THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 3: Growth Rate and Algorithm Efficiency
(with thanks to Harald Søndergaard)

Toby Murray

toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

@tobycmurray

Note on Tutorials

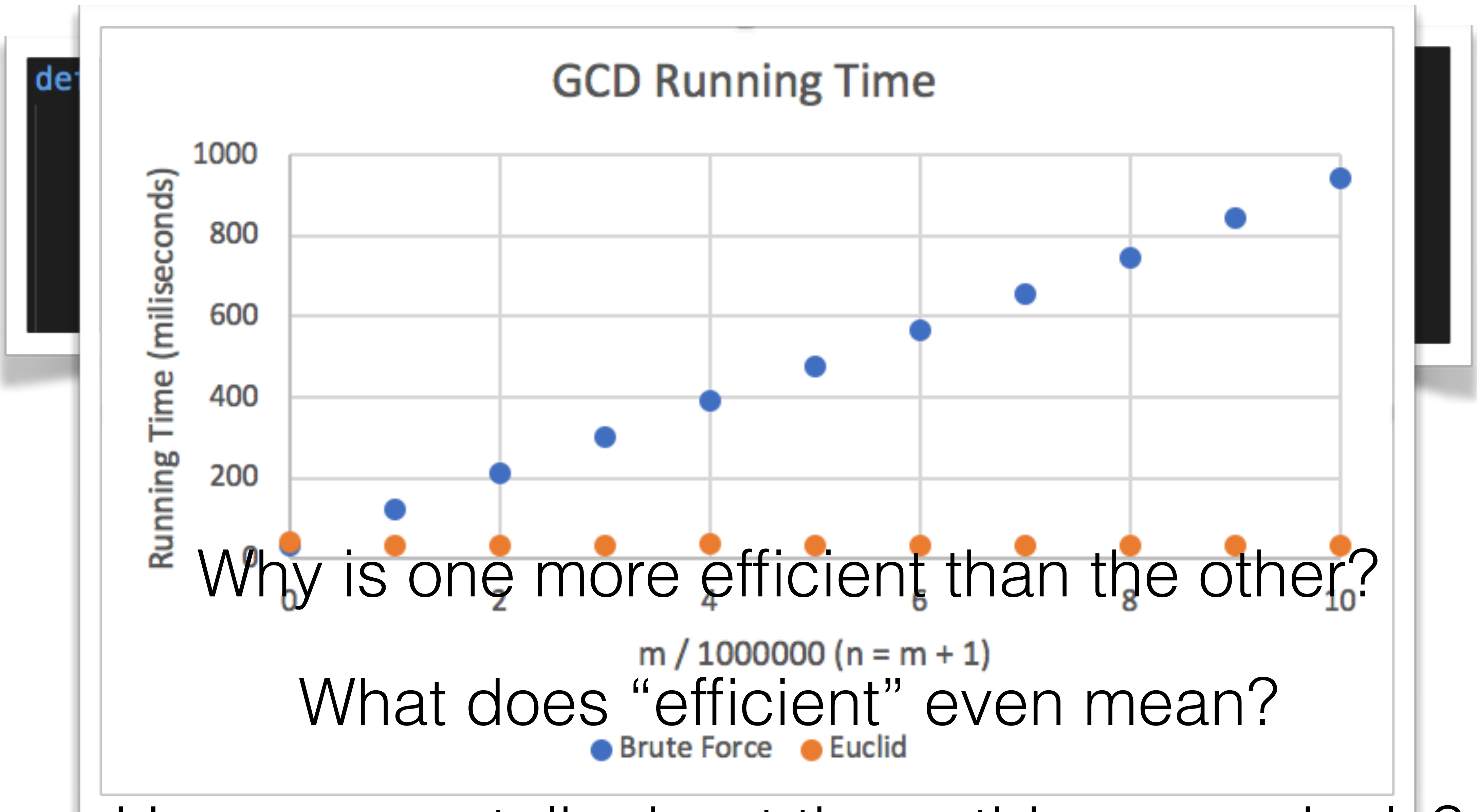
- Many tutorial sessions full but some also quite empty.
- If you are looking for a smaller class to attend, try:
  - Naiyun - Monday 12.00 - David Caro 205 (2 students)
  - Partha - Monday 15.15 - David Caro 205 (1 student)
  - Jiajia - Monday 15.15 - David Caro 201 (5 students)
  - Assaf - Monday 14.15 - David Caro 205 (7 students)

Update

- Compulsory Quizzes (first one closes Tuesday Week 3)
- Tutorials started this week
- Background knowledge catch-up tutorials:
  - Weeks 2 and 3
  - Wednesday 1-2pm, David Caro, Room 210
  - Friday 11am-12pm, Alice Hoy, Room 330
- Consultation Hours
- Discussion Board

Algorithm Efficiency

Two **algorithms** for computing gcd:



Why is one more efficient than the other?

What does "efficient" even mean?

How can we talk about these things precisely?

Linear Search Example

THE UNIVERSITY OF MELBOURNE

A: Y x: 7 n: 7 j: [illegible]

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

A[j]

↓

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Let's trace the execution of find(Y,7,7)

(returns 4)

Linear Search Example

THE UNIVERSITY OF MELBOURNE

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

How many times does the loop run to find 7?

5.

How many times does the loop run to find 6? 1.

How many times does the loop run to find 99? 7.

(the length of the array)

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. 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 to quantify (2)—size of the input

- Express (1) as a function of n

Linear Search Example

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

How should we measure the size, *n*, of the input to this algorithm?

n = the length of the array

How should we quantify the cost to run this algorithm?

roughly, number of times the loop runs

(later in this lecture we will be more precise)

Linear Search Example

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

What is the **worst case** input?

an array that doesn't contain the item, x, we are searching for

Worst case time complexity: *n*
(since the loop runs *n* times in that case)

Linear Search Example

THE UNIVERSITY OF MELBOURNE

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

What is the **best case** input?

an array that has the item, x,
we are searching for in the first position

Best case time complexity: 1
(since the loop runs once in that case)

Estimating Time Consumption

THE UNIVERSITY OF MELBOURNE

- Number of loop iterations is not a good estimate of running time.
- Better is to identify the algorithm's **basic operation** and how many times it is performed
- If c is the cost of a **basic operation** and $g(n)$ is the number of times the operation is performed for input size n,

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

Linear Search Example

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

What is the **basic operation** here?

the comparison $A[j] = x$

Rule of thumb: the most expensive operation executed each time in the inner-most loop of the program

Examples: Input Size and Basic Operation

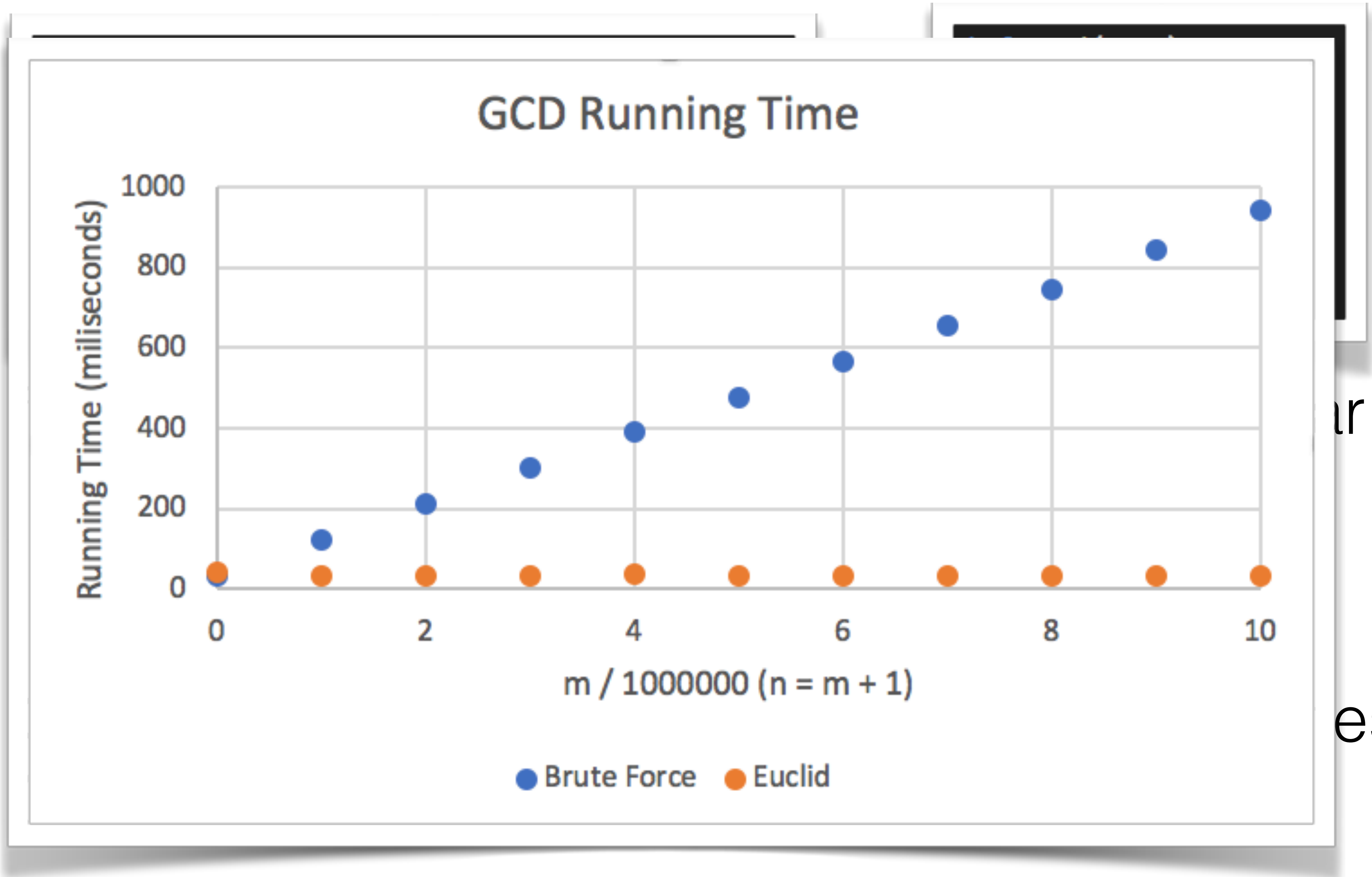
Problem	Size Measure	Basic Operation
Search in a list of n items	n	Key comparison
Multiply two matrices of floats	Matrix size (rows x columns)	Float multiplication
Compute a^n	$\log n$	Float multiplication
Graph problem	Number of nodes and edges	Visiting a node

Best, Average and Worst Case

- The running time *t(n)* may well depend on more than just *n*
- **Worse case:** analysis makes the most pessimistic assumptions about the input
- **Best case:** analysis makes the most optimistic assumptions about the input
- **Average case:** analysis aims to find the **expected** running time across all possible input of size *n*
(Note: **not** an average of the worst and best cases)
- **Amortised** analysis takes context of running an algorithm into account, calculates cost **spread over many runs**. Used for "self-organising" data structures that adapt to their usage

Large Input is what Matters

- Small input does not properly stress an algorithm



ar

- ess

Guessing Game Example

THE UNIVERSITY OF MELBOURNE

- Guess which number I am thinking of, between 1 and *n* (inclusive). I will tell you if it is higher or lower than each guess.

1 50 51 75 100

Wrong. My number is higher than 50.

Wrong. My number is lower than 75.

We are **halving** the search space each time.

Basic operation:

(Worse case) complexity: log *n*

The Tyranny of Growth Rate

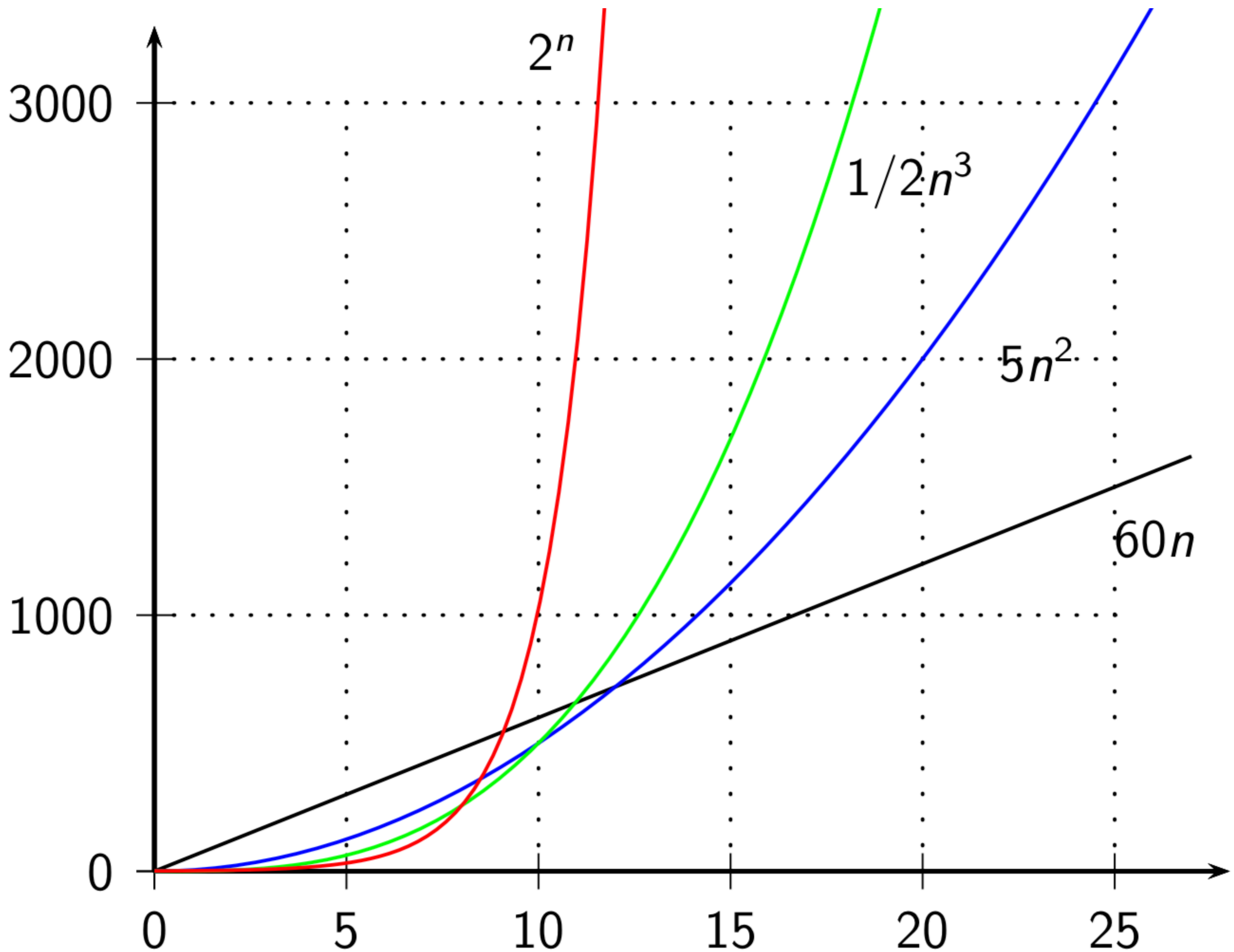
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3	10^1	$3 \cdot 10^1$	10^2	10^3	10^3	$4 \cdot 10^6$
10^2	7	10^2	$7 \cdot 10^2$	10^4	10^6	10^{30}	$9 \cdot 10^{157}$
10^3	10	10^3	$1 \cdot 10^4$	10^6	10^9	-	-

10^{30} is 1,000 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 an conquer” solutions, for example lookup in a balanced search tree
- **Linear** (**n**): When each input 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

- $f(n) < g(n)$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
- That is, g approaches infinity faster than f
- $1 < \log n < n^{\varepsilon} < n^{c} < n^{\log n} < c^{n} < n^{n}$
  where $0 < \varepsilon < 1 < c$
- In asymptotic analysis, **think big!**
  - e.g., $\log n < n^{0.0001}$, even though for $n = 10^{100}$, $100 > 1.023$.
  - Try it for $n = 10^{1000000}$

Big-Oh Notation

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

- **Formal definition:** We write

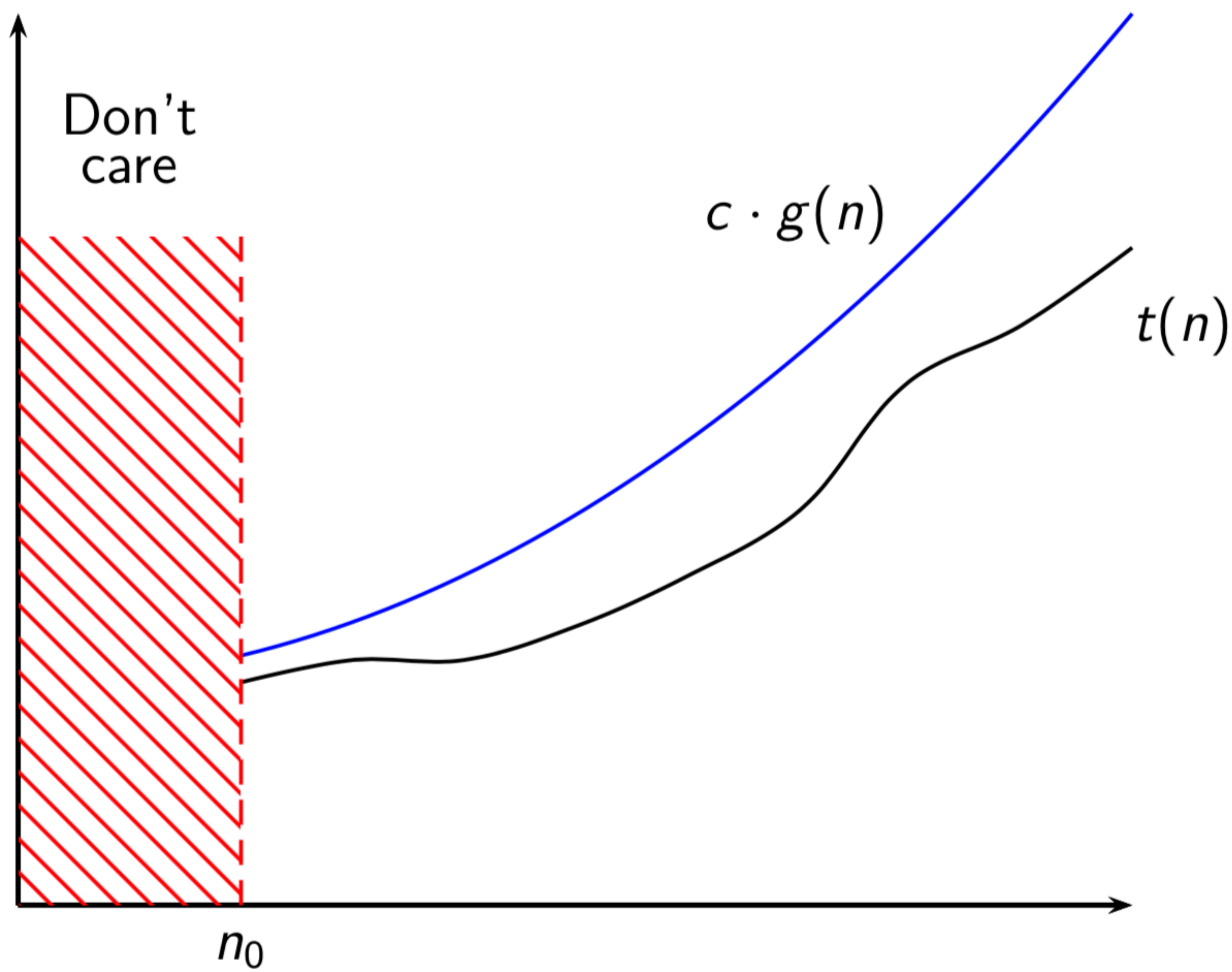
$$t(n) \in O(g(n))$$

when, **for some *c* and $\boldsymbol{n_0}$**

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

- For example: $1 + 2 + \ldots + n \in O(n^2)$

Big-Oh: What t(n) ∈ O(g(n)) Means



Big-Oh Pitfalls

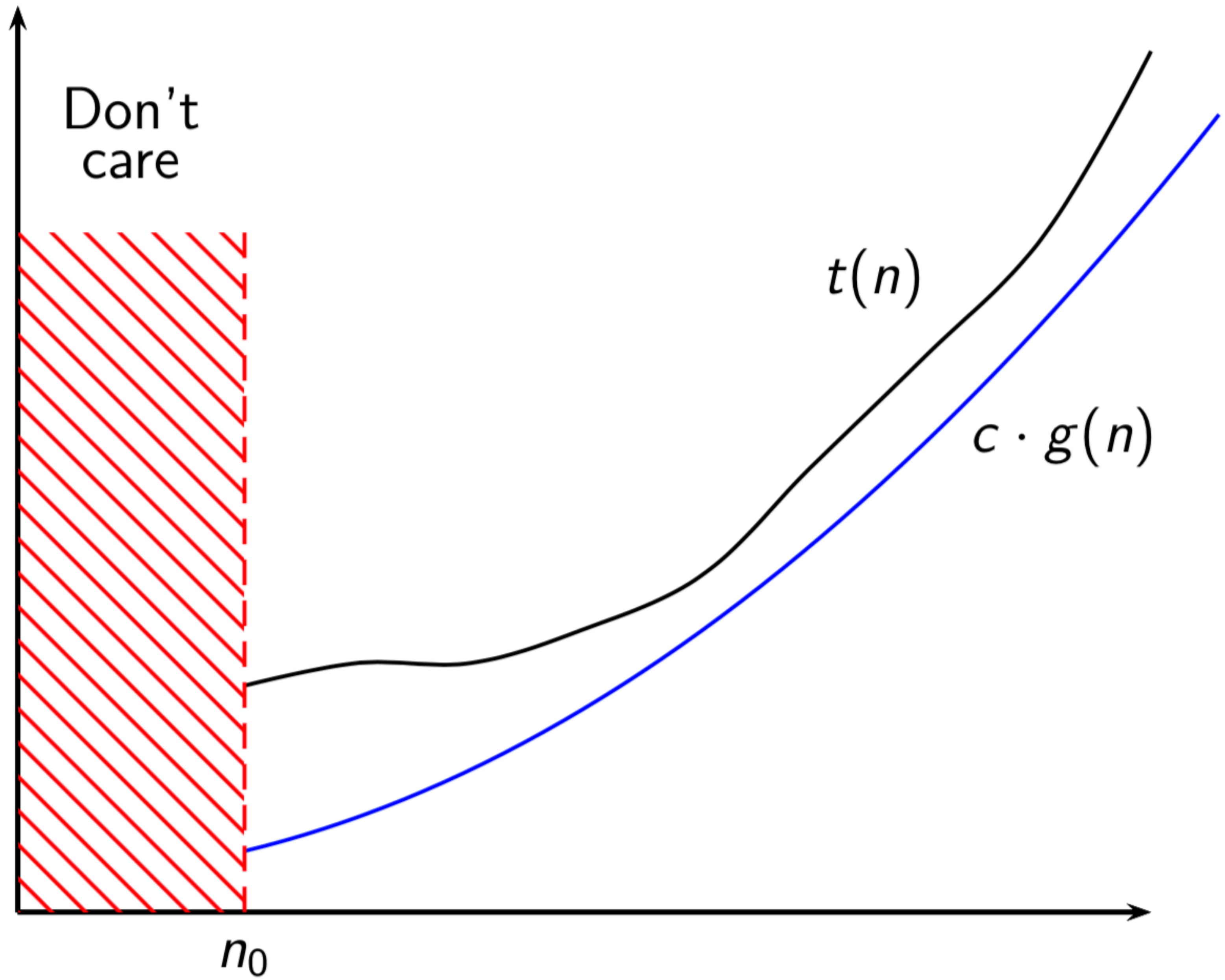
- Levitin's notation $t(n) \in O(g(n))$ is meaningful, but not standard.
- Other authors use $t(n) = O(g(n))$ for the same thing.
- As O provides an upper bound, it is correct to say both $3n \in O(n^2)$ and $3n \in O(n)$ (so you can see why using '=' is confusing); the latter, $3n \in O(n)$, is of course more precise and useful.
- Note that c and n_0 may be large.

Big-Omega and Big-Theta

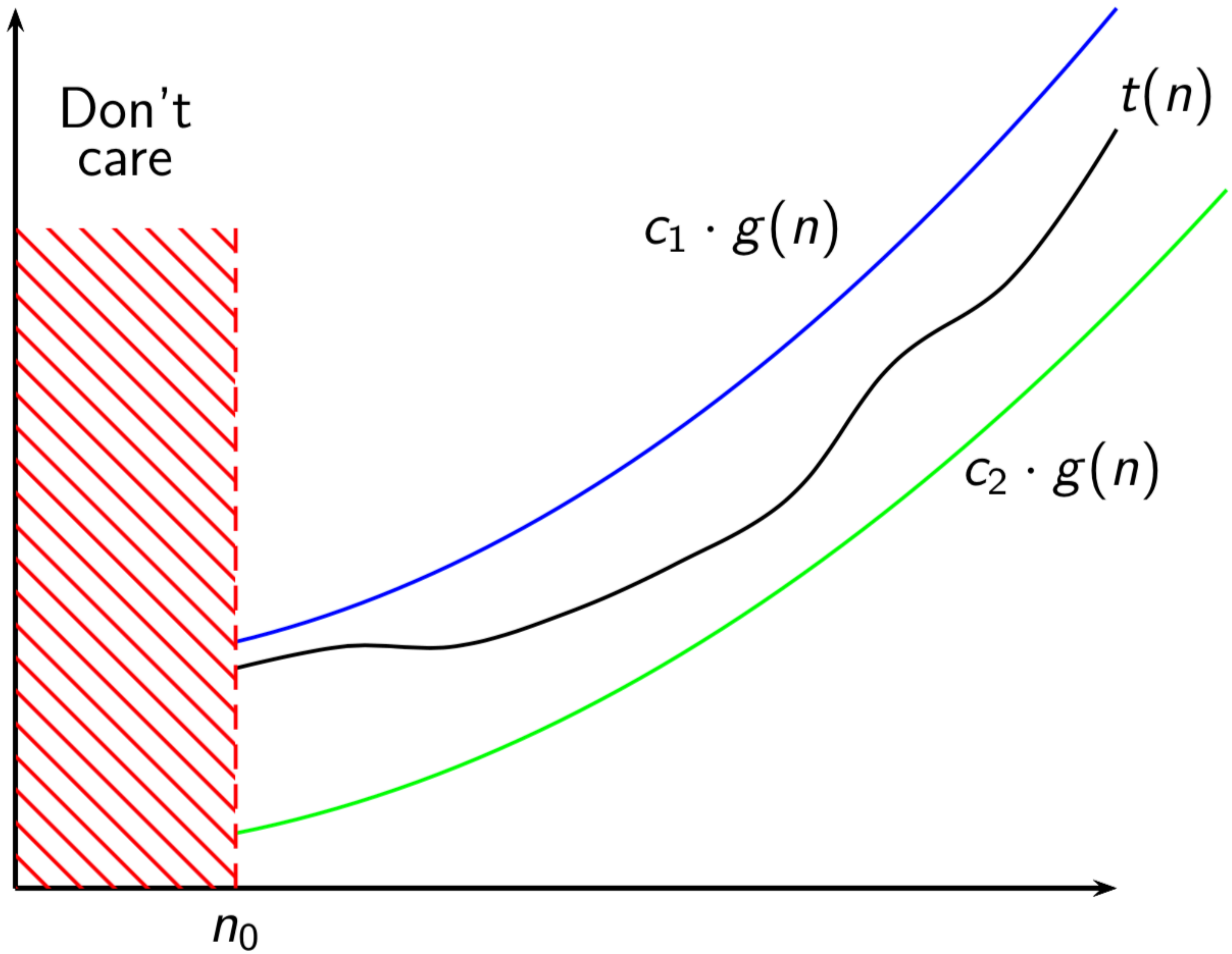
- **Big Omega:** $\Omega(g(n))$ denotes the set of functions that grow no slower than g, asymptotically, so Ω is for **lower bounds**.
  - $t(n) \in \Omega(g(n))$ iff $n > n_0 \;\Rightarrow\; t(n) > c \cdot g(n)$,

    for some n_0 and c.
- **Big Theta:** Θ is for **exact** order of growth.
  - $\mathrm{t}(n) \in \Theta(g(n))$ 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.

  $t(n) \in O(g(n))$ iff: $n > n_0 \Rightarrow t(n) < c \cdot g(n)$

- **Exercise:** use this to show that
  $1 + 2 + \ldots + n \in O(n^2)$

- Also show that:
  $17n^2 + 85n + 1024 \in O(n^2)$

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

THE UNIVERSITY OF MELBOURNE

Find some c and n_0 such that, for all $n > n_0$

$$1 + 2 + \ldots + n < c \cdot n^2$$

$$1 + 2 + \ldots + n$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2 + n}{2}$$

$$< n^2 + n \quad (\text{for } n > 0)$$

$$< n^2 + n^2 \quad (\text{for } n > 1)$$

$$= 2n^2$$

APPENDIX A

Useful Formulas for the Analysis of Algorithms

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

Choose $n_0 = 1$, $c = 2$

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

Find some c and n_0 such that, for all $n > n_0$

$$17n^2 + 85n + 1024 < c \cdot n^2$$

Guess c = 18 Need to prove:

$$17n^2 + 85n + 1024 < 18n^2$$

i.e. $85n + 1024 < n^2$

Guess $n_0 = 1024$ Check if: $85n_0 + 1024 < n_0^2$

$$85 \cdot 1024 + 1024 < 1024 \cdot 1024$$

i.e. $86 \cdot 1024 < 1024 \cdot 1024$ Clearly true.

Choose c = 18, n_0 = 1024

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

Find some c and n_0 such that, for all $n > n_0$

$$17n^2 + 85n + 1024 < c \cdot n^2$$

Alternative: Let c = 17 + 85 + 1024

$$17n^2 + 85n + 1024$$

$$< 17n^2 + 85n^2 + 1024n^2 \quad (\text{for } n > 1)$$

$$= (17 + 85 + 1024)n^2$$

Choose c = 17 + 85 + 1024, n_0 = 1

Of course, this works for *any* polynomial.