size (rows times columns)	Float multiplication
Compute $a^n$	$\log n$	Float multiplication
Graph problem	Number of nodes and edges	Visiting a node

# Best, Average, or Worst Case?

The running time  $t(n)$  may well depend on more than just  $n$ .

**Worst-case** analysis makes the most adverse assumptions about input.

**Best-case** analysis makes optimistic assumptions.

**Average-case** analysis aims to find the **expected** running time across all possible input of size  $n$ .

(Note: This is not an average of the worst and best cases.)

**Amortised** analysis takes the context of running an algorithm into account and calculates cost **spread over many runs**.

# Large Input Is What Matters

Small input does not provide a stress test for an algorithm.

As an alternative to Euclid's algorithm (Lecture 1) we can find the greatest common divisor of  $m$  and  $n$  by testing each  $k$  no greater than the smaller of  $m$  and  $n$ , to see if it divides both.

For small input  $(m, n)$ , both these versions of  $gcd$  are fast.

Only as we let  $m$  and  $n$  grow large do we witness (big) differences in performance.

# The Tyranny of Growth Rate

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
$10^1$	3	$10^1$	$3 \cdot 10^1$	$10^2$	$10^3$	$10^3$	$4 \cdot 10^6$
$10^2$	7	$10^2$	$7 \cdot 10^2$	$10^4$	$10^6$	$10^{30}$	$9 \cdot 10^{157}$
$10^3$	10	$10^3$	$1 \cdot 10^4$	$10^6$	$10^9$	—	—

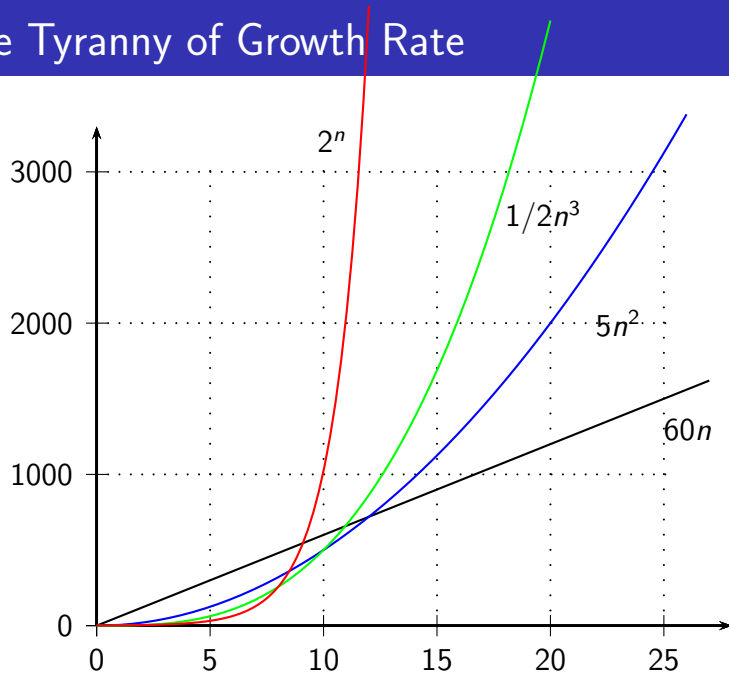
$10^{30}$  is one thousand times the number of nano-seconds since the Big Bang.

At a rate of a trillion ( $10^{12}$ ) operations per second, executing  $2^{100}$  operations would take a computer in the order of  $10^{10}$  years.

That is more than the estimated age of the Earth.



# The Tyranny of Growth Rate



# Functions Often Met in Algorithm Classification

1: Running time independent of input.

$\log n$ : Typical for “divide and conquer” solutions, for example, lookup in a balanced search tree.

Linear: When each input element must be processed once.

$n \log n$ : Each input element processed once and processing involves other elements too, for example, sorting.

$n^2$ ,  $n^3$ : Quadratic, cubic. Processing all pairs (triples) of elements.

$2^n$ : Exponential. Processing all subsets of elements.

# Asymptotic Analysis

We are interested in the **growth rate** of functions:

- Ignore constant factors
- Ignore small input sizes

# Asymptotics

$$f(n) \prec g(n) \text{ iff } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

That is:  $g$  approaches infinity faster than  $f$ . For example,

$$1 \prec \log n \prec n^\epsilon \prec n^c \prec n^{\log n} \prec c^n \prec n^n$$

where  $0 < \epsilon < 1 < c$ .

In asymptotic analysis, **think big!**

For example,  $\log n \prec n^{0.0001}$ , even though for  $n = 10^{100}$ ,  $100 > 1.023$ .

# Big-Oh Notation

$O(g(n))$  denotes the set of functions that grow no faster than  $g$ , asymptotically.

We write

$$t(n) \in O(g(n))$$

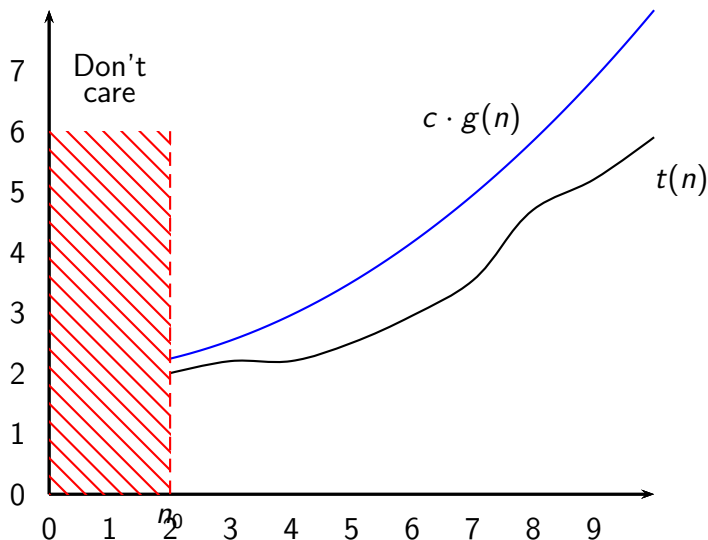
when, for some  $c$  and  $n_0$ ,

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

For example,

$$1 + 2 + \cdots + n \in O(n^2)$$

# Big-Oh: What $t(n) \in O(g(n))$ Means



# Big-Oh Pitfalls

Levitin's notation  $t(n) \in O(g(n))$  is meaningful, but not standard.

Other authors use  $t(n) = O(g(n))$  for the same thing.

As  $O$  provides an upper bound, it is correct to say both  $3n \in O(n^2)$  and  $3n \in O(n)$  (so you can see why using '=' is confusing); the latter,  $3n \in O(n)$ , is of course more precise and useful.

Note that  $c$  and  $n_0$  may be large.

# Big-Omega and Big-Theta

$\Omega(g(n))$  denotes the set of functions that grow no slower than  $g$ , asymptotically, so  $\Omega$  is for **lower** bounds.

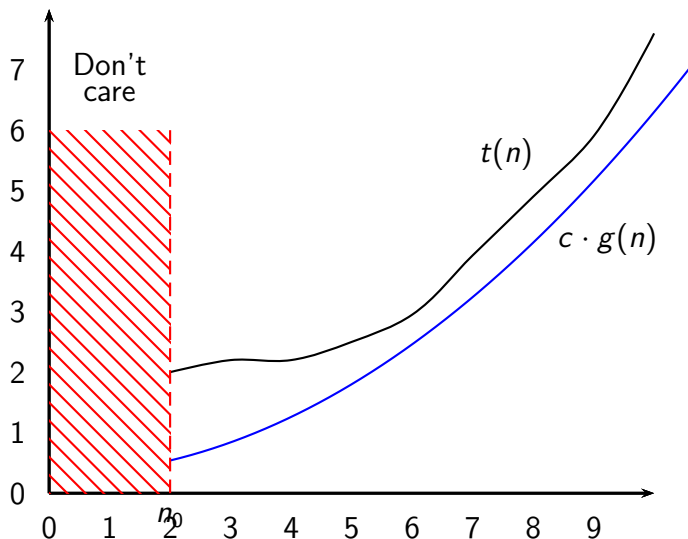
$t(n) \in \Omega(g(n))$  iff  $n > n_0 \Rightarrow t(n) > c \cdot g(n)$ , for some  $n_0$  and  $c$ .

$\Theta$  is for **exact** order of growth.

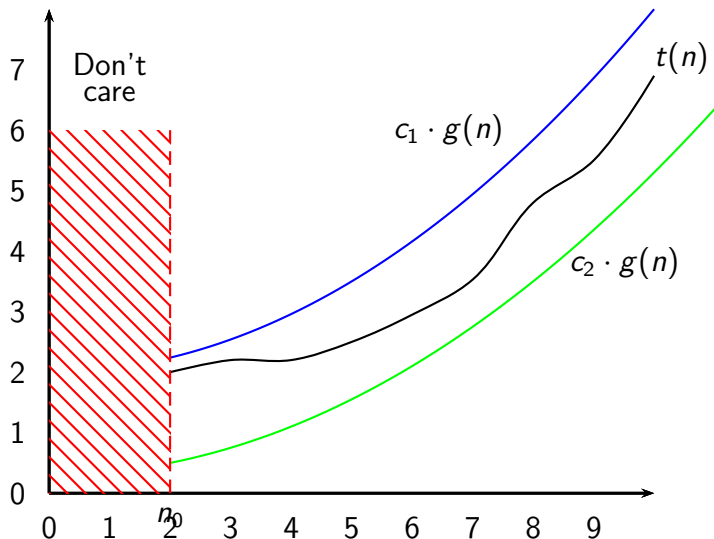
$t(n) \in \Theta(g(n))$  iff  $t(n) \in O(g(n))$  **and**  $t(n) \in \Omega(g(n))$ .



# Big-Omega: What $t(n) \in \Omega(g(n))$ Means



# Big-Theta: What $t(n) \in \Theta(g(n))$ Means



# Establishing Growth Rate

We can use the definition of  $O$  directly.

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

**Exercise:** Use this to show that

$$1 + 2 + \dots + n \in O(n^2)$$

Also show that

$$17n^2 + 85n + 1024 \in O(n^2)$$



# Next Up

We go through some examples of time complexity analysis for specific algorithms.