"Advanced Algorithmics"

P. Laroque

september 2014*

1 Introduction to complexity

1.1 Why should I compute complexity?

A programming problem often has several different solutions. Depending on the chosen criteria, one of them will be considered as the "best" solution. One of those criteria – important, as we are going to discover – is the complexity of the underlying algorithm. This complexity is generally a two-dimension concept:

- complexity in time, which tells about the order of magnitude of the time needed to complete the program, based on certain (input) parameters;
- complexity in space, which tells about the order of magnitude of the memory space needed to execute the program.

Here, we focus on the first aspect, even if the two are generally tightly coupled (as we'll see during one of the labs).

Let's take a simple example as an introduction: the search of an information in a set (represented by an array) sorted according to a given criterion that we'll call the *key*. One can immediately think of two ways to find the information needed:

1. Start a sequential search form the beginning of the array, taking advantage of the fact that it is sorted to stop as soon as we find a key

*$Id: algo-en.lyx 1966 2014-10-06 09:13:43Z phil $

value greater or equal to the one we're looking for. If the key is the one we searched, then we have found our information, else the information is not in the array.

2. We compare the desired key to that of the middle element in the array. In case of equality, we stop on success; if the searched key is less than (*resp.* greater than) the middle key, we just have to look in the first (*resp.* second) half of the array. We can iterate the same process until we deal with a 1-element array, which immediately gives the answer.

Informally, the reader probably "feels" that the second approach is (much) more performant than the first:

- in the first case, each test reduces the size of the search by 1 element;
- in the second case, each test reduces it by one half.

One can see that, when searching for an information in a random (but sorted) array, the first approach will statistically lead to a number of tests proportional to the number of informations stored in the array: in average, we'll have to examine half of the array to get the answer.

However with the second approach, we divide the size of the search space by two at each test, so it's easy to see that the number of tests to lead is proportional to the *logarithm* of the array size.

Let's try to quantitatively estimate the difference: if we consider a big quantity of information, for instance a dictionary (10^6 words and definitions), and that a test for word equality needs one millisecond of computing time, the first method gives an answer after $5.10^5.10^{-3} = 500s$, or about eight minutes. The second needs $10^{-3}.\log_2 10^6 ms$, which is about 20 milliseconds...

The goal of this first section is to provide several basic notions to make such estimations in simple cases. We first come back to simple notions as the limit of a function or sequence as the parameter tends to plus infinity, then we present a system of notation (know as "Landau notation") and the main classes of complexity used for algorithms.

We then introduce rules to compute complexity in the case of iterative algorithms, then – later in this document, see 1.5 – in the case of recursive algorithms. We then give a few examples to illustrate the usefulness of this type of computation.

1.2 Function complexity - Landau notation - classes of complexity

By "function complexity" we mean the behaviour of the function when its parameter (generally an integer – to represent, as in the introductory example, the size of the search space; that is why, in the rest of this section, we use the word "sequence" rather than "function") continuously increases to tend to plus infinity: we talk about asymptotic behaviour of the function / sequence.

1.2.1 The concept of limit

A sequence of real numbers is said to converge to a finite limit l near infinity, which is noted

$$\lim_{n\to\infty} u_n = l$$

if and only if

$$\forall \varepsilon > 0, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n > n_0 \Rightarrow |u_n - l| < \varepsilon$$

it is said to converge to plus infinity, noted

$$\lim_{n\to\infty} u_n = +\infty$$

if and only if

$$\forall \alpha > 0, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n > n_0 \Rightarrow u_n > \alpha$$

In the real life, almost all sequences corresponding to actual algorithms fall in the second category. The tools that follow help differentiate groups (classes) in this category.

1.2.2 Landau Notations

Let u and v be two sequences of positive real numbers (we will actually only consider increasing sequences, which are the only ones that make sense in our context). We say that v *dominates* u, or that v *is an upper bound almost everywhere for* u, noted $u \in O(v)$, if and only if

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n > n_0 \Rightarrow u_n \leq c.v_n$$

u is also said to be *asymptotically dominated by* v, noted $v \in \Omega(u)$.

u and v are said to be *of the same order*, noted $u \in \Theta(v)$, if and only if

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n > n_0 \Rightarrow c_1.v_n \leq u_n \leq c_2.v_n$$

For instance any polynomial of degree n is of the same order as the monomial x^n.

1.2.3 Properties

1. reflexivity for O and Θ: $u \in O(u)$ and $u \in \Theta(u)$
2. symmetry for Θ:
$$u \in \Theta(v) \Rightarrow v \in \Theta(u)$$
3. transitivity for O and Θ:
$$\left.\begin{array}{r} u \in O(v) \\ v \in O(w) \end{array}\right\} \Rightarrow u \in O(w)$$
(same for Θ)
4. $\forall \lambda > 0, \lambda u \in O(u)$ (same for Θ)
5. $u + v \in O(max(u, v))$ (same for Θ)
6. $\left.\begin{array}{r} u_1 \in O(v_1) \\ u_2 \in O(v_2) \end{array}\right\} \Rightarrow u_1.u_2 \in O(v_1.v_2)$ (same for Θ)
7. $u - v \geq 0 \Rightarrow u - v \in O(u)$
8. $\left.\begin{array}{r} u - v \geq 0 \\ \lim_{n \to +\infty} \frac{v}{u} = 0 \end{array}\right\} \Rightarrow u - v \in \Theta(u)$

One can notice that the first three properties make Θ be an equivalence relation. We can thus talk about equivalence classes of complexity.

Proof for the next three prpoerties is left as an exercise. If u and v are two real positive sequences:

$$\lim_{n \to +\infty} \frac{u(n)}{v(n)} = a \neq 0 \Rightarrow u \in \Theta(v)$$

$$\lim_{n\to+\infty} \frac{u(n)}{v(n)} = 0 \Rightarrow u \in O(v), u \notin \Theta(v)$$

$$\lim_{n\to+\infty} \frac{u(n)}{v(n)} = +\infty \Rightarrow u \in \Omega(v), v \notin \Theta(u)$$

A few comments:

- In the first case, when a is 1 the two sequences are said to be *asymptotically equivalent*, noted $u \sim v$.
- In the second situation, u is said to be *negligible compared to* v, noted $u \in o(v)$
- Those three properties allow - most of the time - for a classification of positive sequences. In the case when quotient $\frac{u(n)}{v(n)}$ doesn't have a limit when n tends to infinity, one needs to go back to the definitions of O and Θ to compare the two sequences. In some situations, one can determine the result using the "Hospital rule" (the limit of a rational function is the limit of the quotient of its higher-degree terms) and then apply the properties defined above. For instance, if we consider

$$\begin{cases} u_{2n} = 4n, u_{2n+1} = 2n+1 \\ v_n = 2n \end{cases},$$

one can easily check that $\frac{u(n)}{v(n)}$ has no limit, but nevertheless we still get $u = \Theta(v)$ because $\forall n, \frac{1}{2}v(n) \leq u(n) \leq v(n)$.

1.2.4 Comparison scales

To get an idea of the behaviour of a sequence in the neighborhood of infinity, one can try to compare it with a set of *reference sequences*.

Definition: A *comparison scale* is a set S of real positive sequences such that

1.

$$\forall u \in S, \exists n_0 \in \mathbb{N}, n > n_0 \Rightarrow u(n) > 0$$

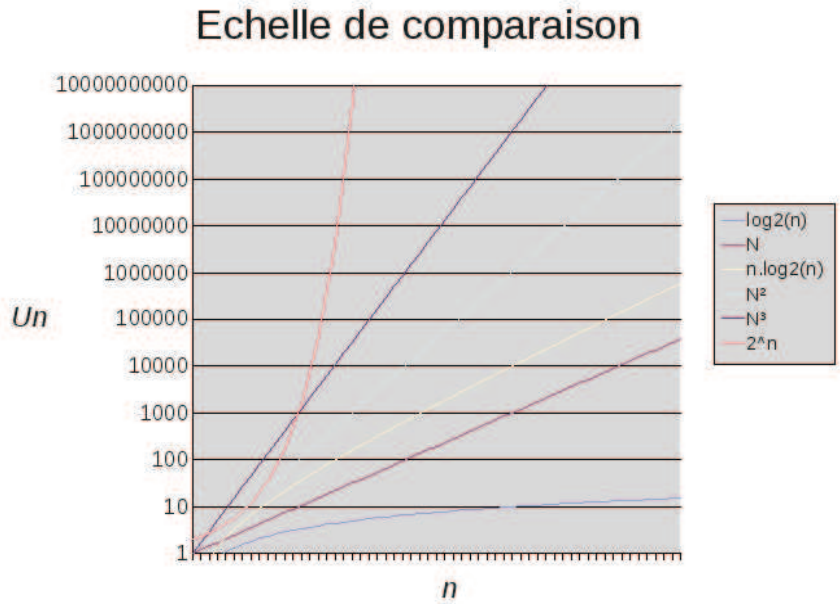

Figure 1: A classical, widely used comparison scale

2.

$$\forall u \in S, u \neq 1 \Rightarrow \lim_{n \to +\infty} u(n) \in \{0, +\infty\}$$

3.

$$\forall u, v \in S^2, u \neq v \Rightarrow \begin{cases} u = o(v) \\ or \\ v = o(u) \end{cases}$$

Exemples: A few examples of comparison scales:

- The set of sequences $n^\alpha, \alpha \in \mathbb{R}$
- The set of sequences $n^\alpha \log(n)^\beta, (\alpha, \beta) \in \mathbb{R}^{\not\leq}$
- The set of sequences $\exp(\alpha n^\beta).n^\gamma.\log(n)^\delta, (\alpha, \beta, \gamma, \delta) \in \mathbb{R}^4, \beta > 0$

The most commonly used scales in this course are the first two and the set of sequences of the form $2^{n^k}, k \in \mathbb{N}$. Figure 1 shows such a classical scale.

1.3 Context of complexity evaluation

In the general case, the algorithms used to solve a problem involve two categories of resources: CPU time and memory space. An efficient algorithm tries to minimize both; however, most of the time, it happens that winning on the one side means losing on the other: one speak of the *time / space compromise*. Almost all of the algorithms we are going to study here focus on the analysis of time complexity, letting aside space complexity (or roughly evaluating it). When not explicitely stated, we talk about time complexity from now on in this document.

Precise computation of the complexity is not possible in general; one has to work based on a set of simplifying hypotheses. This course relies on the following assumptions:

1. One explicitly chooses one or more operations considered as *atomic* or *fundamental*, i.e. executing in what is considered as a unit of time. Most of the time that concerns scalar assignment and/or comparison. This will then be the unit of measure for complexity.

2. Algorithms usually contain branches (conditional clauses) which forbid the exact static computation of the complexity: branches can be of different cost. To simplify, we usually choose one of the following three options: *(i)* the *best* case, in which we assume that the program systematically chooses the lower-cost branch, *(ii)* the *worst* case, in which we assume it always chooses the higher-cost branch, and *(iii)* the *average* case, which involves a (probabilistic) evaluation of the cost of each branch.

The interest of computing complexity in the average case is obvious, but it's of course (and by far) the most difficult evaluation to make. That is why one often chooses the worst case, which provides an upper bound for the time needed to complete the program execution.

The rest of this section defines rules to compute complexity for iterative algorithms. Recursive algorithms will be studied later in this document, particularly when we deal with tree-like structures.

1.4 Evaluating complexity in an iterative context

Computation rules in the iterative context are rather simple:

- the complexity for a sequence of statements is the sum of the complexity of each statement of the sequence;
- in case of a loop, if we call C_i the complexity of iteration with value i for the loop index and C that of the whole loop, we get

$$C = \sum_i C_i$$

- in case of a conditional of type
  `if (cond) I1 else I2`
  where `I1` and `I2` are two algorithms we get

$$C(cond) + \inf(C(I_1), C(I_2)) \leq C(if) \leq C(cond) + \sup(C(I_1), C(I_2))$$

  (lower value corresponds to the best case, higher value to the worst case);
- the complexity for a (iterative) function call is that of the statements in the function body;
- the complexity for a recursive function must be computed by means of a *recurrence equation* (see 1.5).

1.4.1 First simple example

Let's consider a very simple example, the search for the highest value in an array of integers. The java code for the function is something like that:

```
int getMax(int[] a) {
  int max = a[0];                     (1)
  for (int i = 1; i < a.length; i++) (2)
    if (a[i] > max)                   (3)
      max = a[i];                     (4)
  return max;                         (5)
}
```

Let's choose two fundamental operations, assignment (a) and comparison (c) between integers.

Based on the above rules we are able to compute the complexity for each statement and sum them to get the final result. The algorithm is composed of

a sequence of three statements (1), (2) and (5), since (3) and (4) are internal to the `for` loop (2).

We can immediately evaluate value for lines (1) and (5): $C_1 = 1a + 0c = a, C_5 = 0a + 0c = 0$.

To compute C_2 one needs C_3, which in turns needs C_4. This value is immediate: $C_4 = 1a + 0c = a$.

Since (3) is a conditional, we have to choose the context of its evaluation. Let's start simple, with the worst case. If the condition is false, we have no assignment to execute. In the other case we have one assignment to make. In both cases we must make a comparison. We then get, in the worst case:

$$C_3 = c + a$$

In the average case, in a random array we have a probability 0.5 for the test to hold true:

$$C_3 = c + \frac{1}{2}a$$

Since (2) is a loop, we have a sum to compute. We see that the upper bound of the sum is not constant (it's one less than the array size): the result will be a function of the size. This is almost always the case: complexity depends not only on the choice of fundamental operation(s), but also on parameters related to the *input data* for the algorithm. We then have to talk about $C(n)$ (where n represents the size of the array) and not simply about C to refer to the complexity of the function.

In all cases, we have

$$C_2(n) = \sum_{i=1}^{n-1} C_3$$

so, in the worst case

$$C_2(n) = \sum_{i=1}^{n-1} (c + a) = (n-1)(c+a)$$

and in the average case

$$C_2(n) = \sum_{i=1}^{n-1} (c + \frac{1}{2}a) = (n-1)(c+\frac{a}{2})$$

function complexity is thus, in the worst case,

$$C(n) = C_1 + C_2(n) + C_5 = a + (n-1)(c+a) = (n-1)c + n.a$$

and in the average case

$$C(n) = a + (n-1)(c + \frac{a}{2}) = (n-1)c + \left(\frac{n+1}{2}\right) a$$

1.5 Introduction to recurrence equations

We saw (see section 1.4) how to evaluate the complexity of an iterative algorithm. Rules described so far are however insufficient in a recursive case. One has then to work in two steps:

1. Determine a recurrence equation giving *(i)* the complexity of the algorithm in the simplest case(s) – the initial states of the system – and *(ii)* the process to compute the complexity in a given state when we know it in a state slightly closer to one of the simple cases previously determined. It's the same approach as proof by induction, be it about integers or about structures like trees. We get a system of the form

$$\begin{cases} C_{E_{init}} &= \exp_{init} \\ C_E &= f\left(E, C_{E'}\right) \end{cases}$$

   where E' is a state "closer" to initial state E_{init} than E is.

2. Solve this system of (recurrence) equations.

In the general case, the second step is very complex and beyond the scope of this course. Nonetheless, we give in this section several examples representing simple cases that can pop up when studying classical algorithms, in which the "state" is often represented by only one or two numerical values.

1.5.1 Example of logarithmic complexity

We can frequently expect a logarithmic complexity when each recursive call (supposed unique inside the function) divides the size of the search space by a constant factor. It is for instance the case in the binary search inside a