Chapter 3

Sorting

3.1 The motivation

In the previous chapter we saw how a linear program could be solved. We can find every vertex of the feasible region and then evaluate the objective function at all vertices. By sorting this values we then have the optimal solution.

However, we claimed that this method was impractically slow, as the computational cost can grow as m^m (sometimes called exponentially, or geometrically). We should make this explicit. Assume that one operation (finding one vertex and computing the objective function there) takes one microsecond, 1μs. Then compare how long it would take if the computational cost were linear (grows as m), quadratic (grows as m^2), or geometric (grows as 2^m or as m^m) as the number of vertices m increases:

Vertices (m)	Linear (m)	Quadratic (m^2)	2^m	m^m
1	1μs	1μs	1μs	1μs
10	10μs	$10^2\mu$s	$\sim 10^3\mu$s	~ 3 hours
100	100μs	10^{-2}s	$\sim 10^{22}$ years	$\sim 10^{192}$ years

We see that the power law cases (linear and quadratic) both complete in fractions of a second for a hundred vertices, whilst neither geometric case will complete within the lifetime of the universe. Note that Chvátal, writing around 1980, comments that problems with $m \geq 10^3$ were being solved routinely. In modern usage linear programs with $m \geq 10^6$ are being solved in under a minute by the best solvers.

3.2 The problem

The problem, for this chapter, is to understand how to analyse an algorithm. Once we can measure how fast, in principle, an algorithm completes its task, we can use this to compare different techniques.

We will not apply this analysis to linear programming algorithms at first, as there is a simpler case at hand. That is the final step in our impractical algorithm for linear programs: sorting a list of numbers.

3.2.1 Sorting

The *sorting problem* is to take a list a of objects with a *partial order* $\leq$, sort the list into non-decreasing order.

The type of the objects is left unspecified. They could be real numbers or integers, which will be our standard examples. They could be letters or words. The key point is that it must be possible to compare any two objects in the list to give an order. That is, $\forall x, y \in a$, the comparison

$x \leq y$ must be either true or false. A list containing all integers is fine. A list containing a mix of letters and numbers is not (without further assumptions). A list containing complex numbers is not fine either.

We assume the list has length n and refer to the individual entries in the list as a_i, where $i \in \{1, \ldots, n\}$ is the *index*. We are looking to construct a *permutation operator* $\pi \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ which is a bijective function on the indexes. The permutation operator tells us where to go in the original list to find the object at a given position in the sorted list.

Formally, the problem is to find a permutation π such that

$$a_{\pi(i)} \leq a_{\pi(i+1)} \quad \forall i \in \{1, \ldots, n\}. \tag{3.1}$$

As a concrete example, consider the list $a = (7, 2, 2, 6, 1, 4)$. By eye we see the sorted list is $s = (1, 2, 2, 4, 6, 7)$. Looking at the indices we construct the permutation:

$$\begin{array}{ccccccccc} a & : & 7 & 2 & 2 & 6 & 1 & 4 \\ \text{index} & : & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \quad \to \quad \begin{array}{ccccccccc} s & : & 1 & 2 & 2 & 4 & 6 & 7 \\ \pi(\text{index}) & : & 5 & 2 & 3 & 6 & 4 & 1 \end{array} \tag{3.2}$$

3.3 Algorithms

We look at three different sorting algorithms before introduing a detailed analysis.

3.3.1 Brute Force Sort

This is a two step process.

1. Enumerate all possible permutations π of length n.
2. Return the first permutation giving a sorted list.

This is closely linked to the impractical algorithm introduced to solve the linear programming problem.

It is possible that only one permutation sorts the list, and that (by bad luck) it is the last one constructed. Given a list with n items we can construct $n!$ permutations of that list (consider how many positions the first object can take, then, having fixed the first, how many the second can take, and so on). For large n we have that $n! \sim n^n$, and we have seen how rapidly that grows.

3.3.2 Selection sort

This is an iterative process.

1. Find the smallest object.
2. Swap it with the object in position one (if strictly smaller).
3. Find the next smallest object in the list; alternatively, consider the list *except* the first object.
4. Swap it with the object in position two (if strictly smaller).
5. Continue to the end of the list.

When implemented in code this algorithm requires two loops. The outer loop runs over the whole list and corresponds to the position we are going to swap the "smallest" object into. The inner loop is to find the smallest object, and runs over only part of the list.

The steps for $a = (7, 2, 2, 6, 1, 4)$ are as follows. When we find the smallest object it will be in the sublist denoted by square brackets:

1. (a) : Smallest object in $([7, 2, 2, 6, 1, 4])$ is 1 at index 5.
   (b) : Swap with object at index 1, giving $(1, 2, 2, 6, 7, 4)$.

2. (a) : Smallest object in $(1, [2, 2, 6, 7, 4])$ is 2 at index 2.
   (b) : Do not swap as already at index 2, giving $(1, 2, 2, 6, 7, 4)$.
3. (a) : Smallest object in $(1, 2, [2, 6, 7, 4])$ is 2 at index 3.
   (b) : Do not swap as already at index 3, giving $(1, 2, 2, 6, 7, 4)$.
4. (a) : Smallest object in $(1, 2, 2, [6, 7, 4])$ is 4 at index 6.
   (b) : Swap with object at index 4, giving $(1, 2, 2, 4, 7, 6)$.
5. (a) : Smallest object in $(1, 2, 2, 4, [7, 6])$ is 6 at index 6.
   (b) : Swap with object at index 5, giving $(1, 2, 2, 4, 6, 7)$.
6. We have reached the last object, so the list is sorted.

3.3.3 Insertion sort

Conceptually this process splits the list a into two pieces, or sublists. One is sorted and the other unsorted. We write $a = (s|u)$ where $|$ marks the separation between the sublists. At the start s is empty and $u = a$. Then, while u is not empty, we

1. Compare the first entry of the unsorted list u with each entry of the sorted list s in turn *from the right.*
2. If $u_1 < s_i$, swap the entry in the sorted list with the first entry in the unsorted list. Do this for each entry in the sorted list until u_1 is in order.
3. Move the separation marker one entry to the right (so that s_i, which was swapped with u_1, is still in the sorted list).

As with selection sort, when implemented in code we have two loops, one running over the whole list and one over some part of the list. We may expect the computational cost of the two to be similar.

The steps for $a = (7, 2, 2, 6, 1, 4)$ are as follows.

1. (a) $(|7, 2, 2, 6, 1, 4)$ has nothing in the unsorted list.
   (b) Move the separation marker to the right, giving $(7|2, 2, 6, 1, 4)$.
2. (a) As $2 < 7$, swap s_1 and u_1, giving $(2|7, 2, 6, 1, 4)$.
   (b) Move the separation marker to the right, giving $(2, 7|2, 6, 1, 4)$.
3. (a) As $2 < 7$, swap s_2 and u_1, giving $(2, 2|7, 6, 1, 4)$.
   (b) As 2 is not less than 2 no further sorting is needed.
   (c) Move the separation marker to the right, giving $(2, 2, 7|6, 1, 4)$.
4. (a) As $6 < 7$, swap s_3 and u_1, giving $(2, 2, 6|7, 1, 4)$.
   (b) As 6 is not less than 2 no further sorting is needed.
   (c) Move the separation marker to the right, giving $(2, 2, 6, 7|1, 4)$.
5. (a) As $1 < 7$, swap s_4 and u_1, giving $(2, 2, 6, 1|7, 4)$.
   (b) As $1 < 6$, swap s_3 and (what was) u_1, giving $(2, 2, 1, 6|7, 4)$.
   (c) As $1 < 2$, swap s_2 and (what was) u_1, giving $(2, 1, 2, 6|7, 4)$.
   (d) As $1 < 2$, swap s_1 and (what was) u_1, giving $(1, 2, 2, 6|7, 4)$.
   (e) Move the separation marker to the right, giving $(1, 2, 2, 6, 7|4)$.
6. (a) As $4 < 7$, swap s_5 and u_1, giving $(1, 2, 2, 6, 4|7)$.

(b) As $4 < 6$, swap s_4 and (what was) u_1, giving $(1, 2, 2, 4, 6|7)$.

(c) As 4 is not less than 2 no further sorting is needed.

(d) Move the separation marker to the right, giving $(1, 2, 2, 4, 6, 7|)$.

7. The unsorted list is empty, so the sorted list is complete.

3.4 Analysis of algorithms

We have three different algorithms to sort a list, and a rough idea of how they will behave. However, that rough idea could easily be wrong. If the first permutation generated by the brute-force method sorts the list then it is faster than either selection or insertion sort.

Equally, it is generally impractical to work out how long a sorting algorithm may take in all cases and describe the result statistically, as the time taken will diverge to infinity with the size of the list.

Instead, we will look at the *worst case*: what is the maximum length of time an algorithm may take?

3.4.1 Definitions

An *elementary operation* (EO) is any simple operation an algorithm may take with roughly the same cost. Examples would be comparisons, swaps, additions, multiplications, or assigning a value to a variable.

The *instance size* is a number n that characterises the size of the problem. When sorting it would be the length of the list a.

In the *worst case* analysis we reduce every choice that the algorithm makes to the one needing the most elementary operations.

Useful results

The *Gauss series* or arithmetic progression formula is

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{3.3}$$

The size of a set of consecutive integers, $\{i, i+1, \ldots, j\}$ is

$$|\{i, i+1, \ldots, j\}| = j - i + 1. \tag{3.4}$$

3.4.2 Selection Sort

```
1 for i in range(0, n-1):
2     j = i
3     for k in range(i+1, n):
4         if a[k] < a[j]:
5             j = k
6     if a[j] < a[i]:
7         a[i], a[j] = a[j], a[i]
```

This Python code performs selection sort.

- Line 1 loops $n - 1$ times over:
  - Line 2 assigns one value (1 EO);
  - Line 3 loops $n - (i + 1) + 1$ times over:

* Line 4 does one comparison (1 EO);
* Line 5 assigns one value (1 EO);

The total cost of the loop is $2(n-i)$ EOs;

– Line 6 does one comparison (1 EO);

– Line 7 does one comparison (1 EO);

The total cost of the loop is

$$\sum_{i=1}^{n-1}(1+2(n-i)+2) = \sum_{i=1}^{n-1}(2n+3) - 2\sum_{i-1}^{n-1} i \tag{3.5}$$

$$= (n-1)(2n+3) - 2\frac{(n-1)n}{2} \tag{3.6}$$

$$= (n-1)(n+3) = n^2 + 2n - 3. \tag{3.7}$$

We see that the analysis gives the worst case of selection sort as $\sim n^2$. As shown earlier this grows much more slowly that the worst case $\sim n^n$ for the brute-force algorithm. Selection sort may be practical.

3.4.3 Insertion Sort

```
1 for i in range(1, n):
2     for j in range(i, 0, -1):
3         if a[j] < a[j-1]:
4             a[i], a[j] = a[j], a[i]
5         else:
6             break
```

This Python code performs insertion sort.

- Line 1 loops $n-1$ times over:

  – Line 2 loops i times over:

    * Line 3 does one comparison (1 EO);
    * Line 4 does one swap (1 EO);
    * Lines 5 and 6 never occur in the worst case.

    The total cost of the loop is $2(i-i)$ EOs;

  The total cost of the loop is

$$\sum_{i=2}^{n} 2(i-1) = \sum_{i=1}^{n} 2(i-1) \tag{3.8}$$

$$= 2\sum_{i=1}^{n} i - 2\sum_{i=1}^{n} 1 \tag{3.9}$$

$$= \frac{n(n+1)}{2} - 2n \tag{3.10}$$

$$= n^2 - n. \tag{3.11}$$

We see that the analysis gives the worst case of insertion sort as $\sim n^2$. For large n we expect insertion sort to be slightly faster than selection sort.

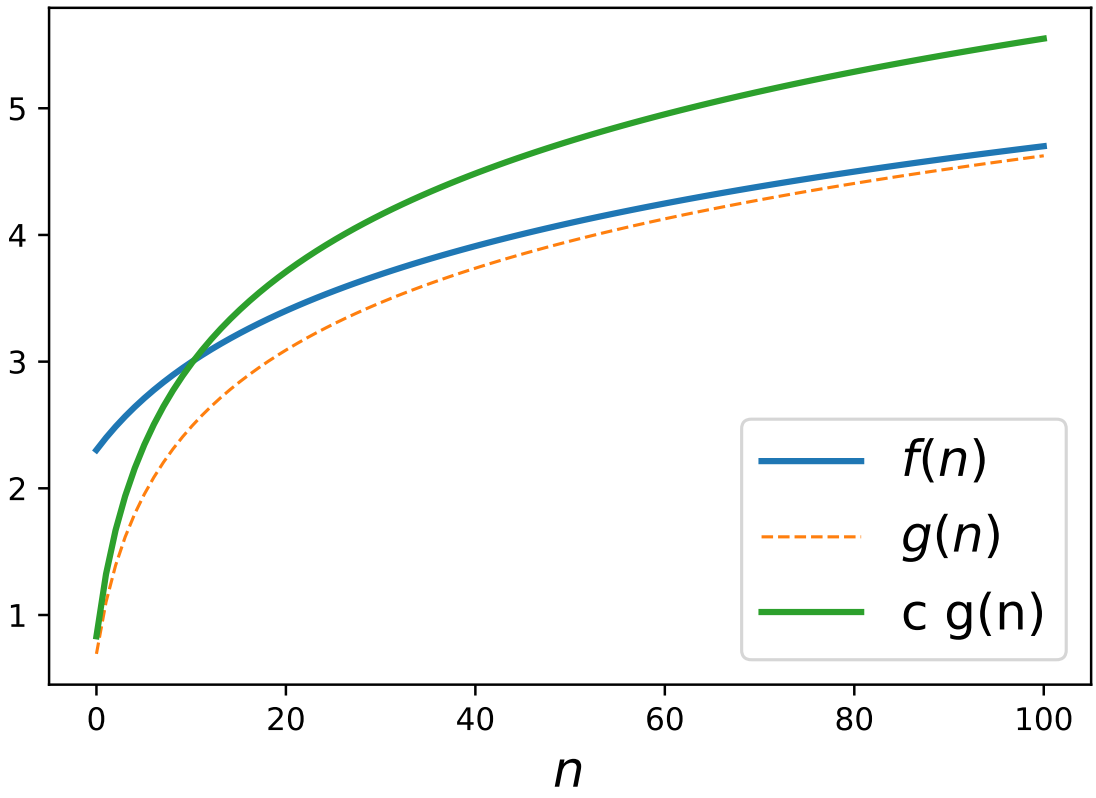

Figure 3.1: The function f is of the same order as the function g if it can be bounded by some constant multiple of g for all positive real numbers.

3.5 Computational complexity

In general, comparing algorithms by computing the number of elementary operations explicitly can be painfully hard to do. Also, it is often not useful. What we care about is how the cost (the number of elementary operations) grows as the instance size n grows. We therefore only care about the *largest* term in the *worst case* as $n \to \infty$.

3.5.1 Big $\mathcal{O}$ notation

Given two functions $f, g \colon \mathbb{R} \to \mathbb{R}$, we say that $f = \mathcal{O}(g)$, or f is of the same order as g, if $\exists n_0, c \in \mathbb{R}^+ \setminus \{0\}$ such that

$$f(n) \leq cg(n) \quad \forall n \geq n_0. \tag{3.12}$$

In other words, for large enough values f can be bounded by scaling g by a constant. This is illustrated in figure 3.1.

The purpose of computational complexity analysis is to bound any function f in terms of a small, simple set of functions g. We want the functions to be simple to make the analysis easier. Equally, we want the specific function used to be as close as possible to f, so the bound does not over-estimate too much.

3.5.2 Computational complexity

Consider two algorithms A and B that solve the same problem P. Let $f_A(n), f_B(n)$ be the number of elementary operations needed for each algorithm as a function of the instance size. Then $\mathcal{O}(f_A)$

is the *computational complexity* of algorithm A (similarly for algorithm B), and we consider A to be *more efficient* than B if

$$\mathcal{O}(f_A) < \mathcal{O}(f_B) \tag{3.13}$$

asymptotically, in the limit as $n \to \infty$.

Examples include

- An $\mathcal{O}(\log n)$ algorithm is more efficient than an $\mathcal{O}(n)$ algorithm.
- An $\mathcal{O}(n^2)$ algorithm is more efficient than an $\mathcal{O}(n^3)$ algorithm.
- An $\mathcal{O}(n^k)$ algorithm is more efficient than an $\mathcal{O}(2^n)$ algorithm, if $k \in \mathbb{N}$.

There are two implicit assumptions we have to make about the functions $f(n)$ to make progress. One is that the number of elementary operations is strictly non-negative so that $f\colon \mathbb{R}^+ \setminus \{0\}$. This seems natural. The other is that f is strictly increasing. This need not be true in general, but is usually a very good approximation.

3.5.3 Properties of the asymptotic model

When functions are the same order

Property 3.5.1. *Let $f, g\colon \mathbb{R}^+ \setminus \{0\} \to \mathbb{R}^+ \setminus \{0\}$. If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in (0, \infty) \tag{3.14}$$

then $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$.

Proof. The definition of the limit implies $\exists \epsilon > 0, N > 0$ such that

$$\left| \frac{f(n)}{g(n)} - l \right| < \epsilon, \quad \forall n \geq N. \tag{3.15}$$

This is equivalent to

$$l - \epsilon < \frac{f(n)}{g(n)} < l + \epsilon, \quad \forall n \geq N. \tag{3.16}$$

We now have two cases.

1. From the upper bound we have $f(n) < (l + \epsilon)g(n)$ for all $n \geq N$. By setting $n_0 = N$ and $c = (l + \epsilon)$, the definition gives that $f = \mathcal{O}(g)$.
2. From the lower bound we have (assuming, without loss of generality, that $\epsilon < l$) $g(n) < (l - \epsilon)^{-1} f(n)$. By setting $n_0 = N$ and $c = (l - \epsilon)^{-1}$, the definition gives that $g = \mathcal{O}(f)$.

□

This shows that (non-zero!) constant factors are essentially irrelevant.

When one function is of lower order

Property 3.5.2. *Let $f, g\colon \mathbb{R}^+ \setminus \{0\} \to \mathbb{R}^+ \setminus \{0\}$. If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{3.17}$$

then $f = \mathcal{O}(g)$ but $g \neq \mathcal{O}(f)$.

Proof. This is very similar to the previous proof.

The definition of the limit implies $\exists \epsilon > 0, N > 0$ such that

$$\left| \frac{f(n)}{g(n)} - 0 \right| < \epsilon, \quad \forall n \geq N. \tag{3.18}$$

This is equivalent to

$$-\epsilon < \frac{f(n)}{g(n)} < \epsilon, \quad \forall n \geq N. \tag{3.19}$$

We now have two cases.

1. From the upper bound we have $f(n) < \epsilon g(n)$ for all $n \geq N$. By setting $n_0 = N$ and $c = \epsilon$, the definition gives that $f = \mathcal{O}(g)$.
2. By contradiction, assume $g = \mathcal{O}(f)$. By definition, there must exist $n_0, c \in \mathbb{R}^+ \setminus \{0\}$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$. Thus

$$\frac{f(n)}{g(n)} \geq \frac{1}{c} > 0, \quad \forall n \geq n_0. \tag{3.20}$$

   This contradicts the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Therefore $g \neq \mathcal{O}(f)$.

□

In this case we say that f is of *lower order* than g.

Constants are irrelevant

Property 3.5.3. *Assume $a, b \in \mathbb{R}^+ \setminus \{0\}$. Then*

$$af(n) + b = \mathcal{O}(f(n)) \tag{3.21}$$

Proof. This follows from theorem 3.5.1, as

$$\lim_{n\to\infty} \frac{af(n) + b}{f(n)} = a \in (0, \infty). \tag{3.22}$$

□

All logarithms are the same

Property 3.5.4. *Assume $a, b \in \mathbb{R}^+ \setminus \{0\}$. Then*

$$\log_a n = \mathcal{O}\left(\log_b n\right). \tag{3.23}$$

Proof. This follows from the previous case, as

$$\log_a n = \left(\frac{1}{\log_b a}\right) \log_b n \tag{3.24}$$

and the term in brackets is a constant. □

Lower order terms are irrelevant

Property 3.5.5. *If g is of lower order than f then*

$$f(n) + g(n) = \mathcal{O}(f). \tag{3.25}$$

Proof.

$$\lim_{n\to\infty} \frac{f(n) + g(n)}{f(n)} = \lim_{n\to\infty} \left(1 + \frac{g(n)}{f(n)}\right) \tag{3.26}$$

$$= 1. \tag{3.27}$$

Hence theorem 3.5.1 gives the result. □

Shifts are irrelevant

Property 3.5.6. *For a monotone increasing function f,*

$$f(n+a) = \mathcal{O}(f). \tag{3.28}$$

Rounding operators are irrelevant

Property 3.5.7. *The ceiling operator $\lceil n \rceil$ returns the smallest integer greater than n. We have*

$$\lceil f(n) \rceil = \mathcal{O}(f). \tag{3.29}$$

Proof. This follows from

$$\lceil f(n) \rceil \leq f(n) + 1 = \mathcal{O}(f). \tag{3.30}$$

□

3.6 Cobham-Edmonds thesis

A suggestion from 1965 by Alan Cobham and Jack Edmonds is that a problem P is *tractable* if

1. there is an algorithm A that solves P;
2. A has computational complexity at most polynomial in the size of instances of P.

Generally we think of tractable problems as solvable on a computer, whilst problems that are not tractable are not. In many specific cases this is untrue, as "average" instances of problems may be solved by the algorithm A much faster than the worst case suggests.

This chapter has shown that the sorting problem is tractable, by constructing two algorithms (selection and insertion sort) that are $\mathcal{O}(n^2)$. Neither are the fastest sorting algorithm available. It is important to note that the existence of a slower algorithm (such as brute-force sort, with complexity $\mathcal{O}(n!)$) does not mean that the *problem* in intractable.