Introduction

1.1 Complexity

- **Definition of an algorithm:** is a sequence of computational steps that transforms input into output."

- **Definition of the complexity of an algorithm:** is the number of elementary operations it must perform to complete a calculation based on the size of the input data.

- **Definition of Algorithm Efficiency:** The efficiency of an algorithm is measured by the increase in computation time as a function of data size.

Data size

- The size of the data (or of the inputs) will depend on the coding of these inputs.

- For example, depending on the problem, the inputs and their sizes can be:
- **Elements**: number of elements
- **Numbers**: number of bits necessary for the representation of these;
- **Polynomials**: the degree, the number of nonzero coefficients;
- **Matrices**: max(m , n), m . n, m + n;
- **Graphs**: number of vertices, number of arcs, product of the two;
- **Lists**, tables, files: number of boxes, elements;
- **Words**: their length.

Calculation time

The calculation time of a program depends on several elements:

- The amount of data.
- Their encoding;
- The quality of the code generated by the compiler;
- The nature and speed of language instructions;
- The quality of programming;
- The efficiency of the algorithm.

- The complexity of the algorithms studied will depend neither on the computer, nor on the language used, nor on the programmer, nor on the implementation quite simply, it depends on a number of fundamental operations.
- The fundamental operations depend on the problem to be solved.

Problem	Fundamental operation
Searching for an item in a list	Comparison
Sorting a list, a file, ...	Comparisons, displacements
Multiplication of real matrices	Multiplications and additions
Addition of binary integers	Binary operation

Table 1: fundamental operations according to the problems

1.2 Cost of operations

For the time complexity, there are several possibilities:

• First possibility: calculate (as a function of n) the number of elementary operations (addition, comparison, assignment, etc.) required by the execution then multiply it by the average time for each of them;

• For an algorithm with essentially numerical calculations, count the costly operations (multiplications, root, exponential, etc.).

• Otherwise count the number of calls to the most frequent operation.

1.2.1 Sequential cost

- Sequence : $T_{sequence}(n) = \sum T_{elements\sim of\sim sequence}(n)$
- Alternative: **if C then J else K** : $T(n) = T_C(n) + \max(T_J(n), T_K(n))$
- Bounded iteration: **for i from j to k do B** : $T(n) = (k - j + 1)(T_{header}(n) + T_B(n)) + T_{header}(n)$ where the header is put the index assignment and the continuation test
- Unbounded iteration: **while C do B :** $T(n) = NB_{loops}.(T_B(n) + T_C(n)) + T_C(n)$ with NB_{loops} the number of loops which is evaluated by inductive method
- **Repeat B until C :** $T(n) = NB_{loops}.(T_B(n) + T_C(n))$

1.2.2 Recursive cost

The recursive method uses three steps at each level of the recursion:

• **Divide** the problem into a number of sub-problems;

• **Rule** over sub-problems by solving them recursively. If the size of the subproblem is small enough, it is solved immediately;

• **Combine** the solutions of the sub-problems to produce the solution of the original problem.

1.3 The different measures of complexity

- Let **A** be an algorithm, **n** an integer, **D_n** the **set** of inputs of size **n** , an input **d** **$\in D_n$** and **$cost_A$(d)** the number of fundamental operations performed by **A** with input **d** .
- Best case Complexity : $Min_A = \min\left\{\cos t_A(d) \,/\, d \in D_n\right\}$
- Worst case complexity: $Max_A = \max\left\{\cos t_A(d) \,/\, d \in D_n\right\}$
- The average case complexity : $Avg_A(n) = \sum_{d \in D_n} p_{proba\sim dist\sim law}(d).\cos t_A(d)$

Where p(d) is probabilistic distribution law of inputs.

for example: for uniform distribution law: $Avg_A(n) = \frac{1}{card(D_n)} \sum_{d \in D_n} \cos t_A(d)$

1.4 Comparison between algorithms

- to compare two algorithms, we will compare their growth rate or order of magnitude i.e. the execution time of the algorithm for a large number of data (inputs).

- An algorithm is more efficient than another if its worst-case running time has a lower order of magnitude.

- For the measurement of the complexity the notation O (Notation of Landau or big O) is often used.

1.4.1 Properties of big O

- If f(n) ∈ O(g(n)), then k.f(n) ∈ O(g(n))
- In particular, if a.b∈O(1) then a^{n+b}∈O(a^n)
- If e(n) ∈ O(g(n)) and f(n) ∈ O(h(n)) and if g(n) ∈ O(h(n)) then e(n)+ f(n) ∈ O(h(n))
- Particularly, $\mathrm{f(n)} = \sum_{d=0}^{D} a_d n^d \in O(n^D)$
- If e(n) ∈ O(g(n)) and f(n) ∈ O(h(n)), then e(n)f(n) ∈ O(g(n)h(n))

- Lower bound (**big Omega Ω**): f(n) ∈ Ω(g(n)) if g(n) ∈ O(f(n))
- Upper and lower bound (**theta** Θ): f(n) ∈ Θ(g(n)) if f(n)∈ O(g(n))∩Ω(g(n))

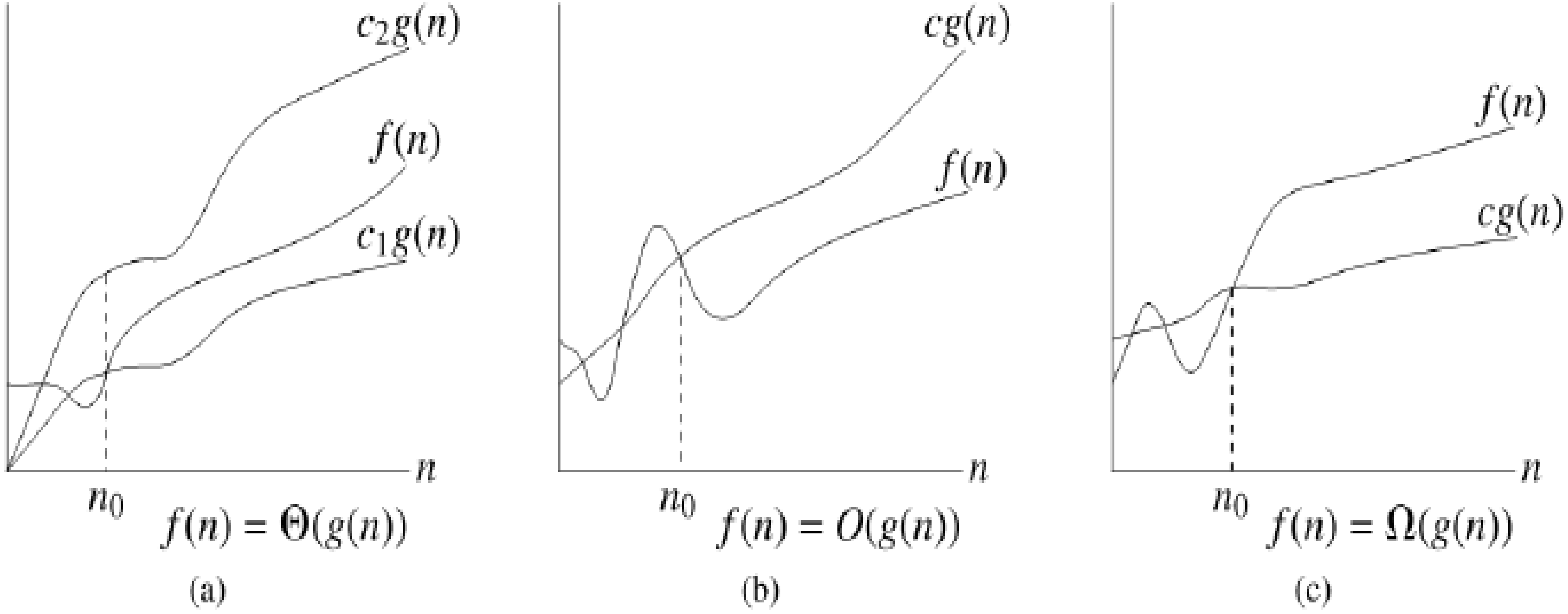

Illustration : Graphical definition of the different notations used in the calculation of complexity

1.5 Classification of algorithms

Complexity	Description
O(1) Complexity Constant time complexity	Execution does not depend on the number of input elements but always takes place in a constant number of operations
O(log(n)) Complexity Logarithmic complexity	The execution time increases slightly with n . This scenario occurs when the size of the problem is divided by a constant entity at each iteration.
O(n) Complexity Linear complexity	This is typically the case of a program with a loop from 1 to n and the body of the loop performs work of constant duration and independent of n .
O(n log(n)) Complexity n-logarithmic complexity	Occurs in algorithms where at each iteration the size of the problem is divided by a constant with each time a linear path of the data (ex: the "quick sort" sorting algorithm)
O(n^2) Complexity Quadratic complexity	Typically this is the case of algorithms with two nested loops each ranging from 1 to n and with the body of the internal loop which is constant.
O(n^3) Complexity Cubic complexity	A three nested loops algorithm.
O(n^p) Complexity Polynomial complexity	All the previous complexities are included in this one. An algorithm is said to be practicable, efficient if it is polynomial.
O(2^n) Complexity Exponential complexity	Algorithms of this kind are called "naive" because they are inefficient and unusable as soon as n exceeds 50.

1.5.1 Complexity of a problem

- The complexity of a problem A is the complexity of the best algorithm that solves A.

Problem	Algorithm name	Complexity
Array element access	Element access in integer array	0(1)
Search in sorted array	Binary search	0(log(n))
Search in unsorted array	Unsorted search	0(n)
Sorting array	Quick sort	0(n log(n))
	Bubble sort	$0(n^2)$
Multiplication of matrices	Square matrix multiplication	$0(n^3)$
Traveling salesman	Exhaustive search	$0(2^n)$