

# COMP20007 Design of Algorithms



## Growth Rate and Algorithm Efficiency

---

Lars Kulik

Lecture 3

Semester 1, 2023

# 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  $n$ .

# The RAM Word Model

## Assumptions

- Data is represented in words of fixed length in bits (e.g., 64-bit)
- Fundamental operations on words take one unit of time

Basic arithmetic:  $+$ ,  $-$ ,  $\times$ ,  $/$

Memory access: load, store  $\Rightarrow$  pipeline  $\Rightarrow$   $cp1 \rightarrow 1$

Comparisons:  $<$ ,  $>$ ,  $=$ ,  $\neq$ ,  $\geq$ ,  $\leq$

Logical operators:  $\&\&$ ,  $\|\|$

Bitwise operators:  $\&$ ,  $|$

The goal of analysis is to count the operations based on parameters such as input size or problem size.

# 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) $m \times n$	Float multiplication
Graph problem	Number of nodes and edges $m$ nodes $n$ 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. Are they the worst  $n$  things your algorithm could see?

✱ **Best-case** analysis makes optimistic assumptions. Are they the best  $n$  things the algorithm could see?

**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 but assumes that your input is drawn randomly from all possible inputs of size  $n$ .)

avg  $n(\text{random}) \rightarrow E[t(n)]$

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

# Average-case Analysis: Sequential Search

```
function SEQUENTIALSEARCH(size: n A[0..n-1], K)
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n then
        return i
    else
        return -1
```

If the probability of a successful search is equal to  $p$  ( $0 \leq p \leq 1$ ), then the average number of average number of key comparisons  $C_{avg}(n)$  is

$$p \times \underbrace{(n+1)/2}_{\substack{\uparrow \\ \text{Expected}}} + n \times (1-p).$$



# Large Input Is What Matters

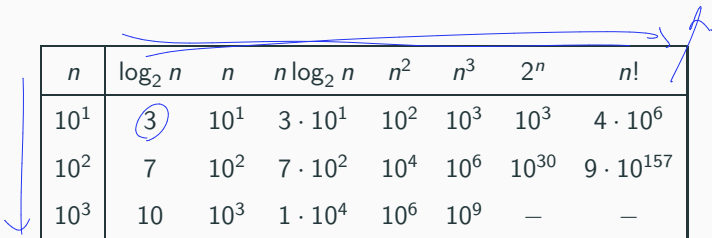
Small input does not provide a stress test for an algorithm.

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



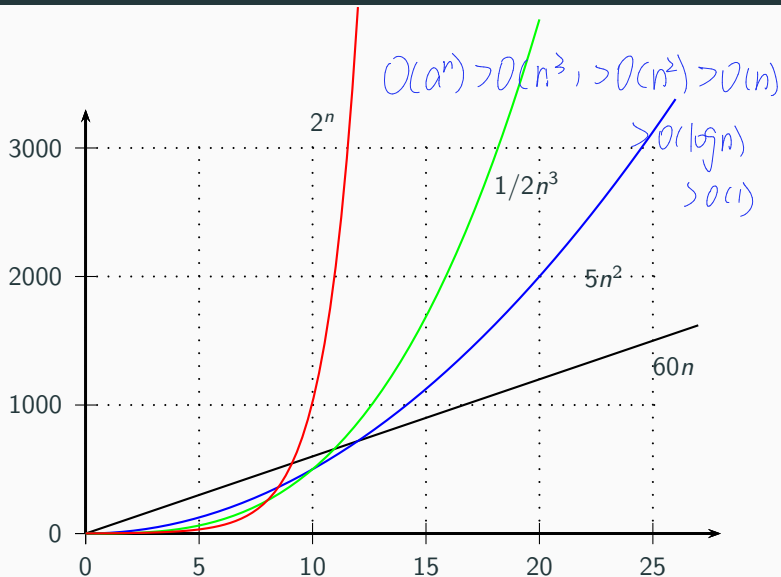
$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
$10^1$	3	$10^1$	$3 \cdot 10^1$	$10^2$	$10^3$	$10^3$	$4 \cdot 10^6$
$10^2$	7	$10^2$	$7 \cdot 10^2$	$10^4$	$10^6$	$10^{30}$	$9 \cdot 10^{157}$
$10^3$	10	$10^3$	$1 \cdot 10^4$	$10^6$	$10^9$	—	—

$10^{30}$  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.

$n$ : 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

$f(n) \prec g(n)$  if and only if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ .

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

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

$$t(n) \in O(g(n)) \Rightarrow t(n) < g(n) \cdot c \quad (n \rightarrow \infty) \\ \exists n > n_0$$

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

We write

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

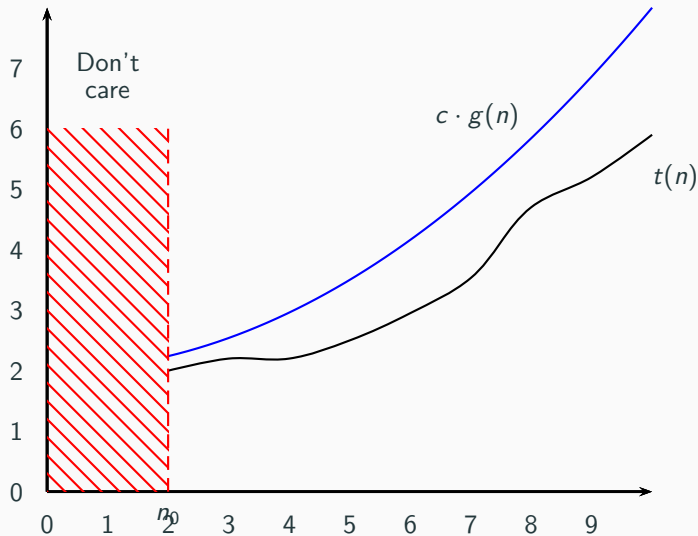
when, for some  $c$  and  $n_0$ ,

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

For example,

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

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





# Big-Oh Pitfalls

$O \rightarrow$  upper

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

$\Omega$  - lower

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

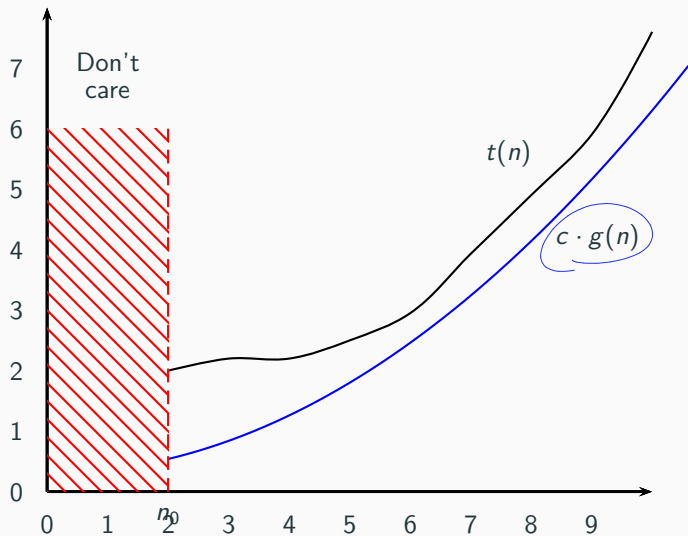
$t(n) \in \Omega(g(n))$  iff  $n > n_0 \Rightarrow t(n) > c \cdot g(n)$ , for some  $n_0$  and  $c$ .

$\Theta$  is for exact order of growth.

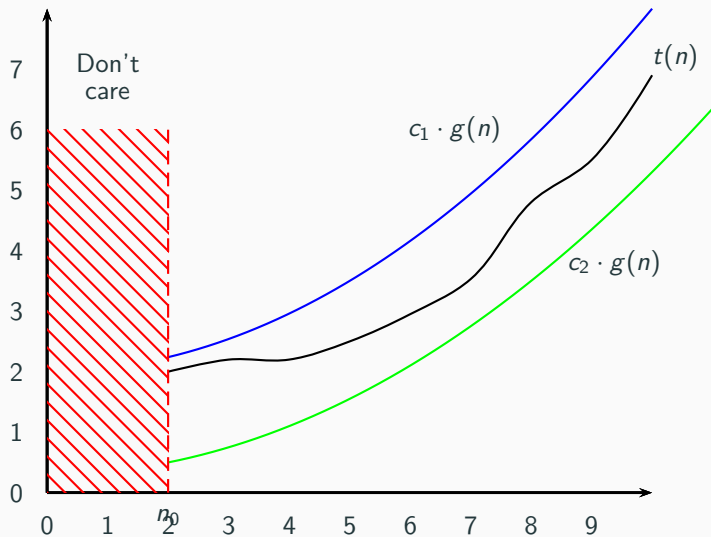
$t(n) \in \Theta(g(n))$  iff  $t(n) \in O(g(n))$  and  $t(n) \in \Omega(g(n))$ .

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

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

Also show that

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



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

# COMP20007 Design of Algorithms

## Analysis of Algorithms

---

Lars Kulik

Lecture 4

Semester 1, 2023

# Establishing Growth Rate

In the last lecture we proved  $t(n) \in O(g(n))$  for some cases of  $t$  and  $g$ , using the definition of  $O$  directly:

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

for some  $c$  and  $n_0$ . A more common approach uses

Standard notation.

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g \\ c & \text{implies } t \text{ and } g \text{ have same order of growth} \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g \end{cases}$$

Use this to show that  $1000n \in O(n^2)$ .





# L'Hôpital's Rule

Often it is helpful to use L'Hôpital's rule:

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

where  $t'$  and  $g'$  are the **derivatives** of  $t$  and  $g$ .

For example, we can show that  $\log_2 n$  grows slower than  $\sqrt{n}$ :

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n \ln 2}}{\frac{1}{2\sqrt{n}}} = \lim_{n \rightarrow \infty} \frac{2}{\ln 2} \frac{1}{\sqrt{n}} = 0 \Rightarrow \log_2 n < \sqrt{n}$$

# Induction Trap (Polya)

- $A(n)$ : All horses are the same colour
- Base case:  $A(1)$  is trivially true (only one horse)
- Assume in a set of  $n$  horses, all are the same colour
  - For a set of  $n + 1$  horses, take the subsets  $\{1, \dots, n\}$  and  $\{2, \dots, n + 1\}$ .
  - Both subsets are of size  $n$ , so all horses are the same colour in each subset (by inductive hypotheses).
  - Since  $n - 1$  of the horses are the same in both sets, the horses in both sets must be all the same colour, hence all  $n + 1$  horses are the same colour.
- What went wrong?

$\{1, 2\}$  Can not say  $\{1\}$  the same as  $\{2\}$  just from  $A(1)$  is right.

## Example: Finding the Largest Element in a List

```
function MAXELEMENT( $A[0..n-1]$ )  
     $max \leftarrow A[0]$   
    for  $i \leftarrow 1$  to  $n-1$  do  $\Theta(n)$   
        if  $A[i] > max$  then  
             $max \leftarrow A[i]$   
    return  $max$ 
```

We count the number of comparisons executed for a list of size  $n$ :

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$$

## Example: Selection Sort

```
function SELSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 0$  to  $n-2$  do  
     $min \leftarrow i$   
    for  $j \leftarrow i+1$  to  $n-1$  do  
      if  $A[j] < a[min]$  then  
         $min \leftarrow j$   
    swap  $A[i]$  and  $A[min]$ 
```

$\Theta(n^2)$

We count the number of comparisons executed for a list of size  $n$ :

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = (n-1)^2 - \sum_{i=0}^{n-2} i \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

## Example: Matrix Multiplication

**function**

MATRIXMULT( $A[0..n-1, 0..n-1]$ ,  $B[0..n-1, 0..n-1]$ )

**for**  $i \leftarrow 0$  to  $n-1$  **do**

**for**  $j \leftarrow 0$  to  $n-1$  **do**

$C[i, j] \leftarrow 0.0$

**for**  $k \leftarrow 0$  to  $n-1$  **do**

$C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$

**return**  $C$

$(\cdot) (n^3)$

The number of multiplications executed for a list of size  $n$  is:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$



# Analysing Recursive Algorithms

Let us start with a simple example:

**function**  $F(n)$

**if**  $n = 0$  **then return** 1

**else return**  $F(n - 1) \cdot n$

$$M(0) = 0$$

$$M(n) = M(n-1) + 1$$

$$\Rightarrow M(n) = n \Rightarrow \Theta(n)$$

The basic operation here is the multiplication.

We express the cost recursively as well:

$$M(0) = 0$$

$$M(n) = M(n - 1) + 1 \quad \text{for } n > 0$$

To find a **closed form**, that is, one without recursion, we usually try “telescoping”, or “backward substitutions” in the recursive part.

# Telescoping

The recursive equation was:

$$M(n) = M(n-1) + 1 \quad (\text{for } n > 0)$$

Use the fact  $M(n-1) = M(n-2) + 1$  to expand the right-hand side:

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

and keep going:

$$\dots = [M(n-3) + 1] + 2 = M(n-3) + 3 = \dots = M(n-n) + n = n$$

where we used the base case  $M(0) = 0$  to finish.

## A Second Example: Binary Search in Sorted Array

```
function BINARYSEARCH( $A[], lo, hi, key$ )  
    if  $lo > hi$  then return  $-1$   
     $mid \leftarrow lo + (hi - lo)/2$   
    if  $A[mid] = key$  then return  $mid$   
    else  
        if  $A[mid] > key$  then  
            return BINARYSEARCH( $A, lo, mid - 1, key$ )  
        else return BINARYSEARCH( $A, mid + 1, hi, key$ )
```

$C(0) = 0$   
 $\left\{ \begin{array}{l} C(n) = C(n/2) + 1 \\ \Rightarrow C(n) = \Theta(\log n) \end{array} \right.$

The basic operation is the key comparison. The cost, recursively, in the worst case:

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n/2) + 1 \quad \text{for } n > 0 \end{aligned}$$



# Telescoping

A **smoothness rule** allows us to assume that  $n$  is a power of 2.

The recursive equation was:

$$C(n) = C(n/2) + 1 \text{ (for } n > 0\text{)}$$

Use the fact  $C(n/2) = C(n/4) + 1$  to expand, and keep going:

$$\begin{aligned} C(n) &= C(n/2) + 1 \\ &= [C(n/4) + 1] + 1 \\ &= [[C(n/8) + 1] + 1] + 1 \\ &\vdots \\ &= \underbrace{[[\dots [[C(0) + 1] + 1] + \dots + 1] + 1]}_{1 + \log_2 n \text{ times}} \end{aligned}$$

Hence  $C(n) = \Theta(\log n)$ .

## Logarithmic Functions Have Same Rate of Growth

In  $O$ -expressions we can just write “log” for any logarithmic function, no matter what its base is.

Asymptotically, all logarithmic behaviour is the same, since

$$\log_a x = (\log_a b)(\log_b x)$$

So, for example, if  $\ln$  is the natural logarithm then

$$\begin{aligned}\log_2 n &\in O(\ln n) \\ \ln n &\in O(\log_2 n)\end{aligned}$$

Also note that since  $\log n^c = c \cdot \log n$ , we have, for all constants  $c$ ,

$$\log n^c = O(\log n)$$

## Summarising Reasoning with Big-Oh

$$O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$$

$$c \cdot O(f(n)) = O(f(n))$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)).$$

The first equation justifies throwing smaller summands away.

The second says that constants can be thrown away too.

The third may be used with some nested loops. Suppose we have a loop which is executed  $O(f(n))$  times, and each execution takes time  $O(g(n))$ . Then the execution of the loop takes time  $O(f(n) \cdot g(n))$ .

## Some Useful Formulas

From Stirling's formula:

$$n! \leq \exp\left((n + \frac{1}{2}) \ln n\right)$$

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

Some useful sums:

$$\sum_{i=0}^n i^2 = \frac{n}{3}(n + \frac{1}{2})(n + 1)$$

$$\sum_{i=0}^n (2i + 1) = (n + 1)^2$$

$$\underbrace{\sum_{i=1}^n 1/i}_{O(\log n)} = \underbrace{\sum_{k=1}^n \frac{1}{k}}_{\lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} = \ln n + \gamma}$$

See also Levitin's Appendix A.

Levitin's Appendix B is a tutorial on recurrence relations.

You will become much more familiar with asymptotic analysis as we use it on algorithms that we meet.

We shall begin the study of algorithms by looking at **brute force** approaches.