

# Analysing Algorithms (Part I - Complexity notation)

Daniel Lawson — University of Bristol

Lecture 08.1.1 (v1.0.1)

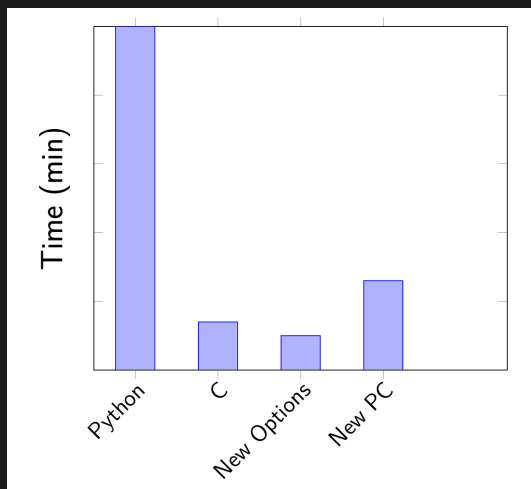
# Signposting

- ▶ This set of lectures is about the conceptual framework for algorithms.
- ▶ Analysing Algorithms is split into three parts:
  - ▶ Part 1: Motivation and Algorithmic Complexity
  - ▶ Part 2: Examining algorithms
  - ▶ Part 3: Turing Machines and Complexity Classes
- ▶ This is Part 1
- ▶ We examine important algorithmic building blocks in 8.2.
- ▶ Thanks to Turing Fellow and Computer Scientist Dan Martin for Tikz pictures and expertise

# ILOs

- ▶ ILO2 Be able to **use and apply basic machine learning** tools
- ▶ ILO4 Be able to use high throughput computing infrastructure and understand appropriate algorithms

## Runtime - motivation



- ▶ Consider our algorithm run on data  $D_1$ :
- ▶ Different programming languages/compiler/hardware
- ▶ How do we predict its runtime elsewhere?

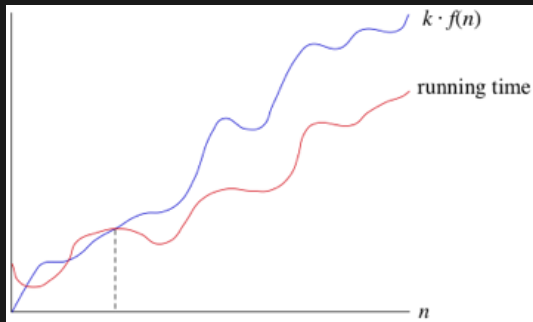
# Why study algorithms?

- ▶ Algorithms underlie every machine-learning method.
- ▶ Theoretical statements about algorithms can be made, including:
  - ▶ How long does an algorithm take to run?
  - ▶ What guarantees can be made about the answer an algorithm returns?
- ▶ In some cases, **carefully chosen algorithms** can achieve either perfect or usefully good performance at a vanishing fraction of the run time of a naive implementation.
- ▶ This can lead to a **solution on a single machine** that is superior to that of a massively parallel implementation using distributed computing.

# Algorithmic concerns

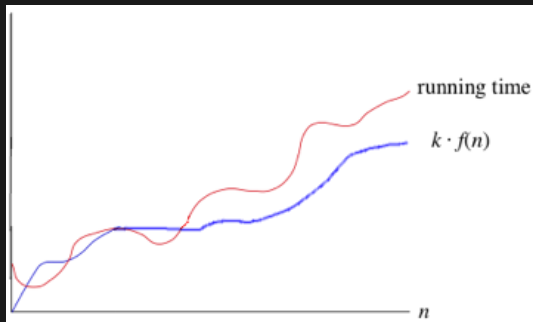
- ▶ We typically care about:
  - ▶ How long does the algorithm run for? Under which circumstances?
  - ▶ How do they trade off **runtime** and **memory requirement**?
- ▶ Some special values include **in-place** methods (which have a constant memory requirement) and **streaming** methods which visit the data exactly once each (usually with a constant-sized memory).
- ▶ Proofs typically describe the scaling of these properties, but in practice the constants are very important!

# Algorithmic complexity: Big O Notation



- ▶  $\mathcal{O}(n)$ : An upper bound as a function of data size  $n$
- ▶  $g(n) = \mathcal{O}(f(n))$ :
  - ▶  $\exists n_0, k \in \mathbb{N}$  such that:
  - ▶  $\forall n \geq n_0$ :
  - ▶  $g(n) \leq k \cdot f(n)$

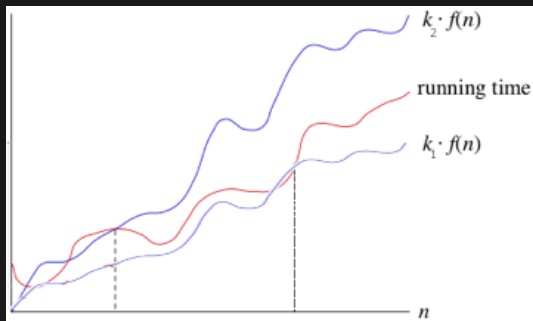
# Algorithmic complexity: Big Omega Notation



- ▶  $\mathcal{O}(n)$ : A lower bound a function of data size  $n$
- ▶  $g(n) = \Omega(f(n))$ :
  - ▶  $\exists n_0, k \in \mathbb{N}$  such that:
  - ▶  $\forall n \geq n_0$ :
  - ▶  $g(n) \geq k \cdot f(n)$



# Algorithmic complexity: Big Theta Notation



- ▶  $\Theta(n)$ : A tight bound as a function of data size  $n$
- ▶  $g(n) = \Theta(f(n))$ :
  - ▶  $\exists n_0, k_1, k_2 \in \mathbb{N}$  such that:
  - ▶  $\forall n \geq n_0$ :
  - ▶  $k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$
- ▶ i.e. the bound is strict.

# Complexity examples

- ▶  $n \in \mathcal{O}(n^2)$ 
  - ▶  $n \in \mathcal{O}(n)$  as well
  - ▶  $n \in \Omega(n)$
- ▶  $2n^2 + n + 10 \in \mathcal{O}(n^2)$
- ▶  $\log(n) \in \mathcal{O}(n^\epsilon)$  for all  $\epsilon > 0$
- ▶ If  $f(n) \in \mathcal{O}(g(n))$  then  $g(n) \in \Omega(f(n))$
- ▶ If  $f(n) \in \mathcal{O}(g(n))$  and  $f(n) \in \Omega(g(n))$  then  $f(n) \in \Theta(g(n))$
- ▶ If  $f_1(n) \in \mathcal{O}(g_1(n))$  and  $f_2(n) \in \mathcal{O}(g_2(n))$  then  $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$
- ▶ If  $f_1(n) \in \mathcal{O}(g_1(n))$  and  $f_2(n) \in \mathcal{O}(g_2(n))$  then  $f_1(n) + f_2(n) \in \mathcal{O}(\max(g_1(n), g_2(n)))$
- ▶  $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

# Algorithmic complexity: Probabilistic Analysis

- ▶ Sometimes we don't want the worst-case behaviour out of all possible inputs
- ▶ In these scenarios **average-case** run time is often reported
  - ▶ This is typically the average over the entire input space
  - ▶ This should make the statistician in you concerned!
- ▶ Randomized algorithms are also important
  - ▶ In these the answer may be random, and take a random amount of time, for a given input!
  - ▶ e.g. MCMC, etc
  - ▶ Again the **expected run time** is often reported
- ▶ We can discuss  $\Theta$ ,  $\Omega$  and  $\mathcal{O}$  of the expected runtime
- ▶ Clearly the distribution of the input data is important
- ▶ Some worst-case scenarios have “measure 0” (i.e. will never occur in practice)

# Complexity and constants

- ▶ Consider the following functions:

```
import time
def constant_fun(n,k):
    time.sleep(k * k);
def linear_fun(n,k):
    for i in range(n):
        time.sleep(1);
```

- ▶ Clearly `linear_fun` is faster for  $n < k^2$ . We need to take into account  $k$  and whether it scales with  $n$ .
- ▶ In practice  $k$  is often truly a constant but can be any scale compared to  $n$ . The accounting therefore needs to retain it.
- ▶ Example: SVD is  $\mathcal{O}(\min(mn^2, m^2n))$
- ▶ Complexity classes **only** describe **asymptotic behaviour** for large  $n$

# Divide and conquer

- ▶ One of the most popular strategies is **divide and conquer**, in which we make many sub-problems, each of which is solvable.
- ▶ This is typically valuable for parallelism
- ▶ It also makes sense to apply the algorithm **recursively**.
  - ▶ In which case we obtain expressions like:

$$T(n) = aT(n/k) + D(n) \quad \text{if} \quad n \geq c,$$

- ▶ and  $T(n) = \Theta(1)$  otherwise.
- ▶ This recursion is a relatively straightforward infinite sum (exercises) and leads to  $T(n) = \Theta(n \log_k(n))$

## Other key concepts

- ▶ **Worst case cost** conditions: can require care when looking up the answer.
  - ▶ For example, some data structures have  $\mathcal{O}(n)$  lookup cost if the data are missing, but much better if the data are present.
  - ▶ Also some costs are predictable and rare, leading to...
- ▶ **Amortised cost**: The long term, average worst case cost, which is often better than the single case cost.
  - ▶ For example, some data structures must be periodically rebuilt when they get too big, an expensive action. But this is done rarely by construction.

# Reflection

- ▶ Does it make sense to say that “ $\mathcal{O}(f(n))$  is at least  $n^2$ ”?
- ▶ In what sense would it matter in a recursive binary algorithm if  $n$  was not in  $2^k$ ?
- ▶ How do complexity statements combine?
- ▶ By the end of the course, you should:
  - ▶ Be able to compute with  $\Theta$ ,  $\Omega$  and  $\mathcal{O}$
  - ▶ Be able to reason at a high level about algorithm value

# Signposting

- ▶ Next up: Analysing Algorithms Part 2: Examining algorithms
- ▶ **References:**
  - ▶ Cormen et al 2010 Introduction to Algorithms is very accessible and recommended.
  - ▶ Arora and Barak 2007 Computational Complexity: A Modern Approach is useful but more advanced.