1 Introduction

Algorithms are all around us. Examples are:

- operating systems
- networks
- biometric algorithms
- barcode / QR code (matrix barcode) reader/scanner
- web search
- traffic lights, cars (ECUs), ...

1.1 Algorithms

An **algorithm** is a well-defined computational procedure that achieves a specified task. Normally, the procedure takes as its *input* an instance of a **computational problem**, and produces as its *output* a solution to the instance of the problem. The procedure needs to work for all instances of the problem.

A computational problem must be *general* and *well-specified*. It describes the input-output relationship: the set of possible inputs, their properties, and the desired output for each input.

Example 1.1. Formally define the sorting problem.

Problem: sorting

Input: a sequence of objects $a_1, a_2, \ldots, a_n$ which are totally ordered:

$$\forall i, j \in \{1, 2, \ldots, n\} : a_i \leq a_j \vee a_j \leq a_i$$

Output: a permutation π of $1, 2, \ldots, n$ such that

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$$

$\triangle$

Ways to describe algorithms:

- Structured English
- Pseudocode
- Program

1.2 Desired Properties

We are interested in algorithms that have the following properties (listed in their order of importance):

- Correct
- Scalable:
  - efficient in time (i.e. fast)
  - efficient in space (i.e. do not require too much memory)
  - efficient in other terms that matter (e.g. power consumption, core utilisation, anytime, etc)
- Others: general, simple, clever, etc.

1.3 Correctness of algorithms

An algorithm is said to be **correct** if for every input instance, it halts and produces the desired output.

The only way to ascertain the correctness of an algorithm is to provide a formal **proof**. A proof uses a set of axioms, assumptions, and a chain of deductive reasoning to establish the correctness of a claim.

Common proof techniques in Computer Science include *induction* and *proof by contradiction*.

Question 1.1. Can test cases be used to prove the correctness of an algorithm?

Critical thinking is important in designing algorithms. Before trying to prove the correctness of an algorithm it is worth trying to prove its incorrectness by finding a counterexample. A **counterexample** for an algorithm is an instance of a problem for which the algorithm does not produce the correct output. A counterexample is a proof of the incorrectness of the algorithm.

Refer to the textbook to see a Job Scheduling problem (Section 1.2), the Travelling Salesman Problem (TSP) (section 1.1), and a collection suggestions on how to find counterexamples (Section 1.3.3).

2 Analysis

The most important property of an algorithm is its *correctness*. Once we know an algorithm is correct, we need to know how efficient it is – that is, how much computational resources it needs to complete its task. Efficiency matters because it determines how scalable an algorithm is.

Among different types of resources that are required by an algorithm, we are usually interested in knowing

- how much *time* it would take to solve a problem (time complexity);
- how much *space* (memory) it requires to do so (space complexity).

Complexities are expressed in the form of a function of some property of the input (usually size).

2.1 The RAM Model of Computation

Talking about efficiency of algorithms without having a computational model in mind is not possible. Yet we do not want to be confined to a specific computing platform (hardware, operating system, programming language, etc). We need an abstract model that a) is close enough to common real computers; and b) is simple enough to reason about.

The *RAM model of computation* is an abstraction of real-world computers that is commonly used in analysis of algorithms. It assumes:

- Accessing any memory cell takes a constant time irrespective of its location and content (1 step)
- Basic operations such as arithmetic $(+, -)$, comparisons $(\leq, =)$ and assignments take a constant time (1 step)
- Loops take time proportional to the number of times they iterate.
- The time a subroutine call takes is equal to the time it takes to execute its body.

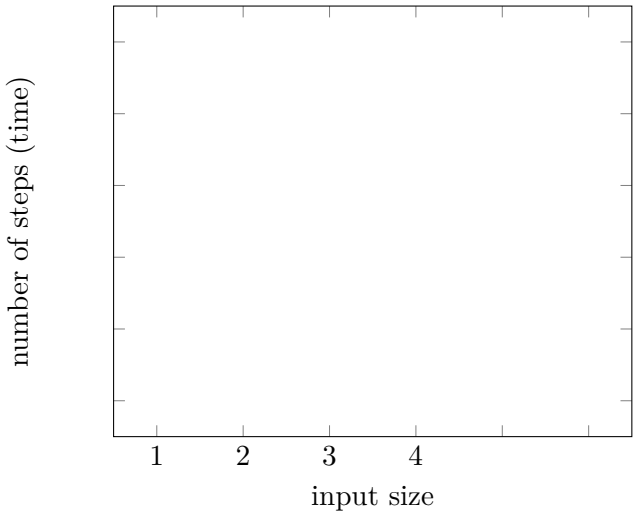
We measure the running time of an algorithm by counting the number of steps it takes to complete.

Remarks:

- The RAM model is not accurate but it can provide rough estimates for the running time of algorithms.
- There are other models of computation (see COSC261).

2.2 Worst-Case, Average-Case, and Best-Case Complexities

An illustration of the worst-case and best-case time functions for a given (unspecified) algorithm:



These cases can be defined more formally. Let

- $\mathcal{X}$ be the set of all possible inputs for a given algorithm;
- $size : \mathcal{X} \to \mathbb{N}$ be a function that measure the size of an input—that is, $size(x)$ is the size of input x; and
- $time : \mathcal{X} \to \mathbb{N}$ measure the time that the given algorithm takes on each input—that is, $time(x)$ is the number of steps (time) the algorithm takes to compute the output for input x.

Then the *worst-case* time function can defined as

$$T_{WC}(n) = \max_{x \in \{x \in \mathcal{X} | size(x) = n\}} time(x) \ .$$

The *best-case* time function can be defined as

$$T_{BC}(n) = \min_{x \in \{x \in \mathcal{X} | size(x) = n\}} time(x) \ .$$

The average case time function can be defined as

$$T_{AC}(n) = \sum_{x \in \{x \in \mathcal{X} | size(x) = n\}} P\{x |\, size(x) = n\}\, time(x) \ .$$

where $P\{x |\, size(x) = n\}$ is the conditional probability density of x. [The average case is usually difficult to obtain and won't be discussed much in this course.]

Question 2.1. Among different cases for complexity, we are often interested in the worst-case. Why?

2.3 Asymptotic Notation

Motivations:

- The exact time complexity of an algorithm (as an expression/function of its input size) can have too much unnecessary details or can be too complicated to analytically obtain.
- We want to be able to measure the complexity of algorithms and compare them disregarding factors such as hardware speed and programming languages.

Asymptotic (Big O) notation is used to specify upper or lower bounds for complexity functions after ignoring multiplicative constants and lower-order terms.

2.3.1 Formal Definitions

$O(g)$ is the set of all functions that grow <u>at most</u> as fast as g (i.e. g is an asymptotic upper bound for them).

$$O(g) = \{f \mid \exists c > 0 : \exists x_0 : \forall x \geq x_0 : 0 \leq f(x) \leq c\, g(x)\}$$

$\Omega(g)$ is the set of all functions that grow <u>at least</u> as fast as g (i.e. g is an asymptotic lower bound for them).

$$\Omega(g) = \{f \mid \exists c > 0 : \exists x_0 : \forall x \geq x_0 : 0 \leq c\, g(x) \leq f(x)\}$$

$\Theta(g)$ is the set of all functions that grow as fast as g (i.e. g is an asymptotic lower and upper bound for them).

$$\Theta(g) = O(g) \cap \Omega(g)$$

2.3.2 Membership Properties

$$f \in O(g) \wedge f \in \Omega(g) \Leftrightarrow f \in \Theta(g)$$

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

2.3.3 Algebraic Properties

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \times f_2 \in O(g_1 \times g_2)$$

$$g_1, g_2 \geq 0 \Rightarrow O(g_1 + g_2) = O(\max(g_1, g_2))$$

These properties (expressed in terms of O), also hold for Ω and consequently for Θ.

2.3.4 Note on usage

The asymptotic notation might appear in two types of mathematical expressions with two slightly different meanings:

1. with an equal sign, for example, $T(n) = O(n^2)$, in which case it must be interpreted as $T(n) \in O(n^2)$; or
2. within an arithmetic expression, for example, $n^2 + \Theta(n)$, in which case it must be interpreted as $n^2 + f(n)$ where f is a function in $\Theta(n)$.

2.4 A visual guide to asymptotic notation

$n!$	$5n!$	$n! + 2^n$	$\ldots$
2^n	5×2^n	2^{n+1}	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n^3	$n^3 + 2n^2 + 4n + 1$	$5n^3 + 5$	$\ldots$
n^2	$4n^2 - n + 3$	$n(n+1)$	$\ldots$
$n \log n$	$n \log n + n$	$n \log n^2$	$\ldots$
n	$7n + 4$	$n + \log n$	$\ldots$
$\log n$	$\log n + 3$	$\log n^5$	$\ldots$
1	$\log 9$	2^{500}	$\ldots$

- Functions on the same line are *equivalent* (have the same order).
- The table is not complete. Each line has infinitely many members. There are infinitely many classes between each pair of lines.
- Any function on a given line *dominates* all the functions on the lines below it.
- O of some expression is the set of functions on the corresponding line and all the lines below it.
- Ω of some expression is the set of functions on the corresponding line and all the lines above it.
- Θ of some expression is the intersection of O and Ω of that expression, that is, the set of functions on the corresponding line.
- O, Ω or Θ of any function on a line is the same as that of any other function on the same line. For example, $O(n) = O(7n + 4) = O(n + \log n) = \ldots$
- Think of different cases of running time as a vertical band with the worst case on top, the best case at the bottom, and other cases in between. For some algorithms, the band may span over multiple lines, for some others, it may be entirely on one line.

- A tight bound on a function (for example, worst-case complexity) or set of functions (for example, all-input complexity) is more informative.
- There is no correspondence between asymptotic bounds (O, Ω, and Θ) and the three named time complexity functions (T_{WC}, T_{BC}, and T_{AC}); these are separate concepts. For example, the Θ notion can be used to express a tight bound for T_{WC}. In fact, such a bound would be more informative than a bound expressed with only big O (upper and lower bounds vs only an upper bound).
- When working with logarithms, the following properties may be useful:

  $\log ab = \log a + \log b$

  $\log a^b = b \log a$

2.5 Analysis of Efficiency

2.5.1 (Amortised) Time Complexity of Operations in Python

Operation	Comment	Complexity
`=`	simple assignment (with no complex RHS evaluation)	$\Theta(1)$
`+,-,*,/,//,abs`	simple operations on small numbers	$\Theta(1)$
List indexing	to read or write a single element	$\Theta(1)$
List length	Python keeps track of the length (no need to count)	$\Theta(1)$
Getting a list slice	where the length of the slice is k	$\Theta(k)$
List copy	where the length of the list is n	$\Theta(n)$
List append	appending one element at the end (right)	$\Theta(1)$
List pop	popping the right-most element	$\Theta(1)$
List insert or pop	at non-negative index k of a list of length n	$\Theta(n-k)$
Membership test on lists	using the `in` operator	$O(n)$
Membership test on sets	using the `in` operator	$\Theta(1)$
`min, max, sum`	on a list of length n	$\Theta(n)$
Sorting	a list of length n	$O(n \log n)$

2.5.2 Examples of time analysis on Python programs

Analyse the time complexity of the following Python programs.

```
def proc(a, b, c):
  n = len(a)
  for i in range(n):
    for j in range(n):
      c[i][j] = 0
      for k in range(n):
        c[i][j] += a[i][k] * b[k][j]
```

```
def proc(n):
  r = 0
  for i in range(n):
    for j in range(i+1, n-1):
      for k in range(j):
        r += 1
  return r
```

```
def proc(list_a):
  list_b = list_a
  n = len(list_b)
  if n >= 2:
    x = list_a[0] + list_a[1]
  s = sum(list_b)
  list_a.sort()
```

```
  def proc(alist):
    for i in range(len(alist)):
      min_index = i
      for j in range(i+1, len(alist)):
       if alist[j] < alist[min_index]:
         min_index = j
      alist[i], alist[min_index] = alist[min_index], alist[i]
```

```
  def proc(alist):
    for i in range(len(alist)):
      a = alist[i]
      j = i
      while j >= 1 and alist[j-1] > a:
        alist[j] = alist[j-1]
        j -= 1
      alist[j] = a
```

```
def proc(a):
    if a[0] == 0:
        return min(a)
    else:
        return insertion_sort(a)
```

```
def proc(n):
  r = 0
  while n > 1:
    r += 1
    n -= 2
```

```
def proc(n):
  r = 0
  while n > 1:
    r += 1
    n //= 2
```

```
def proc(n):
  while n > 1:
    if n \% 2 == 0:
      n //= 2
    else:
      n = 3 * n + 1
```