Chapter 1
Algorithmics and complexity

1. Introduction

- ***An algorithm is a finite sequence of elementary operations constituting a calculation scheme or a problem resolution scheme.***
- The execution time of an algorithm depends on the following factors:
  - **Program data:** generally when the size of the data processed by the program increases, its execution time also increases.

    Example: Sorting an array of 10 elements takes less time than sorting an array of 100 elements.
  - **The quality of the compiler (used language):**

    Example: Programs written in Java are generally slower than those written in C or C++.
  - **The used machine (speed, memory, . . .) :**
  - **The complexity of the algorithm itself.**

2. The complexity concept

Definition: *the complexity of an algorithm is a measure of the number of elementary operations that it performs on a data set. Complexity is expressed as a function (f(N), C(N), …) of the size of the data set .*

- **Elementary operation : is an operation whose execution time is independent of the size N of the data such that:**
  - **Assignment:** x ← 3
  - **Reading:** read (x)
  - **Writing:** write (x);
  - **Comparison (<, >, =,…):** if (x<8)
  - **Arithmetic operations ...etc:** Z= x+y

2. The complexity concept

- **Data size *N* :** is the number of the elements processed by the algorithm.

Examples:

- In the case of sorting an array, ***N* is the number of elements in the array.**
- In the case of calculating a term of a sequence, ***N* is the index of the term**
- In the case of calculating the sum of the elements of a matrix of n*m elements, **the data size is n*m.**

2. Concept of complexity

Objective

- We seek to measure the complexity of an algorithm independently of the machine and the used language, i.e. only as a function of the size of the data ***N*** that the algorithm must process.
- The objective is therefore to find the function (f(N), C(N)…) which expresses the number of operations performed by the algorithm as a function of the data size N.

Example:

- **F(N) = 3N+6**
- **C(n)= $4N^2$**

2. Concept of complexity

Example 1:

Algorithm : sum

N, i, S: integer

Begin

i ←1;

S ←0;

While (i <=N) do

S ← S+i;

i ← i+1;

End while

Write(s);

END

This algorithm calculates the sum of the first N integers.
The data size is **N**.
The sum algorithm performs:

2 Assignments (i ← 1, S←0))

N + 1 comparison (i<=N);
N additions (s+i);
N assignments (S← S+i);
N additions (i+1);
N assignments (i←i+1);

1 display (write(s)):

5N+4 elementary operations

$f(n) = 5N+4$

2. Concept of complexity

1.3 Why calculate the complexity of algorithms:

1. Calculate the time and memory space required to execute a program.

2. Check if an algorithm is executable in a reasonable time.

3. Compare two algorithms solving the same problem to choose the best one.

4. Check whether an algorithm is optimal. Check whether there is a better performing algorithm or not .

3. Rules for calculating the complexity of an algorithm

1) **Simple instruction:** the number of elementary operations in the instruction.

   Example: in the instruction **y ← 3+5*x;** we have 3 elementary operations, an assignment, an addition and a multiplication.

2) **Instruction Sequence:** The complexity of an instruction sequence is equal to the sum of the complexities of the instructions in the sequence.

   Example:

```
Read(x);
Read(y);
z ←x+y;
Write (z);
```

5 operations

3. Rules for calculating the complexity of an algorithm

3) The complexity of a conditional

The complexity of an <if – then – else> statement equals the sum of the complexity of evaluating the condition and the greater of the complexity of the then block and the else block.

If (*condition)* then

……. — **Then block**

Otherwise

……. — **Else block**

Finished

Complexity = **C1+ max(C2,C3)**

Where :

- **C1** is the number of operations performed for the evaluation of the condition.
- **C2** is the number of operations executed if the condition is true (the **Then block**)
- **C3** is the number of operations executed if the condition is false (the Else block)

3. Rules for calculating the complexity of an algorithm

4) **Loop complexity: the complexity of a loop is equal to the sum, over all iterations, of the complexity of evaluating the exit condition and the complexity of the statement block in the body of the loop.**

Example

```
Read(N); i=1;
While (i<=N )do
        S=s+i;
        i=i+1;
End While
```

- In this loop we have N iterations
- The comparison i<=N is performed N+1 times (N iterations of the loop plus once to exit the loop)
- In the instruction block we have 4 instructions each executed N times.
- The complexity of the loop is therefore 5N+1

3. Rules for calculating the complexity of an algorithm

5) Subprograms (functions and procedures)

The complexity of calling a subprogram equals the sum of the complexities of its instructions.

Example:

Algorithm : sum

N, i, S: integer

Function sum (N: integer): integer

i, S: integer

Begin

S ←0; **Op1**

For i ←1 to N do **Op2, Op3, Op4, Op5**

S ← S+i; **Op6** **Op7**

End for

Return (s); **Op8**

End

Operation	Number
Op1	**1**
Op2 (i ←1)	**1**
Op3 (i ≤N)	**N+1**
Op4 (i ←)	**N**
Op5 (i+1)	**N**
Op6	**N**
Op7	**N**
Op8	**1**
Total	**5N + 4**

3N+2

3. Rules for calculating the complexity of an algorithm

```
N: integer;
Begin
                          Op1
        Read (N);
        Write (Sum (N));
End          Op2     Op3
```

Operation	Number
Op1	1
Op2	1
Op 3 (Call of Sum (N))	5N+4
F (N)	5N +6

4. Complexity of recursive algorithms

- The complexity of a recursive algorithm is done by reasoning by recurrence. We must folw the following steps:

Step1: Find the relation that explains the complexity of the algorithm by calculating the complexity of some values of N.

Step2: Consider that the relation is correct for N-1.

Step3: Prove the relation for N.

4. Complexity of recursive algorithms

Example

- Let's consider the following recursive function:

```
factorial function(n:integer):integer
begin
If n=0 then      Op1
    returns 1;    Op2
Else
    returns (n* factorial(n-1))
      Op3     Op4            Op5
end if
END
```

4. Complexity of recursive algorithms

Let C(n) be the running time required for a call to factorial(n).

Step 1:

4 operations

$C(0) = Op1 + Op2 = 2$

$C(1) = Op1 + Op3 + Op4 + Op5 + C(0) = b + C(0)$ ($b = Op1 + Op3 + Op4 + Op5$)

$C(2) = Op1 + Op3 + Op4 + Op5 + C(1) = b + b + C(0) = 2b + C(0)$

$C(3) = Op1 + Op3 + Op4 + Op5 + C(2) = b + 2b + C(0) = 3b + C(0)$

.

Step 2: we consider that the complexity for N-1 is $C(n-1) = (n-1)\, b + C(0)$

Step 3: by the execution of the function we toruve that :

$C(n) = Op1 + Op3 + Op4 + Op5 + C(n-1) = b + (n-1)\, b + C(0) = n.b + C(0)$

$C(n) = 4n + 2$

4. Complexity of recursive algorithms

Exercise: Let the following product function be that calculates the product x*N. Evaluate its complexity.

```
Function product(x, N : integer): integer
Begin
    If N=0 then
            returns 0;
    Else
            If N=1 then
                    return x;
            Else
                    returns (x+ product(x, N-1))
            end if
    end if
END
```

5. Complexity at best and worst

Example: *sequential search for an element in an array of n integers*

```
Function search (n, x: integer, tab: array of integers): boolean
    i: integer; b: boolean ;
Begin
        i ← 1; b ← false;
 While (i <= n) and (b=false) do
    if (tab[i] = x) then
            b=true;
    Endif
    i ←i+1;
 Endwhile
    return b;
End
```

- *The complexity parameter is the size of the input array.*
- *The number of loops varies depending on whether x is in the array or not, and where x is present.*

5. Complexity at best and worst

- ***If x is in the first box of the table:*** *1 loop with the condition b=true* O(1)
- ***If x is in the second box of the table:*** *1 loop with the b= false and 1 loop with b= true* O(2) …
- ***If x is in the last box of the table:*** *N loop turns with b= false and 1 loop turn with b= true.* O(n)
- ***If x is not in the array:*** *N loops with condition b= false.* O(n)
- The **time complexity** of the sequential search algorithm is: $\boxed{O(n)}$
- Because in the worst (and average) case, the algorithm looks at each of the n elements once.

5. Complexity at best and worst

When, for a given value of the complexity parameter, the execution time varies according to the input data, we can distinguish:

- **Worst-case complexity:** maximum execution time, in the worst case.
- **Best-case complexity:** minimum execution time, in the best case (in practice, this complexity is not very useful).
- **Average complexity:** execution time in a median case, or average of execution times.

Most often, **we use worst-case complexity**, because we want to limit the execution time.

6. Landau notation (*O*)

- **What is Landau / Big O notation?**
- It's a way to **measure how fast an algorithm grows** (how much time it takes) when the **input size (n)** becomes very large.
- 👉 Think of it like this:
- If an algorithm takes **twice as long** when you double n, it's **O(n)** (linear).
- If it takes **four times as long**, it's **O(n²)** (quadratic).
- If it doesn't really grow, it's **O(1)** (constant time).
- We ignore details like exact numbers or small differences (like +3 or ×2). We only keep the *big picture* of how fast it grows.

6. Landau notation (*O*)

- **Let the function T(n) represent the evolution of the execution time of a program P as a function of the number of data n. For example:**

$$T(n) = a\, n^2$$

- **In the example above, we say that the complexity of P is $O(n^2)$. This means that there exists a positive constant C such that for n sufficiently large we have:**

$$T(n) \leq Cn^2$$

- **This notation gives an upper bound on the number of operations executed (execution time) by the program P. A program whose complexity is O(f(n)) is a program which has f(n) as a function of growth of the execution time .**

6. Landau notation (*O*)

Example:

Let the function *f* (n) represent the execution time of a program P. *f* (n) = $n^3 + 3n + 6$

if n=10 and c=2 we obtain:

$$f(n) = 10^3 + 3*10 + 6 = 1036 \leq 2 * 10^3 = 2000$$

The complexity of P is said to be O(n^3).

Because there exists a positive constant c (c=2) such that for n sufficiently large (N=10) we have:

$$n^3 + 3n + 6 \leq cn^3$$

7. Algorithmic complexity classes

1. **T(n) = O(1), constant time : execution time independent of the size of the data to be processed.**

2. **T(n) = O(log(n)), logarithmic time : such complexity is usually encountered when the algorithm breaks a large problem into several smaller ones, so that solving just one of these problems leads to solving the initial problem.**

3. **T(n) = O(n), linear time : this complexity is generally obtained when constant time work is performed on each input data.**

4. **T(n) = O(n.log(n)) : the algorithm splits the problem into several smaller subproblems that are solved independently. Solving all of these smaller problems provides the solution to the initial problem.**

7. Algorithmic complexity classes

5. **$T(n) = O(n^2)$, quadratic time : appears in particular when the algorithm considers all pairs of data among the n inputs (e.g. two nested loops)**

   Note: $O(n^3)$ cubic time, $O(n^k)$ polynomial

6. **$T(n) = O(2^n)$, exponential time : often the result of a brutal search for a solution.**

7. Algorithmic complexity classes

Execution times depending on data size.

temps	constant $\theta(1)$	logarithmique $\theta(\log_2 N)$	linéaire $\theta(N)$	$\theta(N \log_2 N)$	polynomial $\theta(N^k)$ quadratique $(k=2)$	$k=3$	exponentiel $\theta(2^N)$
$N = 100$	$1\mu s$	$6.6\mu s$	0.1 ms	0.6 ms	10 ms	1s	10^{16} ans
$N = 1000$	$1\mu s$	$9.9\mu s$	1 ms	9.9 ms	1 s	16.6 min	∞
$N = 10^4$	$1\mu s$	$13.3\mu s$	10 ms	0.1 s	100 s	11.5 jours	∞
$N = 10^5$	$1\mu s$	$16.6\mu s$	0.1 s	1.6 s	2.7 heures	31 ans	∞
$N = 10^6$	$1\mu s$	$19.9\mu s$	1 s	19.9 s	11.5 jours	$3\ 10^4$ ans	∞

NB: figures for a machine capable of performing 10^6 operations per second

How to determine the class ofcomplexity ?

- **What we're trying to find ?**
- **We want to understand how fast the function grows when n becomes very large.**

  That's exactly what Landau notation (Big O) measures.

determining the class of complexity

Step 1: Identify all the terms

- Look at all the parts of the function.
  Example: $F(n) = 3n + 2\log n + 10$
- We can see:
  - a linear term: $3n$
  - a logarithmic term: $2\log n$
  - a constant term: 10

Step 2: Compare their growth rates

- We compare how fast each term grows as n increases.
- ➡️ Generally, growth rates follow this order:
- $\log n < n < n\log n < n^2 < n^3 < 2^n < n!$

- Keep the dominant term :
- The dominant term is the one that grows fastest when n becomes large.

  It will eventually "drown out" all the smaller ones.
- So, for $F(n) = 3n + 2\log n + 10$: 👉 The term $3n$ dominates.

  The others become negligible when n is large.

Step 3: Ignore constants

- **In Landau notation, constants don't matter.**
- **(We ignore multiplications by 2, 3, 10, etc.)**

  Because we care about *the growth shape*, not the exact number.
- **So: $3n + 2\log n + 10 \Rightarrow O(n)$**

General rule (to remember)

- When a function is a sum of several terms, the asymptotic complexity is determined by the term that grows the fastest.
- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 \Rightarrow O(n^k)$

- Full example with our function C(n) :
- $C(n) = n\log n + 8n + 3$
- List the terms: $n\log n, 8n, 3$
- Compare their growth rates: $\log n < n < n\log n$
- The largest is $n\log n$
- We Keep only that one $\rightarrow O(n\log n)$