Week 1
Course Introduction

Algorithms and Data Structures
COMP3506/7505

https://aiatsis.gov.au/explore/map-indigenous-australia

Week 1 – Course Introduction

1. Boring but important admin stuff
2. Why are we here?
3. Introduction to data structures
4. Measuring algorithms – first take
5. Model of computation
6. Algorithm Analysis

Who are we?

- Dr Joel Mackenzie – Coordinator and Lecturer
- Dr Josh Arnold – Lecturer
- **Fantastic** (GOATed) Casual Academic Staff:
  - Alexander x 2, Bahareh, Brandon, Dawn, Gabriel, Georgia, Jack, Kai, Kirra, Lachlan, Markus, Minhao, Nicholas, Ryan, Vladimir, Watheq, Yutong, Zach

`comp3506@eecs.uq.edu.au`

Who are you?

- Extremely smart, well motivated, timely, …
- ~700 students in total
  - Majority are COMP3506 (undergraduate)
    - Various programs: BCompSc, BMath, BSc, MCompSc, …
  - About 10% typically COMP7505 (postgraduate)
    - MCompSc, …

Prerequisite Knowledge

- High School Maths
  - Basic algebra
  - Logarithms (especially base 2 :-)
  - Sequences and series
- Discrete Mathematics (eg. MATH1061/7861)
  - Graphs
  - Set Theory and Relations
  - Basic Proofs
- Programming Skills (CSSE1001/7030 & CSSE2002/7023)
  - Object-oriented programming
  - The **Java** programming language

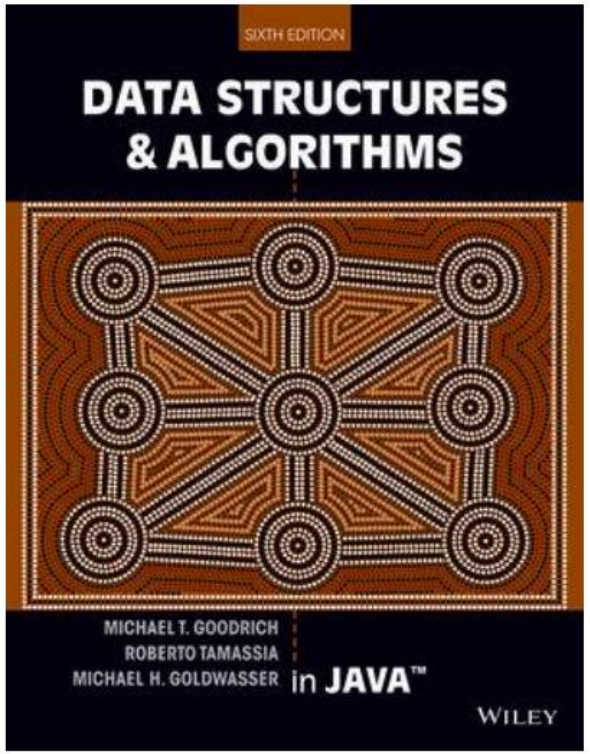
Why don't we use *<my favourite language>????*

- This is **not a programming course**
  - Techniques we explore in this course are *language agnostic*
- The goal of this course is to make you **think like a computer scientist**
- Assignments will be using Java
  - We assume you are a competent Java programmer, including building and running your own tests

What you'll learn in this course

- The goal of this course is to build an understanding of **data structures** and their associated **algorithms** so that you can select an **appropriate** set when designing software
- These algorithms and data structures form the basis of **large and complex software systems** and are necessary in order to create **efficient** and **scalable** programs
- **Think like a computer scientist!**

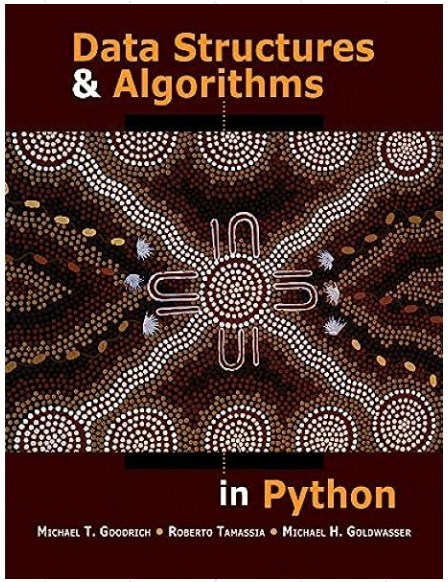
Learning Resources



Data Structures and Algorithms in Java.
M. T. Goodrich, R. Tamassia and M.H. Goldwasser

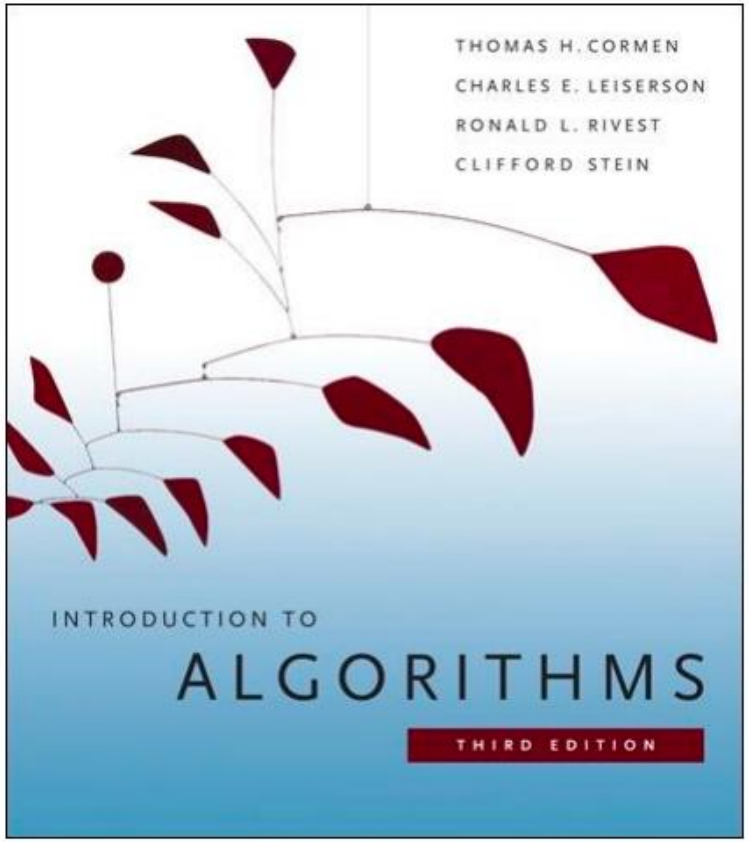
- Also known as "GTG"

Many lectures will correspond to a chapter in this book



There is a Python version available too.

Learning Resources



Introduction to Algorithms. 3rd ed.
T.H. Cormen, C.E. Leiserson,
R.L. Rivest and C. Stein

- Also known as "CLRS"

Good, in-depth, coverage of algorithms and interacting with data structures.

Extends the topics discussed in lectures and tutorials, and covers more advanced topics.

Learning Resources

- **Blackboard (LMS)**
  - Lecture slides, tutorial sheets, lecture recordings, assignment specifications
- **Ed Discussion**
  - Does what it says on the tin
- **Gradescope**
  - Assignment submissions
- **Ed Lessons (voluntary, but recommended)**
  - Some "theory to practice" modules and a Java primer

Learning Activities

Lectures

- **2 hours per week**
- Mondays from 12-2pm, 03-206
- Recorded via Echo360

Applied Classes

- **2 hours per week**
- Start in week 2
- **Very important!**

Weekly drop in hour

- Wednesday 12-1pm
- GP-South: 78-546
- Does not run in Week 3 due to Ekka

Assessment

Assessment	COMP3506 Weighting	COMP7505 Weighting
Problem Sets	10% (Best 4 of 5)	10% (Best 4 of 5)
Assignment 1	20%	20% (+ extra question)
Assignment 2	20%	20% (+ extra question)
Final exam – closed book	50%	50%

Check the course profile for more information (including due dates).

Also check out the "Plan at a Glance" document: I recommend keeping this document handy throughout the semester.

COMP3506/7505 2025 Weekly Plan: Version 0 – June 23, 2025

Week	Lecture	Staff	Applied Class	Readings	Assessment Due
1	Intro, computation models, basic algorithm analysis, asymptotic analysis	Joel	–	GTG: 4.1, 5 (Python: 3.1, 4) / CLRS: 2, 3	–
2	Recursion, analysis of algorithms, comparison-based sorting	Josh	Algorithm analysis I	GTG: 5,12.1-12.3 (Python: 4,12.1–12.3) / CLRS: 2.3, 4, 7	–
3	Linear sorting, Lists and arrays, amortization, linked data structures	Josh	Algorithm analysis II	GTG: 3, 7.1-7.2, 12.3 (Python: 5, 7.1–7.3, 12.4) / CLRS: 8, 10.1–10.2	Problem Set 1
4	Stacks, queues, trees	Josh	Sorting	GTG: 6, 7.4, 8 (Python: 1.8, 6, 8) / CLRS: 10.4	–
5	Priority queues, heaps, adaptable priority queues, maps, sets, multimaps	Josh	Stacks, queues, trees	GTG: 9, 10.1, 10.5 (Python: 9, 10.1, 10.5) / CLRS: 6	Problem Set 2
6	Hashing and hash tables, binary search trees	Joel	Priority queues and heaps	GTG: 10.2-10.3, 11.1-11.3 (Python: 10.2–10.3, 11.1–11.3) / CLRS: 11, 12	Assignment 1
7	Balanced trees (AVL, Splay)	Joel	Maps and hash tables	GTG: 11.3-11.6 (Python: 11.3–11.6) / CLRS: 13, 18	–
8	Graph basics, graph traversal, directed graph algorithms	Joel	Search trees	GTG: 14.1-14.5 (Python: 14.1–14.5) / CLRS: 22	Problem Set 3
9	Shortest path algorithms, minimum spanning trees	Joel	Graph Theory I	GTG: 14.5-14.7 (Python: 14.5–14.7) / CLRS: 21, 23, 24	–
"Mid" Semester Break					
10	*Public Holiday*	–	Graph Theory II	GTG: 13.4 (Python: 13.4) / CLRS: 16.3	Problem Set 4
11	Strings, Patterns, Tries	Joel	Exam Revision I	GTG: 13.1-13.2, 13.5 (Python: 13.1–13.2, 13.5) / CLRS: 32	–
12	Huffman coding, text compression, advanced applications	Joel	Strings and Compression	TBA	Assignment 2
13	Course review	Joel + Josh	Exam Revision II	–	Problem Set 5
Exam	–	–	–	–	Exam (50%)

Plagiarism

Plagiarism is the act of misrepresenting as one's own original work the ideas, interpretations, words or creative works of another. These include published and unpublished documents, designs, music, sounds, images, photographs, computer codes and ideas gained through working in a group. These ideas, interpretations, words or works may be found in print and/or electronic media.

Students are encouraged to read the UQ Academic Integrity and Plagiarism policy (http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct) which makes a comprehensive statement about the University's approach to plagiarism, including the approved use of plagiarism detection software, the consequences of plagiarism and the principles associated with preventing plagiarism.

Plagiarism

- Please discuss problems and potential approaches.
  - Arguing with your friends is fun!
- Please look things up online.
  - Sometimes alternative perspectives are helpful.
- Don’t copy answers.
- Don’t blindly use GenAI (it rots your brain).
- Don’t pay someone to do your assignment.
- Planning to cheat? **Come and seek help.**
- **READ the guide on the LMS!**

Some important policies...

- Extensions: Via my.uq – Please try to make these early.
  - No extensions to Problem Sets will be considered, ever.
- Late policy
  - 10% loss of marks per day for 7 days.
  - Maximum one week extension.
  - Everything is due at 4pm on Fridays.

Week 1 – Course Introduction

1. Boring but important admin stuff
2. **Why are we here?**
3. Introduction to data structures
4. Measuring algorithms – first take
5. Model of computation
6. Algorithm Analysis

A simple problem…

Given: a sequence of n strings $S = \langle s_0, s_1, \ldots, s_{n-1} \rangle$

Problem: Count the number of times each s_i is repeated in the range $[i + 1,\ n - 1]$ for all $i \in [0,\ n - 1]$

S = [hello, hello, world, welcome, hello, welcome]

Algorithm 1

```
read the sequence S of length n
for i=0 to n-1 do
    count = 0
    for j=i+1 to n-1 do
        diff = compare(s[i], s[j])
        if diff == 0 then
            count += 1
    report count
```

Algorithm 1

Works fine, seems fast, easy to implement…

What if we give it more data??

Hmmm…. Ideas?

Algorithm 2

```
read the sequence S of length n
sort(S)
for i=0 to n-1 do
    count = 0
    l = i+1 // start from next element
    if S[i] ≠ S[l]: continue // no match
    // position of the first elem in S[l, n]
    // that is > than S[i]
    u = find_next_geq(S, l, n, S[i])
    report u – l
```

Observation: We can use "sortedness" to our advantage

S = [hello, hello, world, welcome, hello, welcome]

S' = [hello, hello, hello, welcome, welcome, world]

Algorithm 1 vs Algorithm 2: Runtime

Input Size	Algorithm 1	Algorithm 2
n = 10 kB		
n = 100 kB		
n = 1 MB		
n = 10 MB		
n = 100 MB		
N = 1 GB		

Algorithm 1 vs Algorithm 2: Runtime

Input Size	Algorithm 1	Algorithm 2
n = 10 kB	6 millisec	
n = 100 kB	675 millisec	
n = 1 MB	82 sec	
n = 10 MB	> 30 mins	
n = 100 MB	*I'm not*	
N = 1 GB	*running that*	

Algorithm 1 vs Algorithm 2: Runtime

Input Size	Algorithm 1	Algorithm 2
n = 10 kB	6 millisec	1 millisec
n = 100 kB	675 millisec	6 millisec
n = 1 MB	82 sec	86 millisec
n = 10 MB	> 30 mins	1 sec
n = 100 MB	*I'm not*	17 sec
N = 1 GB	*running that*	2 mins

Algorithm 1 vs Algorithm 2: Runtime

Input Size	Algorithm 1	Algorithm 2
n = 10 kB	6 millisec	1 millisec
n = 100 kB	675 millisec	6 millisec
n = 1 MB	82 sec	86 millisec
n = 10 MB	> 30 mins	1 sec
n = 100 MB	*I'm not*	17 sec
N = 1 GB	*running that*	2 mins

Algorithm 3 (magic): 1GB in ??? seconds…

Algorithm 1 vs Algorithm 2: Runtime

Input Size	Algorithm 1	Algorithm 2
n = 10 kB	6 millisec	1 millisec
n = 100 kB	675 millisec	6 millisec
n = 1 MB	82 sec	86 millisec
n = 10 MB	> 30 mins	1 sec
n = 100 MB	*I'm not*	17 sec
N = 1 GB	*running that*	2 mins

Algorithm 3 (magic): 1GB in ~12 seconds…

Algorithm 1 vs Algorithm 2: Runtime

Take home exercise: Can you determine how algorithm 2 is behaving? What is its complexity?

Take home exercise: Can you design a better algorithm? [Algorithm 3?]

What will you get out of today?

1. ~~**Motivation** for the analysis of algorithms~~
2. An **exploration** of possible methods for comparing algorithms and their **shortcomings**
3. A basic model of **computation** and the related **mathematical tools** (analysis techniques) that we will use throughout the remainder of the course
4. You will start believing that maybe, just maybe, algorithms can be fun!

Week 1 – Course Introduction

1. Boring but important admin stuff
2. Why are we here?
3. **Introduction to data structures**
4. Measuring algorithms – first take
5. Model of computation
6. Algorithm Analysis

Algorithms? Problems??

Goal: Solve computational problems with algorithms

A **problem** is a binary relation from **inputs** to **correct outputs**

An **algorithm** is a **procedure** for mapping each **input** to some **output**

Solving a problem

An algorithm *solves* a problem if for all possible inputs, it returns the **correct** output.

In other words, an algorithm is a sequence of computational steps that transform the input to an output.

Often, there are too many inputs to check them all… we often need to reason about correctness – That can be done in terms of mathematical proofs.

Data Structures

What is a **data structure**?

- Data structures define **how** we store and modify non-constant data
- This allows us to support a set of operations on that data
- We think of data structures in terms of *Abstract Data Types*
  - **Different** data structures may implement the **same ADT/API** with **different** performance characteristics

ADTs vs Data Structures

- Abstract Data Types (ADTs, *interfaces, APIs*)
  - Specifies what data we can store
  - What operations we can support and what they mean
- Data structures
  - Tell us **how** to store the data
  - Tell us **how** to support those operations
    - These are algorithms!
- Data structures "implement" ADTs

Static Sequence ADT

Given a list of items X in some order: x_1, x_2, …, x_n

`build(X):`	Make new data structure for items in X
`len(X):`	Return *n*
`get(i):`	Return the element at position *i*
`set(i, x):`	Set x_i to x

The way we store the data and compute those functions depends on the *data structure we use!*

Week 1 – Course Introduction

1. Boring but important admin stuff
2. Why are we here?
3. Introduction to data structures
4. **Measuring algorithms – first take**
5. Model of computation
6. Algorithm Analysis

Algorithm Analysis

How does an algorithm's **runtime** and **memory** usage increase with the **size of the input**?

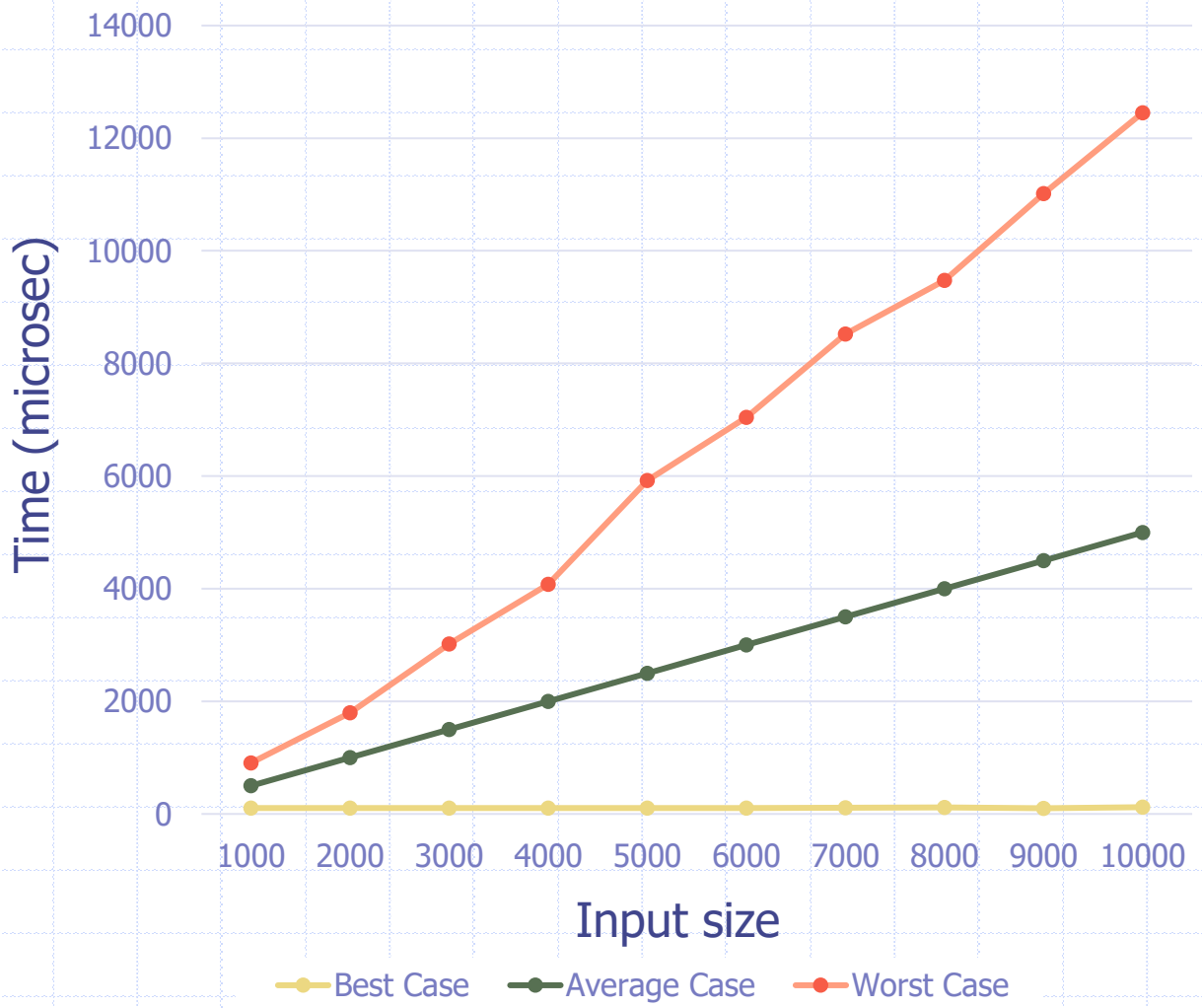
Running Time

Running time of an algorithm typically grows with input size

Average case time is often difficult to determine

We focus on the **worst case** running time:

- easier to analyse
- crucial for many real-world applications



Best, Average, Worst Cases

Given a list of unsorted numbers, *L*, and a specific number, *k*, Return true if *k* is in *L*, or false otherwise.

Consider the following algorithm to solve this problem:

```
public static boolean contains(int[] arr, int k) {
        for (int x : arr) {
                if (x == k) return true;
        }
        return false;
}
```

- What is the **best**, **average**, and **worst** case?

How Can We Analyse Algorithms?

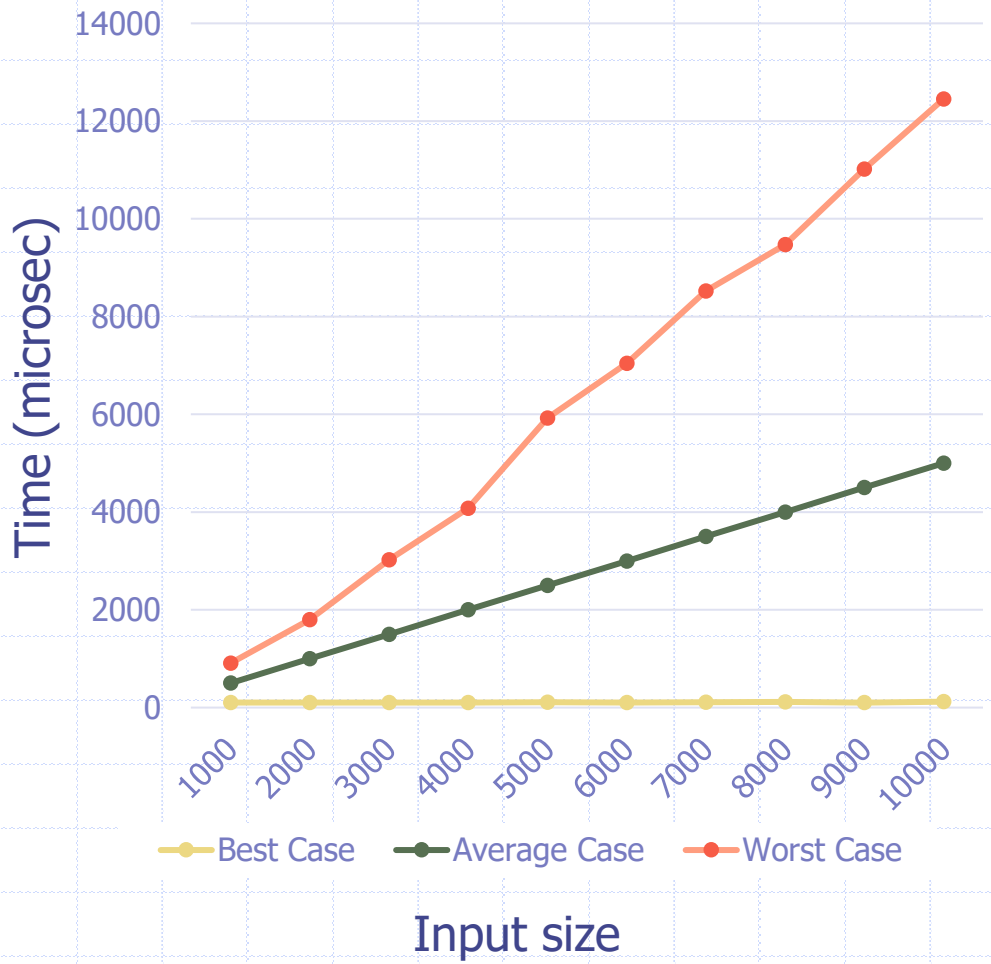
- Experimental (empirical) studies
- Theoretical analysis

Experimental/Empirical Studies

- Write program implementing the algorithm
- Run program with inputs of varying size and composition
- Use a method like `System.nanoTime()` to measure actual time

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

- Plot or tabulate the results



Benefits of Empirical Experiments

- We can readily see how the algorithm behaves with respect to our typical data/use case
- We will observe effects due to hardware, compiler optimizations, constants and coefficients
  - Computers are **magic**
  - Computers are **jerks**
  - **Empirical** analysis allows us to see this!

Limitations of Empirical Experiments ☹

- Need to **implement** the algorithm
  - May be difficult, time consuming, …
- Time may differ based on the **implementation**
- Time may differ based on the **hardware**
- Results may not be indicative of the running time on **other inputs** not included in the experiment
  - How do I know if my inputs are best, average, or worst case?
  - What if the data changes?

Theoretical Analysis ☺

- Use a high-level description of the algorithm
  - instead of an implementation
- Characterise the run-time as a function of input size n
- Takes into account *all possible inputs*
  - Or, at least those that are "bad" (hence, worst-case)
- Evaluation is independent of the hardware and software environment! ☺ Happy days!

Theoretical Analysis Steps

1. **Express algorithm as pseudo-code**
2. Count primitive operations
3. Describe algorithm as f(*n*)
   - function of *n* (the input size)
4. Perform asymptotic analysis
   - express in **asymptotic notation**

Example: Algorithm 1

```
read the sequence S of length n
for i=0 to n-1 do
    count = 0
    for j=i+1 to n-1 do
        diff = compare(s[i], s[j])
        if diff == 0 then
            count += 1
    report count
```

Different Styles… No worries – just be consistent

Algorithm 1: An algorithm with caption

Data: $n \geq 0$
Result: $y = x^n$
$y \leftarrow 1$;
$X \leftarrow x$;
$N \leftarrow n$;
while $N \neq 0$ **do**
  if N *is even* **then**
    $X \leftarrow X \times X$;
    $N \leftarrow \frac{N}{2}$; `/* This is a comment */`
  else
    if N *is odd* **then**
      $y \leftarrow y \times X$;
      $N \leftarrow N - 1$;
    end
  end
end

Algorithm 1 An algorithm with caption

Require: $n \geq 0$
Ensure: $y = x^n$
  $y \leftarrow 1$
  $X \leftarrow x$
  $N \leftarrow n$
  while $N \neq 0$ **do**
    if N is even **then**
      $X \leftarrow X \times X$
      $N \leftarrow \frac{N}{2}$ ▷ This is a comment
    else if N is odd **then**
      $y \leftarrow y \times X$
      $N \leftarrow N - 1$
    end if
  end while

https://www.overleaf.com/learn/latex/Algorithms

Theoretical Analysis Steps

1. Express algorithm as pseudo-code
2. **Count primitive operations**
3. Describe algorithm as f(*n*)
   - function of *n* (the input size)
4. Perform asymptotic analysis
   - express in **asymptotic notation**

But what is an "operation"??

- In this course, we will assume the **Word-RAM** model
  - A word is a sequence of w bits (most of our computers have $w = 64$ bits)
  - Most modern computers are ***byte*** *addressable*
    - The need to be able to access every memory "cell" limits memory size
    - $w \geq$ #bits required to represent the largest memory address

But what is an "operation"??

- In this course, we will assume the **Word-RAM** model
  - Basic arithmetic operations take a **single operation**
    - + - % * // and so on
  - Bitwise operands take a **single operation**
    - &, |, <<, >>, …
  - Comparisons take a **single operation**
    - >, <, ==, !=
  - Accessing or writing a word in memory takes a **single operation**
  - Assignment of a variable takes a **single operation**

Counting Primitive Operations

num = 10	1 operation
num = *A*[10]	2 operations
if *num* < 10:	1 operation
if *num* < A[10]:	??
while *num* < 10:	1 operation **per iteration** + 1

Counting Primitive Operations

Loops

```
for (int i=0; i < n; i++)
{
   ... let counted operations = X
} // count increment
```

$1 + (n + 1)$

$n \cdot X$

n

- "`int i=0`" is executed once
- "`i < n`" conditional is evaluated $n + 1$ times
- "`i++`" increment is evaluated n times
- X instructions within loop are evaluated n times

Theoretical Analysis Steps

1. Express algorithm as pseudo-code
2. Count primitive operations
3. **Describe algorithm as f(*n*)**
   - function of *n* (the input size)
4. Perform asymptotic analysis
   - express in **asymptotic notation**

Describe as f(n) – Function of n

By inspecting the pseudo-code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size!

```
Algorithm arrayMax(A, n)                # operations
  currentMax ← A[0]                     2
  i ← 1                                 1
  while i < n do                        n
    if A[i] > currentMax then           2 • (n – 1)
      currentMax ← A[i]                 2 • (n – 1)
    i ← i  + 1                          n – 1
  return currentMax                     1
Total:                                  6 • n + lower order terms
```

Estimating Running Time

$\texttt{arrayMax}$ executes $6n$ primitive operations, worst case

- Let a be the time taken by the fastest primitive operation
- Let b be the time taken by the slowest primitive operation

Let $T(n)$ be the worst-case time of $\texttt{arrayMax}$

- $a \cdot 6n \leq T(n) \leq b \cdot 6n$

Run time $T(n)$, is bounded by two **linear** functions

- This tells us something _VERY_ useful about our worst-case behaviour. Let's into this idea in more detail...

Theoretical Analysis Steps

1. Express algorithm as pseudo-code
2. Count primitive operations
3. Describe algorithm as f(n)
   - function of n (the input size)
4. **Perform asymptotic analysis**
   - express in **asymptotic notation**

Seven Important Functions

- Seven functions that often appear in algorithm analysis
  - Constant $\approx$ 1
  - Logarithmic $\approx$ $\log_2 n$
  - Linear $\approx$ n
  - N-Log-N $\approx$ $n \log_2 n$
  - Quadratic $\approx$ n^2
  - Cubic $\approx$ n^3
  - Exponential $\approx$ 2^n
  - Factorial $\approx$ $n!$

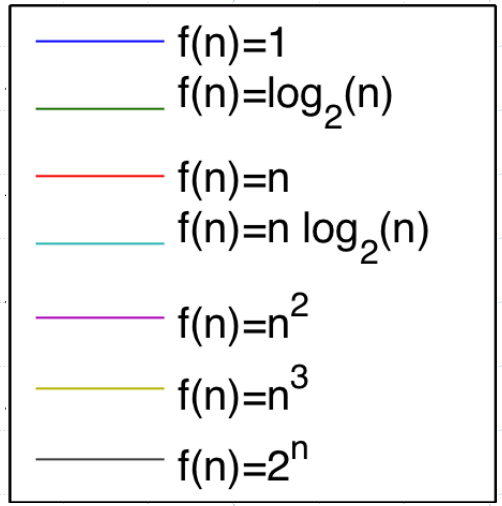
Seven Important Functions

- ~~Seven~~ functions that often appear in algorithm analysis
  - Constant $\approx$ 1
  - Logarithmic $\approx$ $\log_2 n$
  - Linear $\approx$ n
  - N-Log-N $\approx$ $n \log_2 n$
  - Quadratic $\approx$ n^2
  - Cubic $\approx$ n^3
  - Exponential $\approx$ 2^n
  - Factorial $\approx$ $n!$

That's not seven, that's eight!

Stay sharp…

Exercise: Draw these!

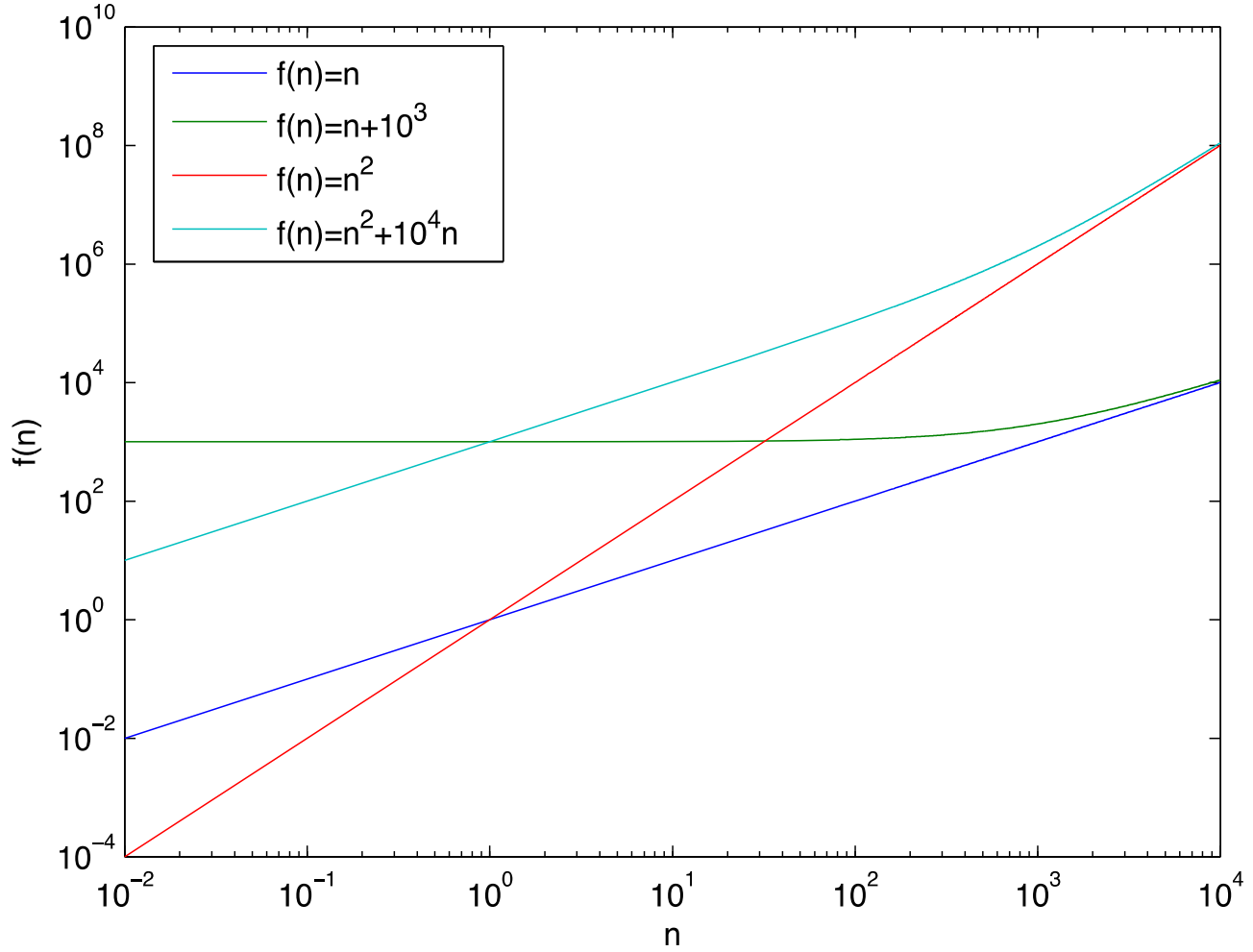


Analysis Process

To perform an asymptotic analysis of the worst-case running time of an algorithm

1. find the worst-case number of primitive operations executed as a function of the input size – f(n)
   - since constant factors and lower-order terms do not affect the growth rate for large n they are usually **disregarded** when counting primitive operations
2. express this function with big-O notation

Why don’t lower order terms matter?



Big-O Notation

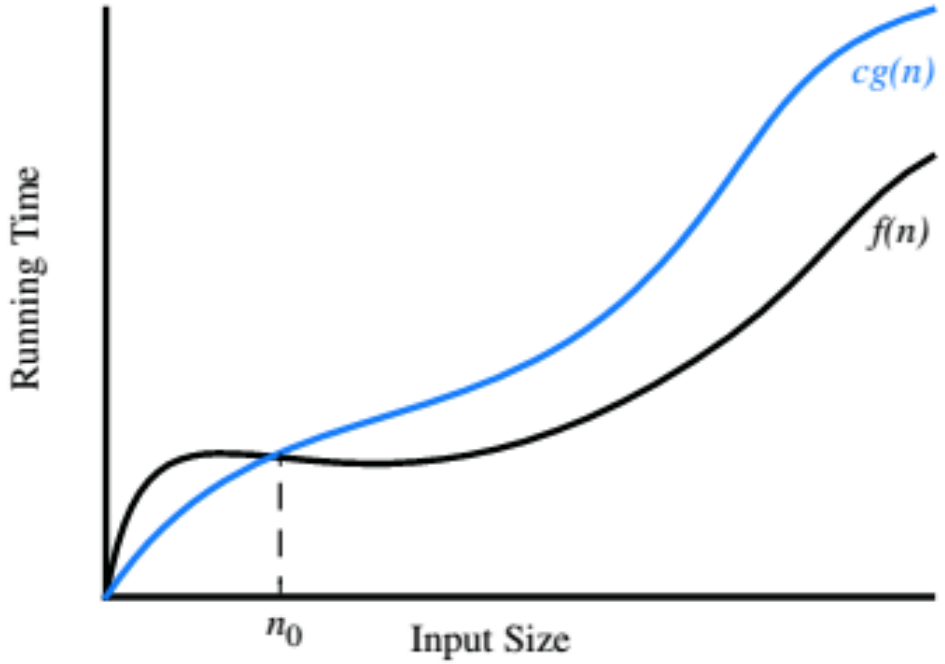
Big-O notation describes an **upper bound** on a **function**

$f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less** than or equal to $g(n)$

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \text{ for } n \geq n_0$$

Illustrating Big-O Notation



- $f(n)$ is $O(g(n))$, since
- $f(n) \leq c \cdot g(n)$ when $n \geq n_0$

Big-O Example

Show that $2n + 10$ is $O(n)$

$f(n) = 2n + 10$

$g(n) = n$

$2n + 10 \leq c \cdot n$

$10 \leq c \cdot n - 2n$

$c \cdot n - 2n \geq 10$

$(c - 2)\, n \geq 10$

$n \geq 10 \div (c - 2)$

Pick $c = 3$ and $n_0 = 10$

$f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

Consider Big-O of *arrayMax*

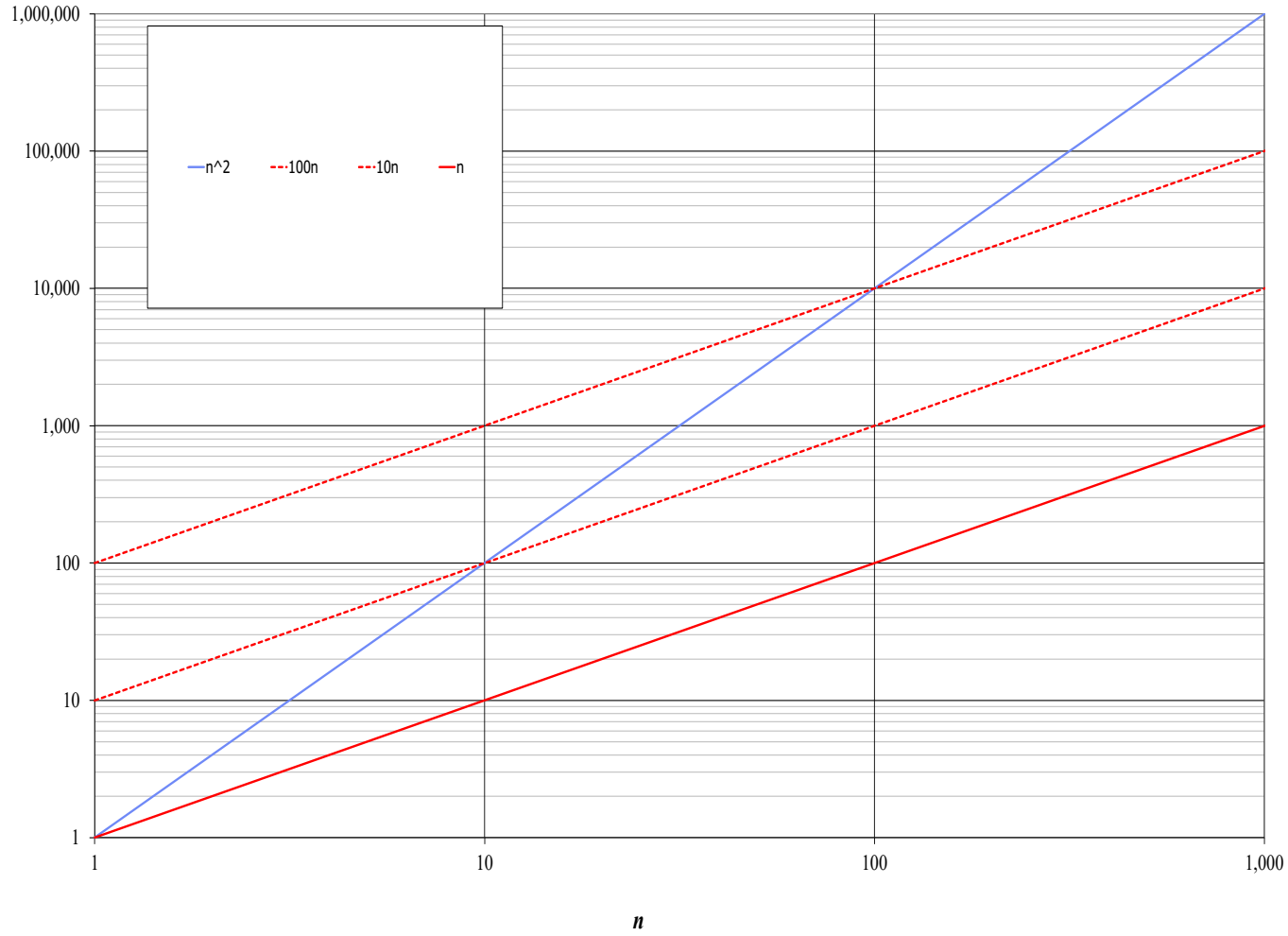
- *arrayMax* executes at most $6n$ primitive operations
  - constant factor "6" can be disregarded from $6n$
  - lower order terms can be disregarded

Thus, we say that algorithm *arrayMax*
"runs in $O(n)$ time" (or, that it runs in "linear time")

Big-O Example

Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \geq n_0$
- Definition cannot be satisfied since c is a constant



Big-O and Growth Rate

> Big-O notation gives an *upper bound* on the *growth rate* of a function

> "$f(n)$ is $O(g(n))$" means the growth rate of $f(n)$ is no more than the growth rate of $g(n)$

> Big-O notation ranks functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-O Simplification Rules

- **Rule 1:** If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$

  > We can drop lower-order terms

  > We can drop constant factors (coefficients)

  eg. $3n^4 + 7n^3 + 5$ is $O(n^4)$

Big-O Simplification Rules

- Can we say that "$2n$ is $O(n^2)$" ?
  - yes – remember big-O **is just an upper bound**
  - but this doesn't give us much information about the function's growth

- **Rule 2:** Use the smallest possible class of functions (the "tightest" possible bound) when describing a function

  > "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$", even if the latter is still mathematically correct…

  > Quiz: Is "$8n$ is $O(n^3)$"?

Big-O Simplification Rules

- **Rule 3:** Use the simplest expression of the class
  - "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

The generic plan to describe a function f in terms of a class of functions is as follows

1. Drop lower order terms from f
2. Drop constant factors from f

Asymptotic Analysis Recap

- What is the big-O running time of this algorithm?

```
Algorithm insertionSort(A, n)
    for i ← 1 to n - 1 do
        key ← A[i]                      } O(1)
        j ← i - 1
        while j >= 0 and A[j] > key do  } O(n) worst-case, O(1) best-case
            A[j + 1] ← A[j]             } O(1)
            j ← j – 1
        A[j + 1] ← key
```

n iterations

- Worst-case (array is in descending order): $O(n^2)$
- Best-case (array is already sorted): $O(n)$

Example

- Give an asymptotic upper bound (in big-O notation) for
  - $\log 8n + \log (n^{2n})$
  - $\log 8n + 2n \log n$
  - $\log n + n \log n$
  - $O(n \log n)$

3. $\log_b a^c = c \log_b a$

(drop constants)

(drop lower order terms)

Let's examine these classes more closely...

Remember: We are describing the **behaviour** of the **algorithm** in terms of the size of the **input, *n***

O(1) Constant Woohoo!

Adapted from Alistair Moffat with permission

O(1)	Constant	Woohoo!
O(log *n*)	Logarithmic	Superb!

Adapted from Alistair Moffat with permission

O(1)	Constant	Woohoo!
O(log *n*)	Logarithmic	Superb!
O(*n*)	Linear	Good!

Adapted from Alistair Moffat with permission

O(1)	Constant	Woohoo!
O(log *n*)	Logarithmic	Superb!
O(*n*)	Linear	Good!
O(*n* log *n*)	Super linear	Nice!

Adapted from Alistair Moffat with permission

O(1)	Constant	Woohoo!
O(log *n*)	Logarithmic	Superb!
O(*n*)	Linear	Good!
O(*n* log *n*)	Super linear	Nice!
O(n^2)	Quadratic	Fair...

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!
$O(\log n)$	Logarithmic	Superb!
$O(n)$	Linear	Good!
$O(n \log n)$	Super linear	Nice!
$O(n^2)$	Quadratic	Fair...
$O(n^3)$	Cubic	Tolerable

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!
$O(\log n)$	Logarithmic	Superb!
$O(n)$	Linear	Good!
$O(n \log n)$	Super linear	Nice!
$O(n^2)$	Quadratic	Fair...
$O(n^3)$	Cubic	Tolerable
$O(2^n)$	Exponential	Intractable

Adapted from Alistair Moffat with permission

If I was given 10^{10} loop iterations
= 10000000000

(10 billion – Maybe 10-30 seconds worth of computation, depends on the language/compiler/computer/…)

What value of *n* could we reach?

$O(1)$	Constant	Woohoo!
$O(\log n)$	Logarithmic	Superb!
$O(n)$	Linear	Good!
$O(n \log n)$	Super linear	Nice!
$O(n^2)$	Quadratic	Fair...
$O(n^3)$	Cubic	Tolerable
$O(2^n)$	Exponential	Intractable

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!
$O(\log n)$	Logarithmic	Superb!
$O(n)$	Linear	Good!
$O(n \log n)$	Super linear	Nice!
$O(n^2)$	Quadratic	Fair...
$O(n^3)$	Cubic	Tolerable
$O(2^n)$	Exponential	(cooked)

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!
$O(\log n)$	Logarithmic	Superb!
$O(n)$	Linear	Good!
$O(n \log n)$	Super linear	Nice!
$O(n^2)$	Quadratic	Fair...
$O(n^3)$	Cubic	Tolerable
$O(2^n)$	Exponential	Intractable

Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	
$O(\log n)$	Logarithmic	Superb!	
$O(n)$	Linear	Good!	
$O(n \log n)$	Super linear	Nice!	
$O(n^2)$	Quadratic	Fair...	
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	
$O(\log n)$	Logarithmic	Superb!	
$O(n)$	Linear	Good!	
$O(n \log n)$	Super linear	Nice!	
$O(n^2)$	Quadratic	Fair...	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	
$O(\log n)$	Logarithmic	Superb!	
$O(n)$	Linear	Good!	
$O(n \log n)$	Super linear	Nice!	Large, $10^{8.5}$
$O(n^2)$	Quadratic	Fair...	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	
$O(\log n)$	Logarithmic	Superb!	
$O(n)$	Linear	Good!	Huge, 10^{10}
$O(n \log n)$	Super linear	Nice!	Large, $10^{8.5}$
$O(n^2)$	Quadratic	Fair...	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	
$O(\log n)$	Logarithmic	Superb!	Astronomical
$O(n)$	Linear	Good!	Huge, 10^{10}
$O(n \log n)$	Super linear	Nice!	Large, $10^{8.5}$
$O(n^2)$	Quadratic	Fair…	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

$O(1)$	Constant	Woohoo!	To infinity
$O(\log n)$	Logarithmic	Superb!	Astronomical
$O(n)$	Linear	Good!	Huge, 10^{10}
$O(n \log n)$	Super linear	Nice!	Large, $10^{8.5}$
$O(n^2)$	Quadratic	Fair…	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

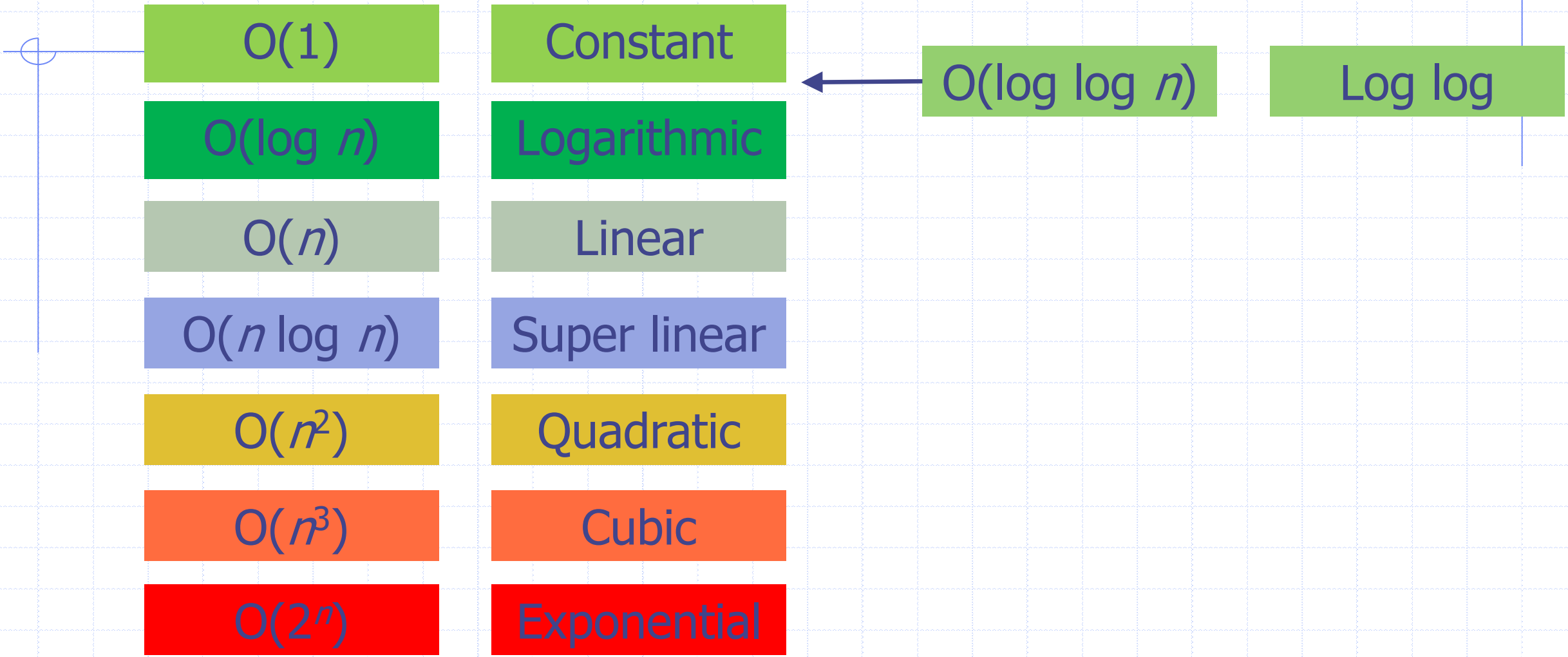
$O(1)$	Constant	Woohoo!	AND BEYOND!!
$O(\log n)$	Logarithmic	Superb!	Astronomical
$O(n)$	Linear	Good!	Huge, 10^{10}
$O(n \log n)$	Super linear	Nice!	Large, $10^{8.5}$
$O(n^2)$	Quadratic	Fair...	Biggish, 10^5
$O(n^3)$	Cubic	Tolerable	Modest, 2150
$O(2^n)$	Exponential	Intractable	Tiny, 33

Adapted from Alistair Moffat with permission

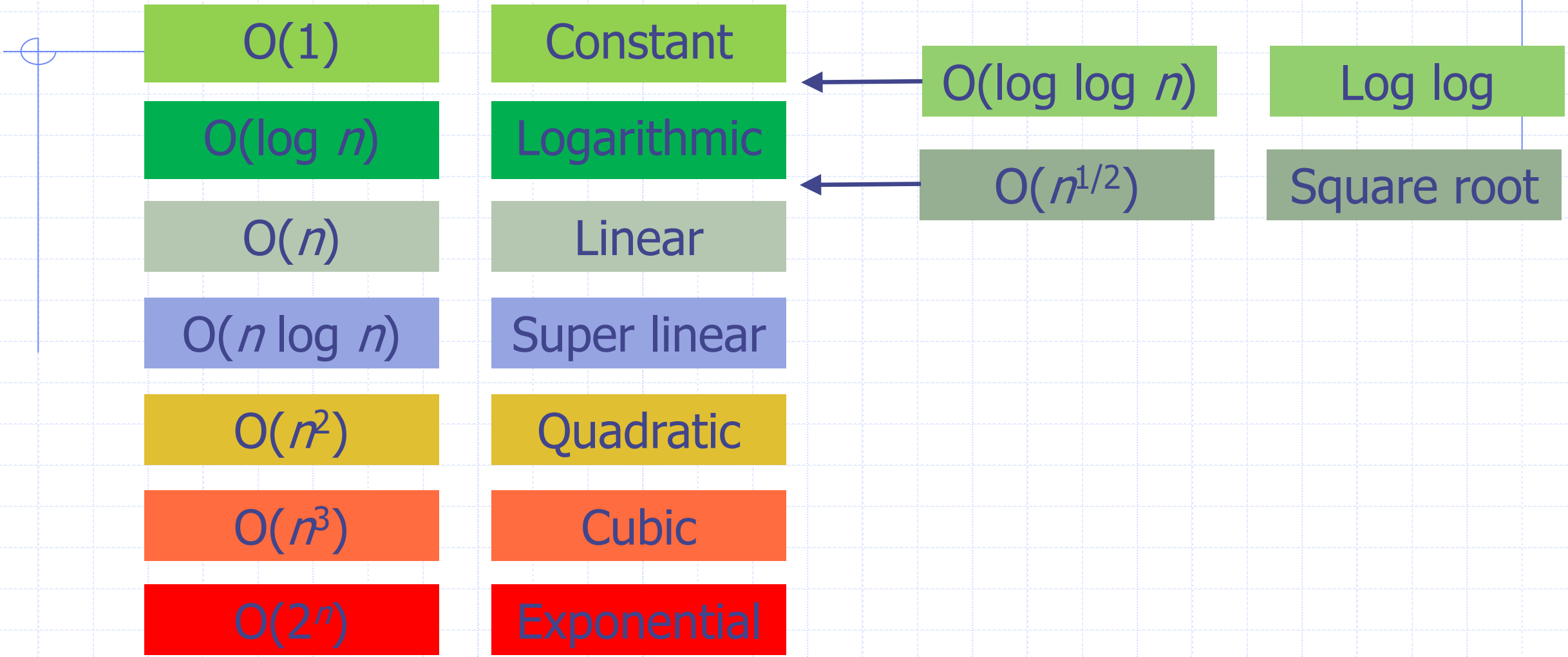
Cool! But what about other functions?

Cool! But what about other functions?

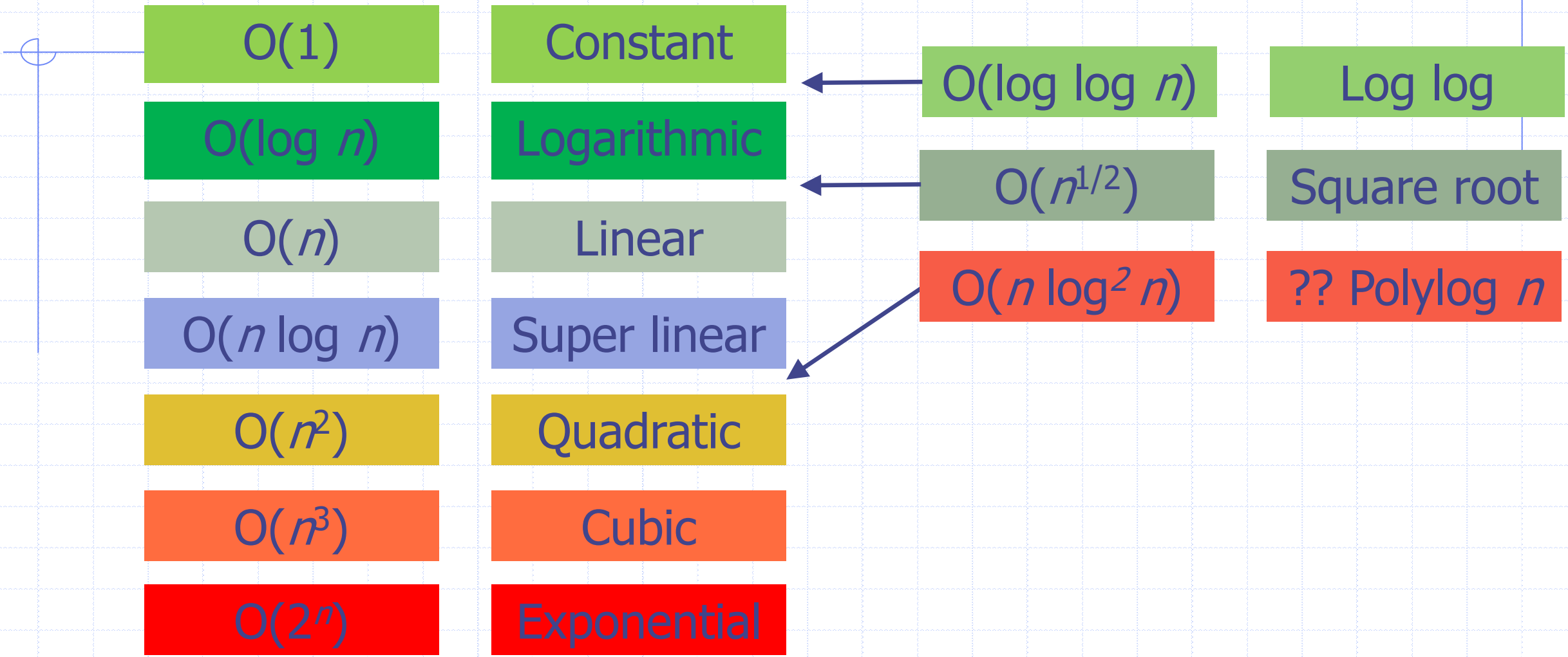
We can use as many as we want!



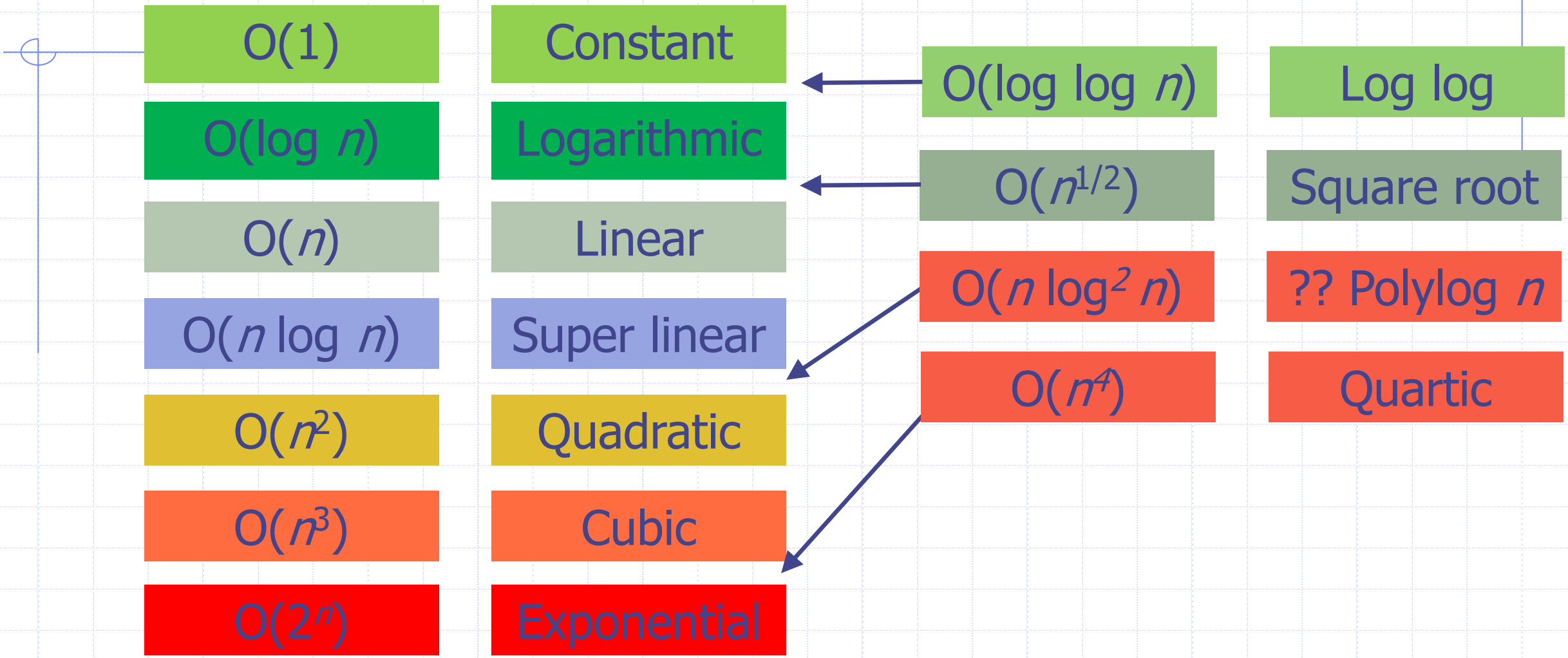
Adapted from Alistair Moffat with permission



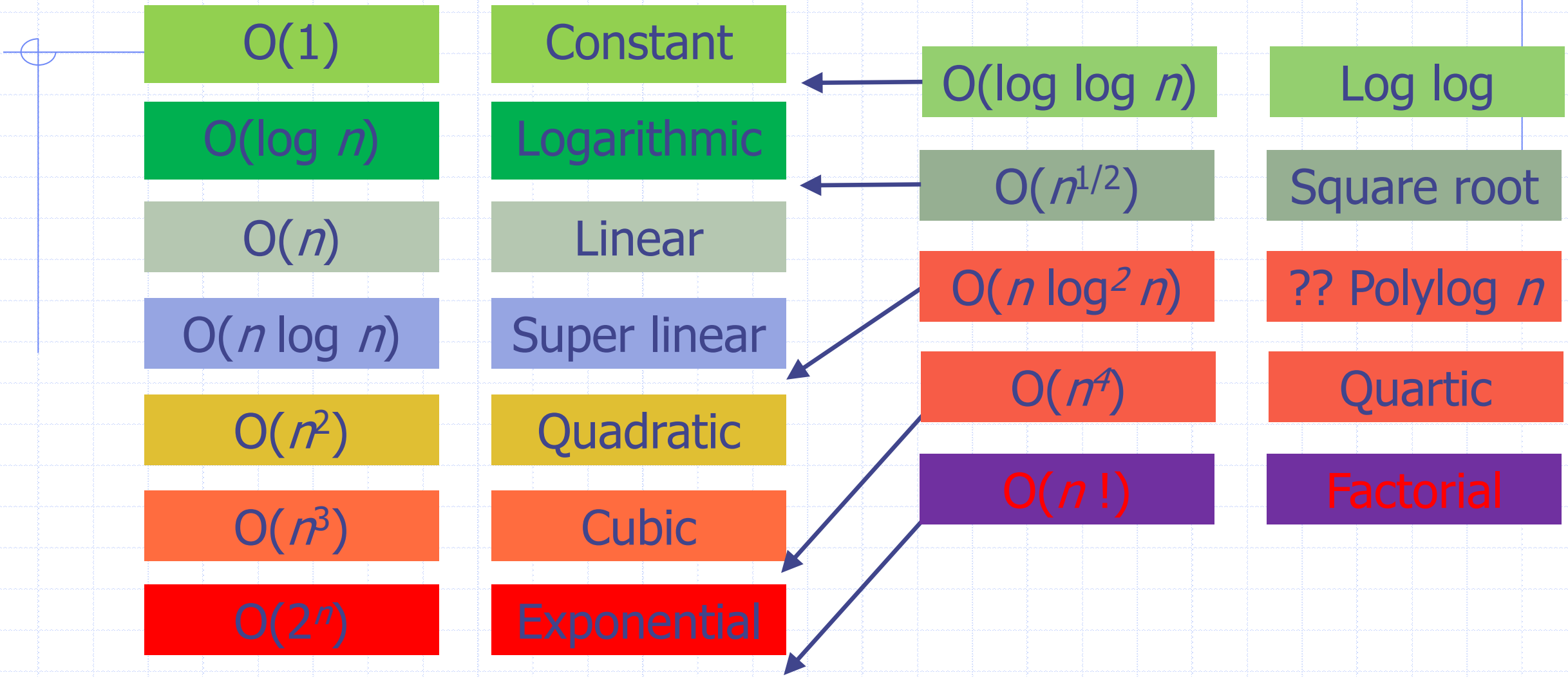
Adapted from Alistair Moffat with permission



Adapted from Alistair Moffat with permission



Adapted from Alistair Moffat with permission



Adapted from Alistair Moffat with permission

Big-**Omega** Notation

Big-Ω notation describes a **lower bound** on a **function**

$f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater** than or equal to $g(n)$

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \geq c \cdot g(n) \text{ for } n \geq n_0$$

Big-O Notation (repeated to compare)

Big-O notation describes an **upper bound** on a **function**

$f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less** than or equal to $g(n)$

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \text{ for } n \geq n_0$$

Big-**Theta** Notation

Big-Θ notation describes a **tight bound** on a **function** (if one exists)

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal to** $g(n)$

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ if there are positive constants c_1, c_2 and n_0 such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for } n \geq n_0$$

By definition, if f(n) is in O(g(n)) **and** f(n) is in Ω(g(n)), then f(n) must be Θ(g(n)) – Exercise: Think about why this is the case.

Big-O/Omega/Theta vs “best, worst, average”

Extremely important: Just because Big-O means “upper bound”, it is **not the same** thing as **worst case**

> Similarly for Theta and Average, or Omega and Best

“Best, Average, Worst” are **CASES**

“O, Omega, Theta” are **BOUNDS**

Big-O/Omega/Theta vs "best, worst, average"

Example: Finding some element k in an unsorted list L containing n items

- Best case: k is the first element
- Worst case: k is not in the list

The worst case is clearly O(n), and it is also Ω(n), so it is thus Θ(n) in the worst case.

The best case is clearly O(1) – how about Ω?

Final word: How scary is **exponential** growth?

Suppose I asked to be paid 1 billionth of a cent for the first student that passed this course...

Suppose I asked to be paid 1 billionth of a cent for the first student that passed this course…

And 2 billionths of a cent for the second…

Suppose I asked to be paid 1 billionth of a cent for the first student that passed this course…

And 2 billionths of a cent for the second…

And 4 billionths of a cent for the third…

Suppose I asked to be paid 1 billionth of a cent for the first student that passed this course…

And 2 billionths of a cent for the second…

And 4 billionths of a cent for the third…

Would you accept this contract?

Suppose I pass 30 of you ☺

That's 2^{30} billionths of a cent

Suppose I pass 30 of you ☺

That's 2^{30} billionths of a cent

Or about 1-2 cents ☹

Suppose I pass 40 of you ☺

That's 2^{40} billionths of a cent

Suppose I pass 40 of you ☺

That's 2^{40} billionths of a cent

Or about $11 ☹

Suppose I pass 100 of you ☺

That's 2^{100} billionths of a cent

Suppose I pass 100 of you ☺

That's 2^{100} billionths of a cent

Or about $12,676,506,002,282,294,014

Suppose I pass everyone…

That's 2^{600} billionths of a cent

$\$4.15 \times 10^{169}$

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

From "The Algorithm Design Manual" - Skiena

What's Next?

- Readings…
  - Data Structures and Algorithms in Python
    - Chapter 3.1 (the rest of 3 is also interesting and useful)
    - Chapter 4
  - Introduction to Algorithms
    - Chapter 2.1, 2.2
  - Keep an eye out for Problem Set One, Assignment One
  - Get your (Java) environment set up

Appendix

- These remaining slides contain some useful information that is not covered in the lecture.

Pseudo-Code Details

- **Control flow**
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- **Method declaration**
  - **Algorithm** ***method*** (***arg*** [, ***arg***…])
    - **Input** …
    - **Output** …
- **Method calls**
  - ***method*** (***arg*** [, ***arg***…])
- **Return value**
  - **return** ***expression***
- **Mathematical formatting**
  - $X \times Y + N^2$
- **Expressions**
  - ← *(Assignment)*
    - Like "=" in Python
  - = *(Equality testing)*
    - Like "==" in Python

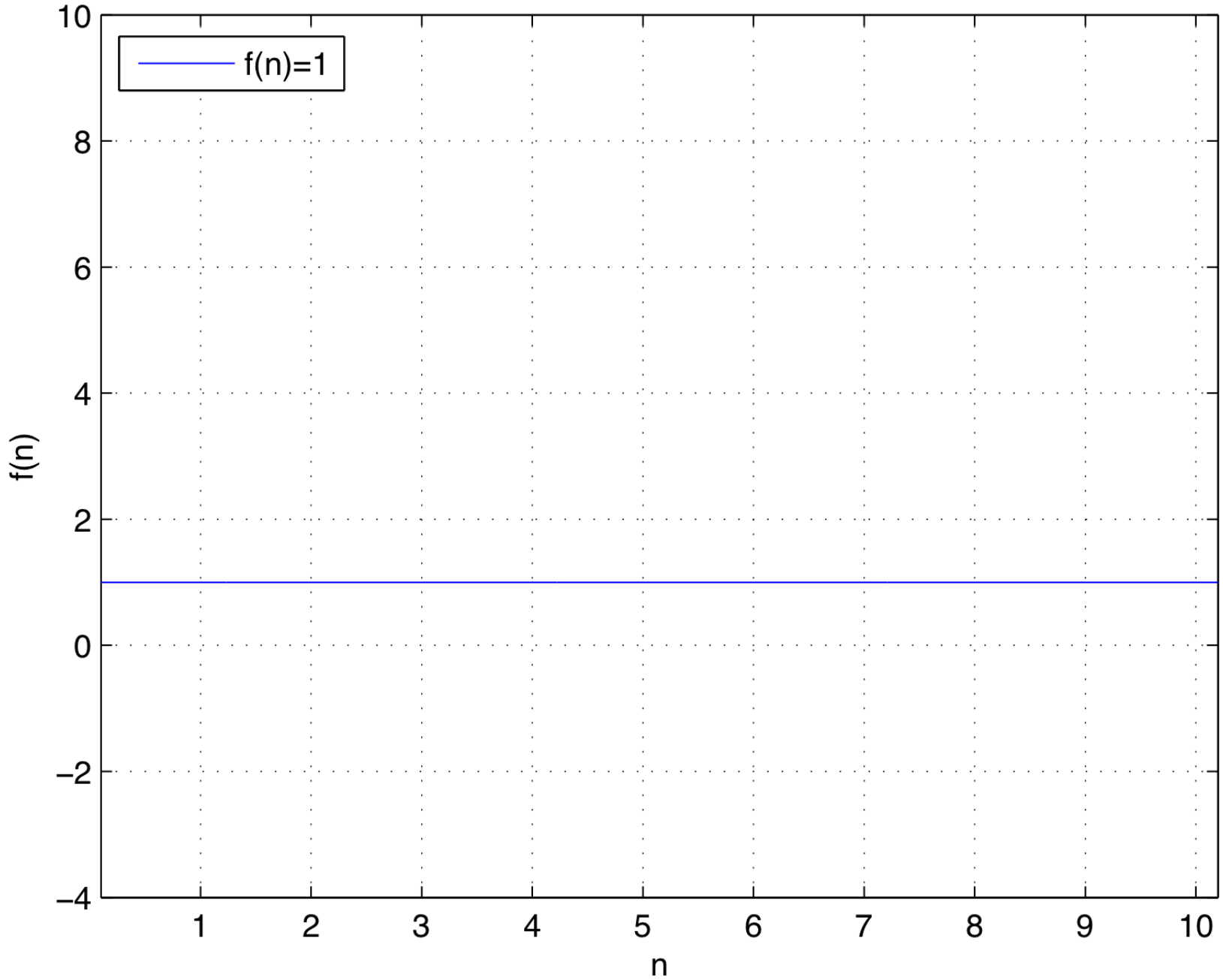
Pseudo-code vs real code

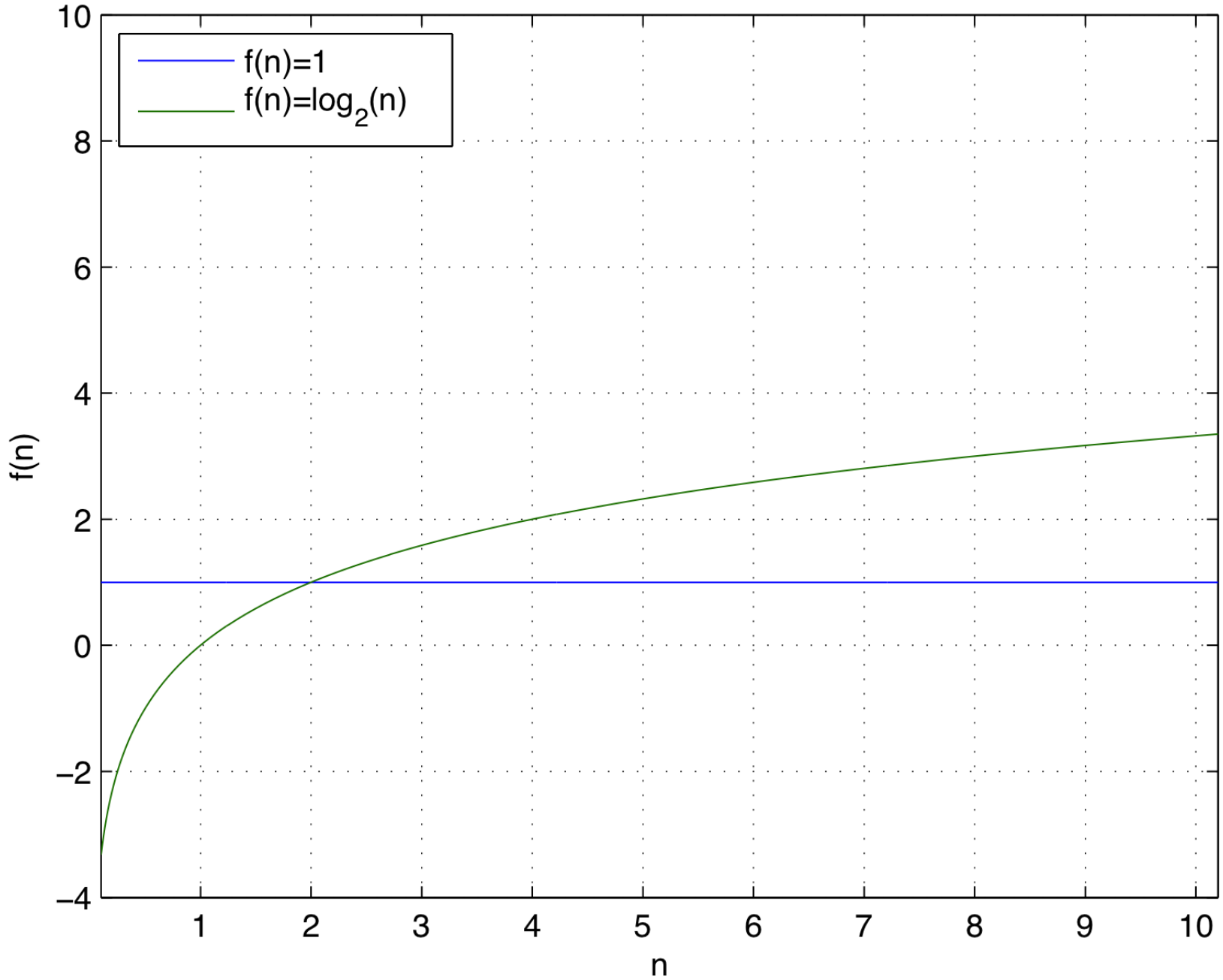
```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    i ← 1
    while i  < n  do
        if A[i] > currentMax then
            currentMax ← A[i]
        i ← i  + 1
    return currentMax
```

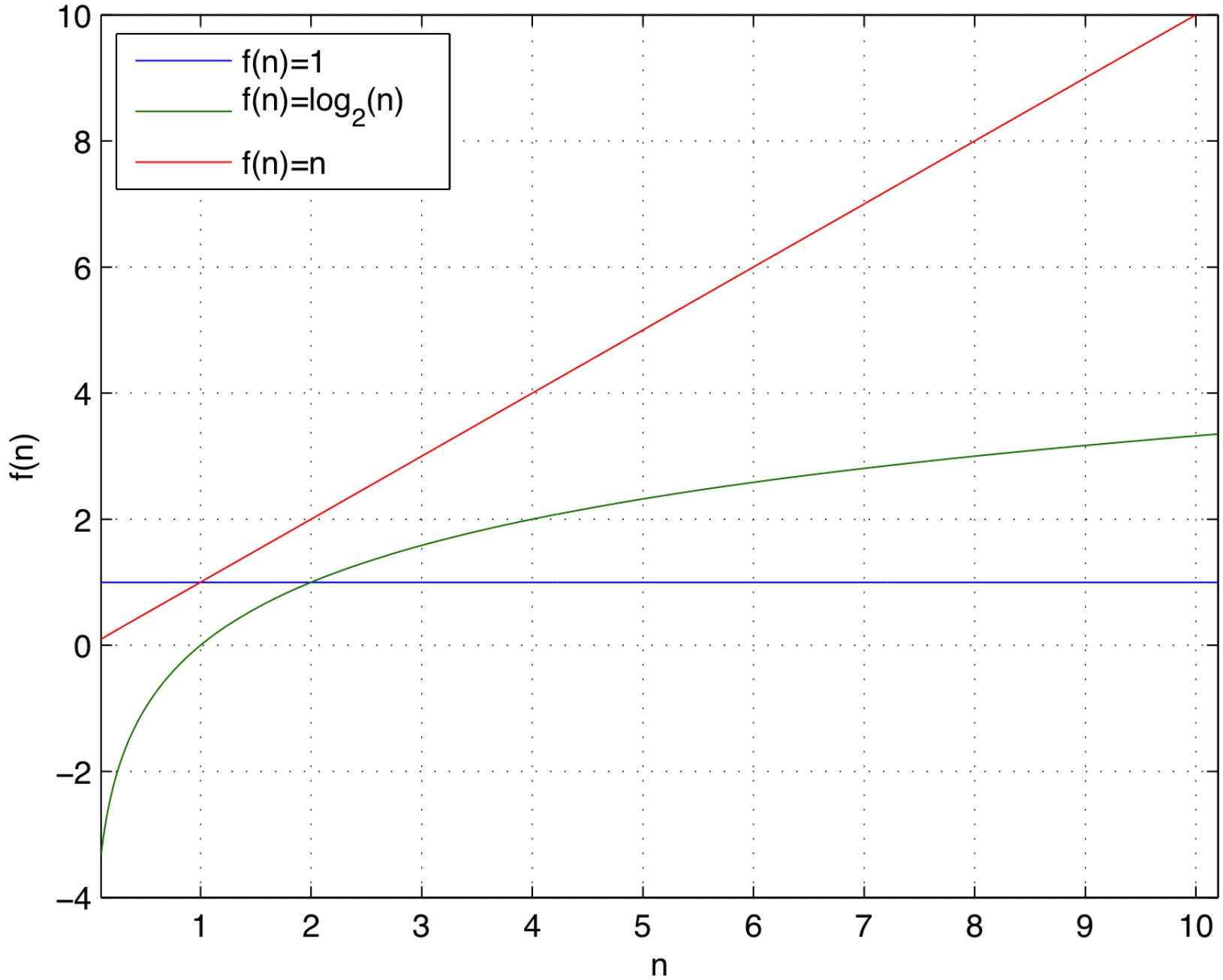
Pseudo-code

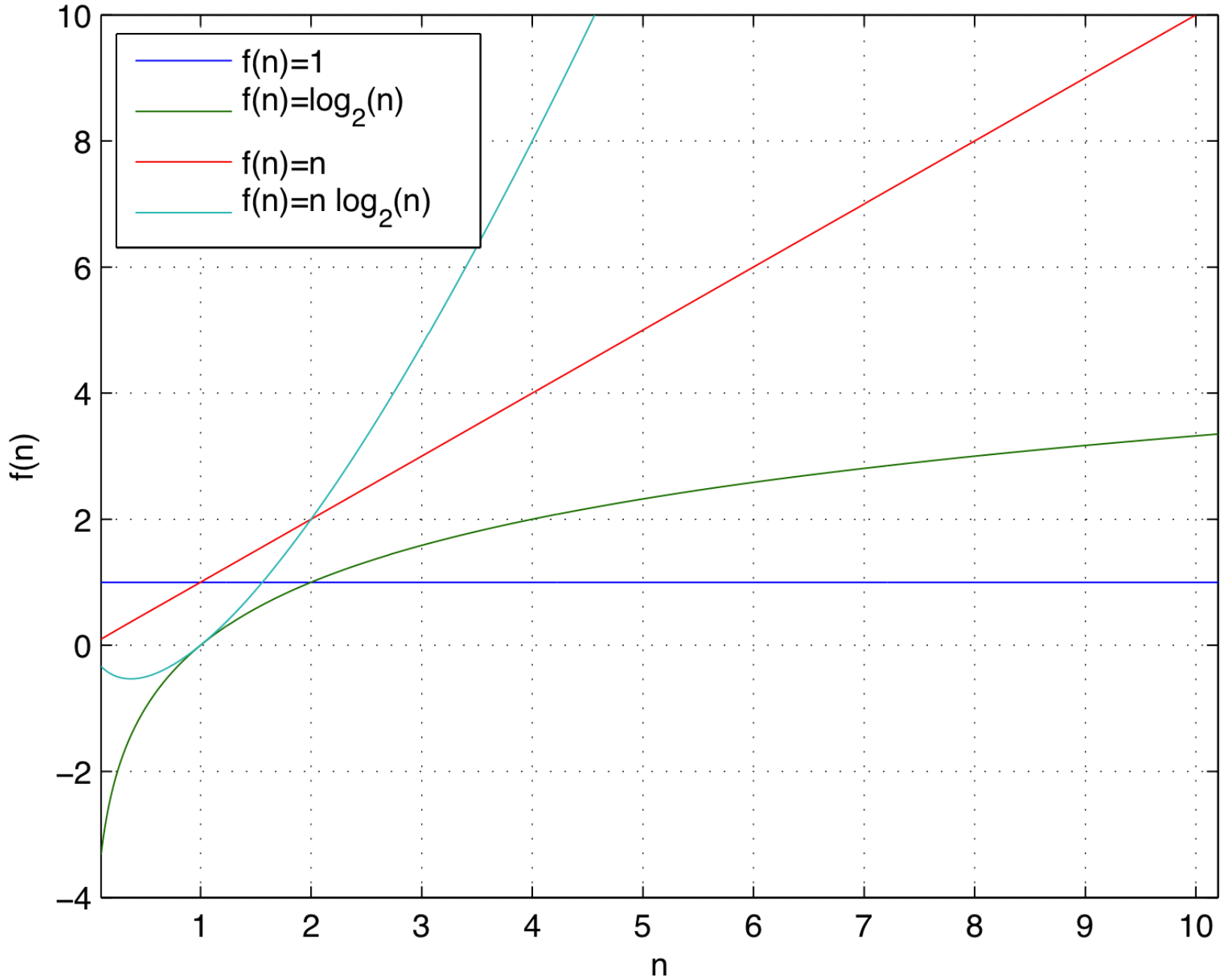
```
def array_max(my_array):
    current_max = my_array[0]
    for element in my_array:
        if element > current_max:
            current_max = element
    return current_max
```

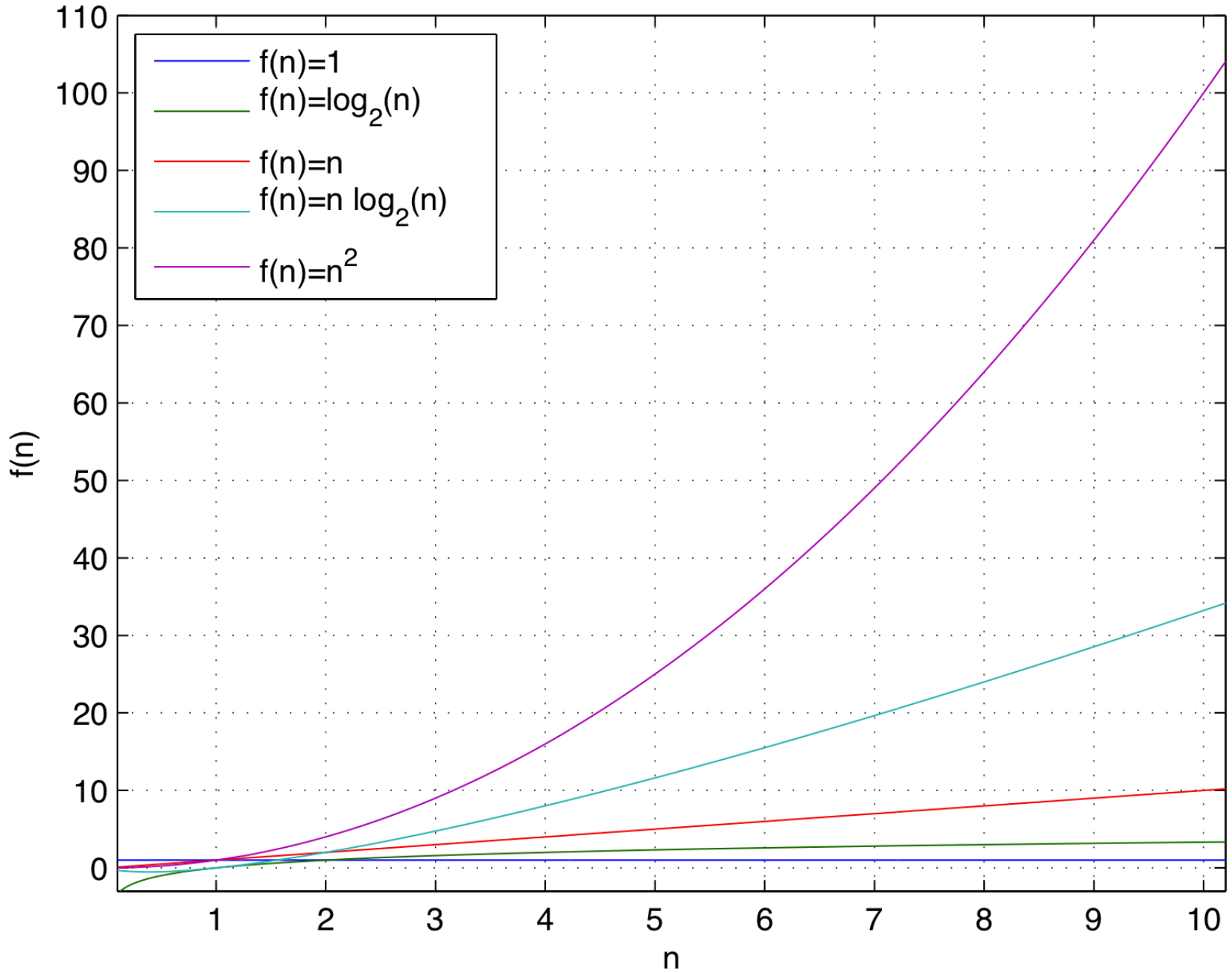
Python code

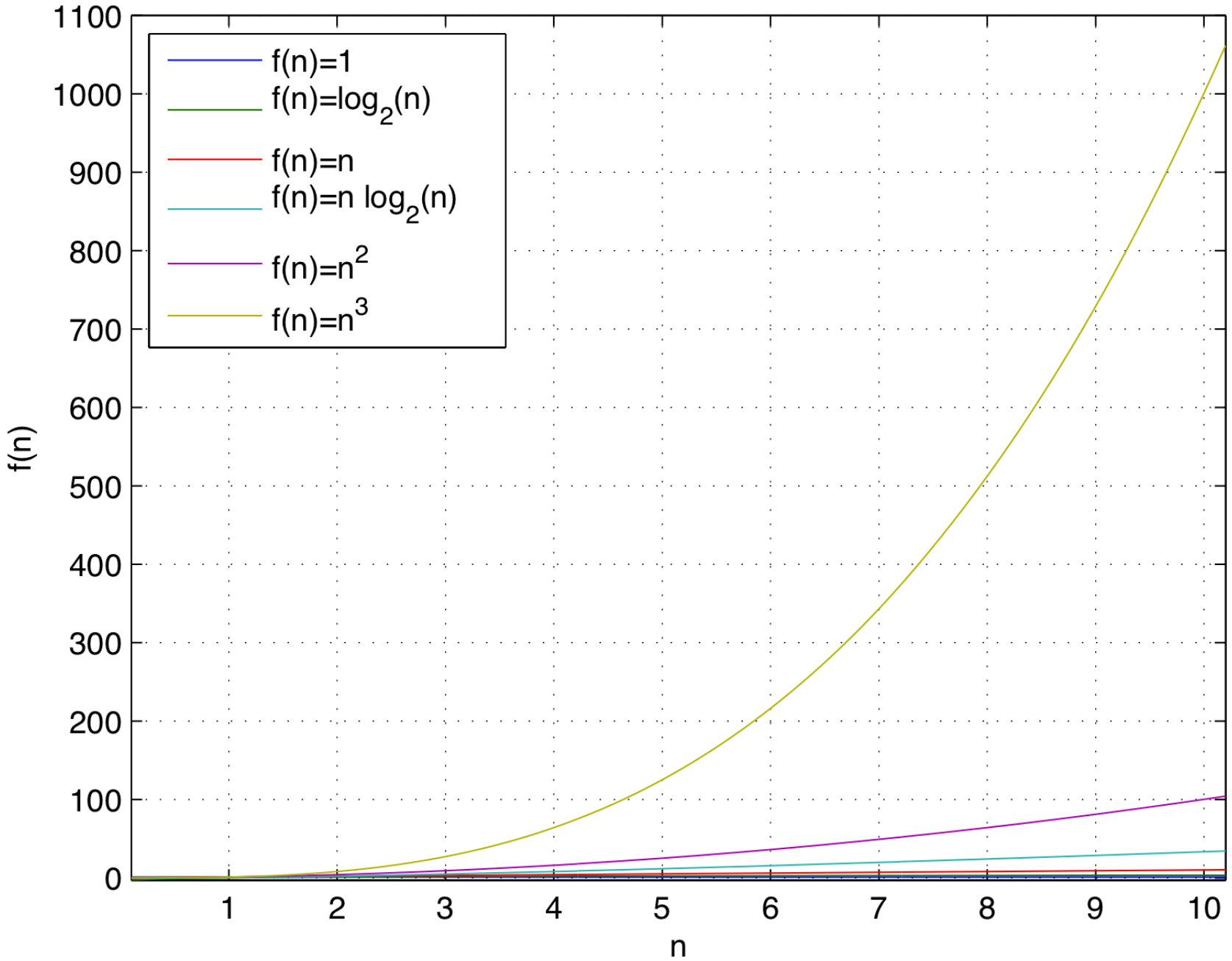

f(n)=1
10
8
6
4
2
0
−2
−4
f(n)
1
2
3
4
5
6
7
8
9
10
n

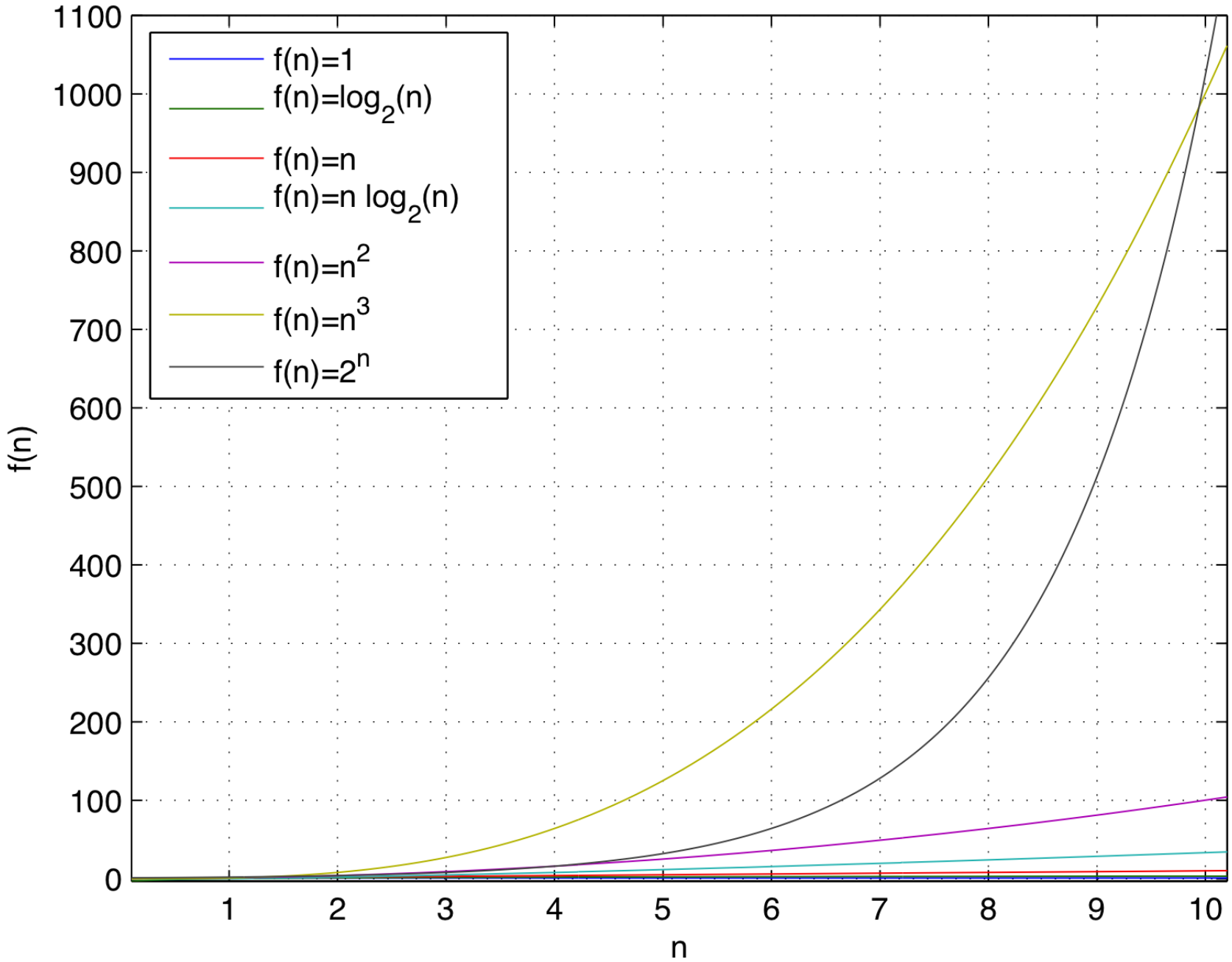

f(n)=1
f(n)=log2(n)
10
8
6
4
2
0
−2
−4
f(n)
1
2
3
4
5
6
7
8
9
10
n


f(n)=1
f(n)=log2(n)
f(n)=n
10
8
6
4
2
0
−2
−4
f(n)
1
2
3
4
5
6
7
8
9
10
n


f(n)=1
f(n)=log2(n)
f(n)=n
f(n)=n log2(n)
f(n)
n
10
8
6
4
2
0
−2
−4
1
2
3
4
5
6
7
8
9
10


f(n)=1
f(n)=log2(n)
f(n)=n
f(n)=n log2(n)
f(n)=n2
f(n)
n
0
10
20
30
40
50
60
70
80
90
100
110
1
2
3
4
5
6
7
8
9
10


f(n)=1
f(n)=log2(n)
f(n)=n
f(n)=n log2(n)
f(n)=n2
f(n)=n3
f(n)
n
0
100
200
300
400
500
600
700
800
900
1000
1100
1
2
3
4
5
6
7
8
9
10


f(n)=1
f(n)=log2(n)
f(n)=n
f(n)=n log2(n)
f(n)=n2
f(n)=n3
f(n)=2n
f(n)
n
0
100
200
300
400
500
600
700
800
900
1000
1100
1
2
3
4
5
6
7
8
9
10

Scary Effects of Growth Rate

```
n ← 64
k ← 1

for i ← 0 to n-1 do
   for j ← 0 to k-1 do
      pick()
      { increment counter j }
   k ← k * 2
   { increment counter i }
```

- Suppose "pick" takes 10^{-9}s to execute
- This **harmless looking loop** would take 585 years!!! to run
  - assuming $82 \cdot 10^9$ instructions per sec
- Would still take 7 years if "pick" takes a single instruction

More Big-O Examples

Plan for finding g in "$f(n)$ is $O(g(n))$"
Drop lower-order terms from f
Drop constant factors from f
Tentatively:
- find "smallest" class of functions
- use "simplest" expression of class

$7n - 2$

$7n - 2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$

true for $c = 7$ and $n_0 = 1$

$3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$

true for $c = 4$ and $n_0 = 21$

$3 \log n + 5$

$3 \log n + 5$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$

true for $c = 8$ and $n_0 = 2$

Using Relatives of Big-O

$5n^2$ is $\Omega(n^2)$

let $c = 1$ and $n_0 = 1$

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

$5n^2$ is $O(n^2)$

let $c = 5$ and $n_0 = 1$

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

$5n^2$ is $\Theta(n^2)$

$f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$.

Let $c = 5$ and $n_0 = 1$

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$

$5 \cdot n2 \leq 5n2 \leq 5 \cdot n2$ for all $n \geq 1$