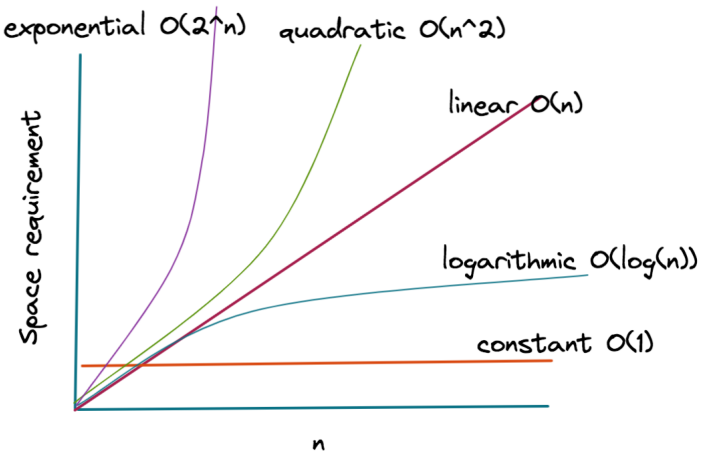


LINMA2111 - Discrete mathematics II

Simon Desmidt
Issambre L'Hermite Dumont

Academic year 2025-2026 - Q1

UCLouvain

Contents

Introduction

1.1 Sorting problems

1.1.1 Introduction

A sorting problem is a problem that consists of taking a sequence of n objects and putting them in order. This kind of problem is made of three main elements:

- Context: set S with a partial order $<$;
- Input: n elements of S;
- Output: permutation of the input elements respecting the order.

To prove the correctness of an algorithm, we generally use the Hoare triple, i.e. a tuple for any input array x_0:

$$\begin{aligned} &\{\text{Algorithm to be used; Precondition; Postcondition}\} \\ &\{IS;\ x = x_0;\ x \text{ is sorted and is a permutation of } x_0\} \end{aligned} \tag{1.1}$$

where IS is the insertion sort algorithm, and x is the sorted array. From now on, we will call "sorting" a sorted permutation.
In practice, to prove the correctness of an algorithm, we define the invariant and the base case, and do an induction step show that the invariant is preserved.

1.1.2 Definition of complexity

For some functions $f, g : \mathbb{N} \to \mathbb{R}^+$,

$$\begin{aligned} &f \in \mathcal{O}(g) \iff \exists c > 0, \exists n_0, \forall n > n_0,\ f(n) \leq cg(n) \\ &f \in \Omega(g) \iff g \in \mathcal{O}(f) \\ &f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ and } f \in \Theta(g) \\ &f \in o(g) \iff \exists c > 0, \exists n_0, \forall n > n_0,\ f(n) < cg(n) \\ &f \in \omega(g) \iff g \in o(f) \end{aligned} \tag{1.2}$$

- The time complexity is the number of operations as a function of the input size;
- The space complexity is the amount of memory used (in addition to the input) as a function of the input size.

We define the average case complexity as an expectation of the time taken by the algorithm on each input possible, with a dependence in the size of the input.

$$t = \mathbb{E}_{x \sim D}[T(x)] \tag{1.3}$$

where D is the distribution of the input.

1.1.3 Divide and conquer

The divide and conquer method consists in three steps:

1. Divide: create smaller subproblems;
2. Recurse: solve them;
3. Combine: merge the solutions.

In sorting problems, a divide-and-conquer algorithm is the merge sort algorithm. It consists in dividing the array in two, sorting each half recursively, and merging the two sorted halves. The merge operation is done in linear time.

1.2 Complexity of sorting algorithms

No sorting algorithm can be faster than $\Omega(n \log(n))$. This can be proven through the complexity of comparison-based algorithms.

Divide-and-conquer algorithms

2.1 Complexity of a multiplication

Given two n-digit numbers, we want to compute their product:

$$\begin{cases} a = a_{n-1}a_{n-2}...a_1a_0 \\ b = b_{n-1}b_{n-2}...b_1b_0 \end{cases} \implies c = a \cdot b = c_{2n-1}...c_0 \tag{2.1}$$

Let us define B as the basis (e.g. 10) and decompose a and b in two parts:

$$\begin{cases} a = \alpha_0 + B\alpha_1 \\ b = \beta_0 + B\beta_1 \end{cases} \implies a \cdot b = \alpha_0\beta_0 + B(\alpha_1\beta_0 + \alpha_0\beta_1) + B^2\alpha_1\beta_1 \tag{2.2}$$

In that case, we find a recurrence relation for the computation time:

$$T(n) = 4T(n/2) + \Theta(n) \tag{2.3}$$

We introduce the Master theorem to solve this equation, see section below. It gives a complexity of $T(n) = \Theta(n^{\log_2(4)}) = \Theta(n^2)$.
To reduce the complexity, we can change the value of the parameter a from 4 to 3 by calculating only 3 products:

$$\begin{cases} \gamma_0 = \alpha_0\beta_0 \\ \gamma_2 = \alpha_1\beta_1 \\ \gamma_1 = (\alpha_0 + \alpha_1)(\beta_0 + \beta_1) - \gamma_0 - \gamma_2 \end{cases} \tag{2.4}$$

This reduces the complexity to $\Theta(n^{1.58})$.
Following a similar reasoning, we can instead divide a and b into 3 sums instead of 2, and get 5 multiplications. This gives a complexity of $\Theta(n^{\log_3(5)}) = \Theta(n^{1.46})$. Dividing in 4, 5, etc, we converge to a complexity of $\Theta(n)$ and this is the optimal complexity for the multiplication of two numbers of n digits. The problem is that the constant in front of the n starts to grow as we divide into more and more limbs, and so a bigger exponent can be enough for most arrays[1].

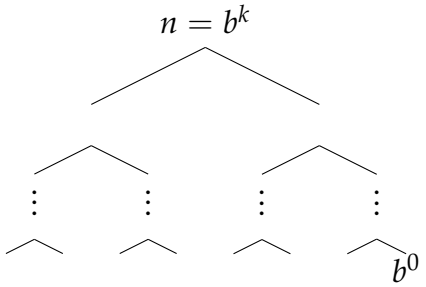
[1] We call galactical algorithm an algorithm that is asymptotically good but the constant grows so large that it is only useful for huge arrays (e.g. $\sim 10^{80}$).

2.2 Master Theorem

Let $a \geq 1$ and $b > 1$ be constants and $f(n)$ a positive function, and let $T(n)$ be defined by $T(0) > 0$ and $T(n) = aT(\lfloor n/b \rfloor) + f(n)$. Then,

- If $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$;
- If $f(n) = \Theta(n^{\log_b(a)})$, then, $T(n) = \Theta(b^{\log_b(a)} \log(n))$;
- If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and if, for some $c > 1$ and n_0 such that $af(\lfloor n/b \rfloor) \leq cf(n)$ for all $n \geq n_0$, then $T(n) = \Theta(f(n))$;

2.2.1 Proof



In that tree, for a level i, the size of the problem is b^i and the number of subproblems is a^{k-i}. By summing up,

$$T(b^k) = \sum_{i=0}^{k} a^{k-i} f(b^i) \tag{2.5}$$

For a function $f(n) = n^\alpha$,

$$T(n) = a^k \sum_{i=0}^{k} \left(\frac{b^\alpha}{a}\right)^i = a^k \frac{1 - (\frac{b^\alpha}{a})^{k+1}}{1 - b^\alpha/a} \tag{2.6}$$

Under the assumption that $\alpha \neq \log_b(a)$. As $k = \log_b(n)$, we can replace above and simplify using the formula $a^{\log_b(n)} = n^{\log_b(a)}$. In the case where $a = b^\alpha$, then

$$T(n) = a^k(k+1) = a^{\log_b(n)}(\log_b(n) + 1) = \Theta(n^{\log_b(a)} \log(n)) \tag{2.7}$$

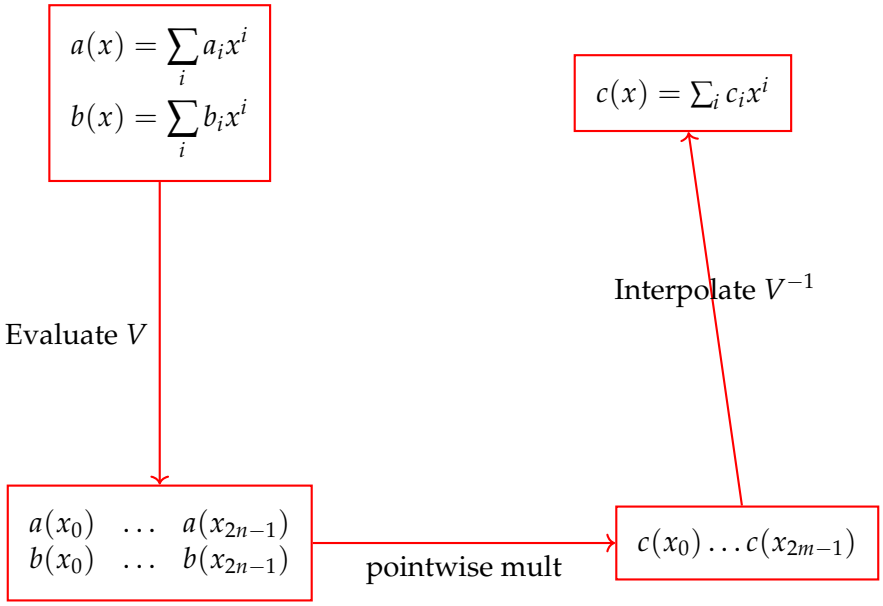
2.3 Multiplying polynomials

Consider $a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$. It can either be represented by its coefficients, or some values $(a(x_0), \ldots, a(x_{n-1}))$ for n distinct values. To compute those values, we can use a matrix-vector product:

$$\begin{bmatrix} 1 & x_0 & \ldots & x_0^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} = Va \tag{2.8}$$

where the matrix V is called the Vandermonde matrix.

2.3.1 Convolution theorem



This method gives bad conditioning, but this is not a problem here as we consider integer matrices.

2.3.2 Discrete Fourier Transform

If we take the points as the n complex roots of 1, i.e. $x_0 = e^{2\pi i/n}$ and $x_j = e^{2\pi ij/n}$, then we get a Discrete Fourier Transform matrix. Defining $\omega_n = x_0$,

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \vdots & \omega_n & \dots & \omega_n^{n-1} \\ \vdots & \ddots & \omega_n^{(i-1)(j-1)} & \vdots \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{bmatrix} \tag{2.9}$$

This gives us

$$V^{-1} = \frac{1}{n}\left[\omega_n^{-(i-1)(j-1)}\right] \tag{2.10}$$

and so $V = \frac{1}{n}V^*$.