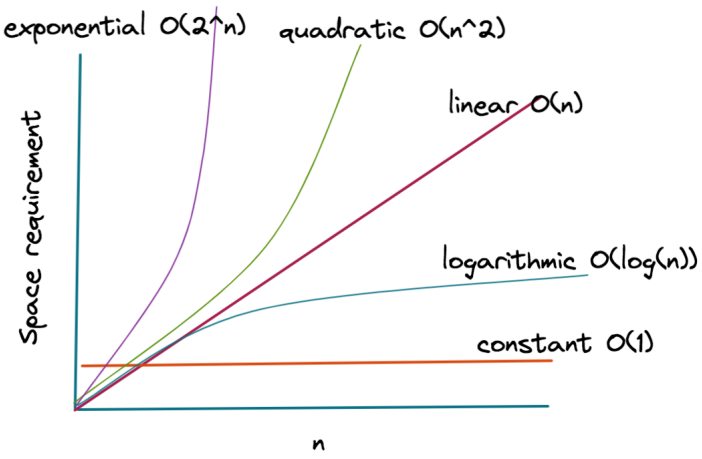


LINMA2111 - Discrete mathematics II

Simon Desmidt
Issambre L'Hermite Dumont

Academic year 2025-2026 - Q1

UCLouvain

Contents

Introduction

1.1 Sorting problems

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.

1.2 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.

1.3 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

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.

2.1 Complexity of an integer 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,

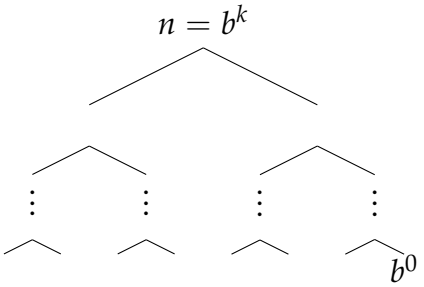
and get 5 multiplications. This gives a complexity of $\Theta(n^{\log_3(5)}) = \Theta(n^{1.46})$. Dividing in 4, 5, etc, we converge to a complexity of $\Theta(n)$ and this is the optimal complexity for the multiplication of two numbers of n digits. The problem is that the constant in front of the n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

2.1.1 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, $T(n) = \Theta(n^{\log_b(a)} \log(n))$;
- 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 $T(n) = \Theta(f(n))$;

2.1.2 Proof of the master theorem



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

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

[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. $\sim 10^{80}$).

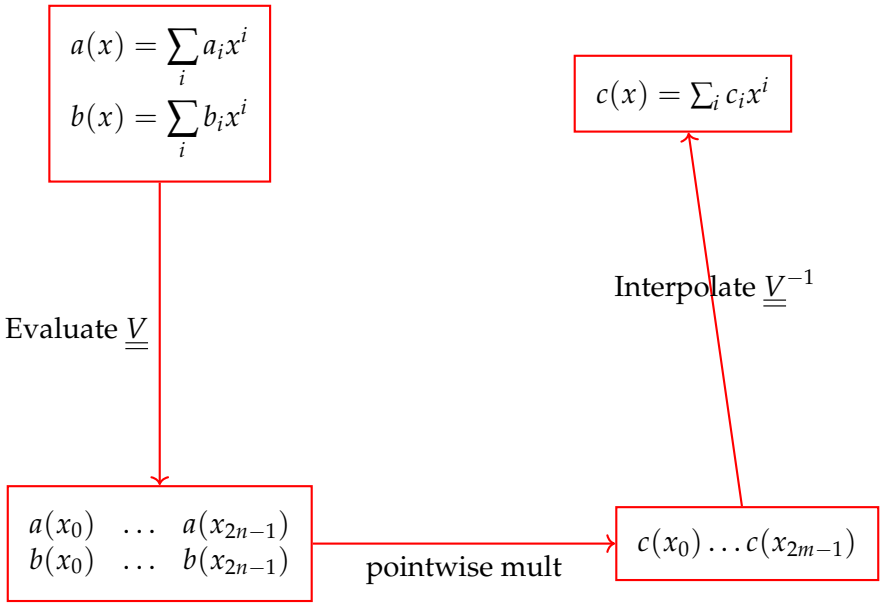
2.2 Multiplying polynomials

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:

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

where the matrix $\underline{\underline{V}}$ is called the Vandermonde matrix.

2.2.1 Convolution theorem



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

2.3 Discrete Fourier Transform

From the previous section, 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 $\omega_n = x_0$,

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

This gives us

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

and so $\underline{\underline{V}} = \frac{1}{n}\underline{\underline{V}}^*$. We conclude on

$$DFT(\mathbf{x}) = \underline{\underline{V}}\mathbf{x} \tag{2.11}$$

2.3.1 Fast Fourier Transform

The Fast Fourier Transform (FFT) algorithm uses the divide-and-conquer approach from above to compute the Fourier transform of a vector x. The exact algorithm is the following.
Assume that $n = 2^k$, $k \geq 1$, $\mathbf{x} = (x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1}) = \underline{\underline{V}}_n\mathbf{x}$. The explicit formula to calculate y_i, $i = 0, \ldots, n/2 - 1$ is

$$\begin{aligned} y_i &= \sum_{j=0}^{n-1} x_j \omega_n^{ji} = \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2ji} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{(2j+1)i} \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{ji} + \omega_n^i \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{ji} \end{aligned} \tag{2.12}$$

And in matrix form:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n/2-1} \end{pmatrix} = \underline{\underline{V}}_{n/2} \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{n-2} \end{pmatrix} + \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \underline{\underline{V}}_{n/2} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix} \tag{2.13}$$

And for the next half of y, we have a similar expression:

$$\begin{aligned} y_{i+n/2} &= \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2ji}\left(\omega_{n/2}^{n/2}\right)^j + \omega_n^{i+n/2} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{ji}\left(\omega_{n/2}^{n/2}\right)^j \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{ji} + (-1)\cdot\omega_n^i \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{ji} \end{aligned} \tag{2.14}$$

And, once again, in matrix form:

$$\begin{pmatrix} y_{n/2} \\ y_{n/2+1} \\ \vdots \\ y_{n-1} \end{pmatrix} = \underline{\underline{V}}_{n/2} \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{n-2} \end{pmatrix} - \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \underline{\underline{V}}_{n/2} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix} \tag{2.15}$$

Let us define three notations:

$$\begin{aligned} \mathbf{x}^{[0]} &= (x_0, \ldots, x_{n/2-1}) \\ \mathbf{x}^{[1]} &= (x_{n/2}, \ldots, x_n) \\ \underline{\underline{T}}_n &= \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \end{aligned} \tag{2.16}$$

Then,

$$\underline{\underline{DFT}}(\mathbf{x}) = \begin{pmatrix} \underline{\underline{DFT}}(\mathbf{x}^{[0]} + \underline{\underline{T}}_n \underline{\underline{DFT}}(\mathbf{x}^{[1]})) \\ \underline{\underline{DFT}}(\mathbf{x}^{[0]} - \underline{\underline{T}}_n \underline{\underline{DFT}}(\mathbf{x}^{[1]})) \end{pmatrix} \tag{2.17}$$

The time complexity of this algorithm is $\Theta(n \log(n))$.

→ Note: there exists other algorithm to compute the FFT, such as the non power-of-two n alorithm.

2.4 Matrix multiplication

Given $A, B \in \mathbb{Z}^{n \times n}$, we want to compute $C = A \cdot B$. The basic algorithm is in $\Theta(n^3)$, but we want a better one.

2.4.1 Straßen algorithm

Let us divide the matrices in blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{2.18}$$

The Straßen algorithm uses the same idea as in section section 2.1: we define 7 block matrices to reduce the number of products done.

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22})B_{11} \\ M_3 = A_{11}(B_{12} - B_{22}) \\ M_4 = A_{22}(B_{21} - B_{12}) \\ M_5 = (A_{11} + A_{12})B_{22} \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \end{cases} \implies C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix} \tag{2.19}$$

Which has a complexity of $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$. The lower bound for the complexity is $\Omega(n^2)$, as we need to make at least one operation per element of C, and the current best algorithm (galactic) is in $\Theta(n^{371339})$.

2.4.2 Matrix inversion

As we can assume intuitively, matrix multiplication and inversion are closely linked. Therefore, the complexity to compute both should be linked too. Let us call $M(n)$ the time complexity of multiplication of matrices of size n, and $I(n)$ the time complexity of inversion of a matrix of size n. Then,

$$D = \begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix} \implies D^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix} \tag{2.20}$$

This means that the complexity of inversion is an upper bound on the complexity of multiplication: $M(n) \leq I(3n) \leq 3^3 I(n)\ \forall n \geq n_0$.

This kind of inequality is called reduction. Given problems A and B, if A can be transformed in a problem B of size $f(n)$ in time $T_R(n)$, then

$$T_A(n) \leq T_R(n) + T_B(f(n)) \tag{2.21}$$

For example, the problem of finding the median of an array can be transformed into the problem of sorting the array.

2.4.3 Matrix inversion upper bound

Let us assume that $A = A^T$ and $A \succ 0$.

$$A = \begin{pmatrix} B & C \\ C^T & D \end{pmatrix} \Longrightarrow A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}CS^{-1}C^TB^{-1} & -B^{-1}CS^{-1} \\ -S^{-1}C^TB^{-1} & S^{-1} \end{pmatrix} \tag{2.22}$$

where $S = D - C^TB^{-1}C$ is the Schur complement of A. From Equation 2.21, $f(n) = \Theta(M(n))$ in our case, and so we have the following relation between inversion and multiplication:

$$I(n) = \mathcal{O}(M(n)) \tag{2.23}$$

$\rightarrow$ Note: the hypotheses that $A = A^T$ and $A \succ 0$ are not binding. If A is invertible but does not verify those conditions, we can work with AA^T that does, and we find the inverse of A with $A^{-1} = A^T(AA^T)^{-1}$.

Dynamic Programming

3.1 Two approaches

There are two approaches for a dynamic programming algorithm: bottom-up and top-down. In the bottom-up approach, we solve the subproblems first, and then use their solutions to solve bigger problems. In the top-down approach, we start from the main problem, and recursively solve the subproblems as needed.

- The main idea of bottom-up is to use memoization, i.e. storing the solutions of subproblems to avoid recomputing them.
- The main idea of top-down is recursion.

Dynamic programming is used to improve time complexity by trading time for space.

3.2 Generating functions

Let $\{a_k\}_{k=0}^{\infty}$ be a sequence. We associate the function $\sum_{k=0}^{\infty} a_k x^k$ to the sequence, on which we have addition, multiplication, differentiation, etc.
Example:

$$a_k = 1 \; \forall k \in \mathbb{N} \Longrightarrow f(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} = x \sum_{k=0}^{\infty} x^k + 1 = xf(x) + 1 \tag{3.1}$$