Foundations of Computer Science
Logic, models, and computations

Chapter: Basic of complexity analysis of algorithms

Course INF412

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of July 16, 2023

Basic of complexity analysis of algorithms

The previous discussions have been concerned with the existence or non-existence of algorithms for solving a given problem, but ignoring an essential practical aspect: the resources needed for its execution, that is to say for example the *computation time* or the *memory* that is required on the machine for its execution.

The objective of the next chapter is to focus on one resource, the *computation time*. In the later chapters, we will evoke other resources such as memory space. We could also talk about parallel time, that is to say the time required on a parallel machine.

Let us however start by better understanding the difference between previous chapters and the following chapters. In previous chapters, we were talking about *computability*, that is to say we asked about the existence of an algorithmic solution to a given problem. We will now focus on *complexity*: That is to say, we focus now on decidable problems, i.e., problems for which an algorithm is known. The question is to decide whether there is an *efficient* algorithm.

This leads first to the question what one calls *efficient*, and how this efficiency can be measured. First of all, we think it is important that our reader has clear ideas on what is called the complexity of an algorithm and the complexity of a problem, which are not the same concept.

Remark 1 *Even if we will talk about average case complexity in this chapter, we will not need this in the coming chapters : We introduce it here mainly to explain why average case complexity is not used much in practice (at least in complexity theory).*

1 Complexity of algorithm

In this chapter, we mostly consider a (decision or general) problem $\mathscr{P}$ for which one knows an algorithm $\mathscr{A}$: This algorithm is known to be correct, and is terminating. It takes as input some data d, and it produces as its output a result $\mathscr{A}(d)$ by using some resources (so we will only talk about decidable problems for decision problems).

Example 1 *The problem $\mathscr{P}$ could for example consist in determining if a given number v is among a list of nb numbers.*

It is clear that one can come up with an algorithm $\mathscr{A}$ to solve this problem. For example:

- *one uses a variable res initially set to* 0*;*
- *one scans the list, and for each element:*
  - *one checks if this element is the number v:*
    - *it this is the case, one sets the variable res to* 1*;*
- *at the end of the loop, on returns res.*

This algorithm certainly is not the most efficient that one can think of. First, we could stop as soon as one sets res to 1*, since the answer is known. Furthermore, one can clearly do something different, such as a* dichotomic search *(a recursive algorithm) if one knows that the list is sorted.*

1.1 First considerations

One always measures the efficiency, that is to say the complexity of an algorithm in terms of an *elementary measure* with integer value: This can be the number of instructions executed, the size of the memory that is used, the number of comparisons made, or any other measure.

One just needs that, given an input d, one knows how to associate the value of this measure, denoted by $\mu(\mathscr{A},d)$, to the algorithm $\mathscr{A}$ on input d. For example, for a sorting algorithm working with comparisons, if the elementary measure μ is he number of comparisons done, $\mu(\mathscr{A},d)$ is the number of comparison performed on the input d (a sequence of integers) by an algorithm $\mathscr{A}$ to produce the result $\mathscr{A}(d)$ (the sorted list).

The function $\mu(\mathscr{A},d)$ depends of $\mathscr{A}$, but also of the input d. The quality of an algorithm $\mathscr{A}$ is hence not an absolute criterion, but a quantitative function $\mu(\mathscr{A},.)$ from the inputs to the integers.

1.2 Worst case complexity of an algorithm

In practice, to understand the function $\mu(\mathscr{A},.)$, one often aims to evaluate the complexity for the inputs of a given *size*: There is often a function size that maps to every input data d, an integer size(d), that corresponds to some natural parameter. For example, this function can be the number of elements for a sorting algorithm, the size of a matrix for computing the determinant, or the sum of the lengths of the strings for a concatenation algorithm.

To go from a function from the inputs to the integers to a function from the integers (the sizes) to the integers, one then considers the *worst case complexity*: The

complexity $\mu(\mathscr{A}, n)$ of algorithm $\mathscr{A}$ on inputs of size n is defined as

$$\mu(\mathscr{A}, n) = \max_{d \text{ input with } \text{size}(d)=n} \mu(\mathscr{A}, d).$$

In words, the complexity $\mu(\mathscr{A}, n)$ is the worst complexity observed on inputs of size n.

By default, when one talks about the *complexity of an algorithm*, one considers the worst case complexity as above.

If one does not know more on the inputs, there is no real hope to do better than this pessimistic view of life, and evaluating the complexity in the worst case (the best case has no particular practical meaning, and in this context, pessimism is far more significant).

1.3 Average case complexity of some algorithm

In order to say more, one must know more about the inputs. For example, that they are distributed according to some probabilistic distribution.

In that case, we can then talk about *average case complexity*: The average case complexity $\mu(\mathscr{A}, n)$ of algorithm $\mathscr{A}$ of inputs of size n is defined as

$$\mu(\mathscr{A}, n) = \mathbb{E}[\mu(\mathscr{A}, d) | d \text{ inputs with } \text{size}(d) = n],$$

where $\mathbb{E}$ denotes expectation (the average).

This is equivalent to

$$\mu(\mathscr{A}, n) = \sum_{d \text{ inputs with } \text{size}(d)=n} \pi(d)\mu(\mathscr{A}, d),$$

where $\pi(d)$ denotes the probability of having this input of size n.

In practice, if the worst case might be rare, and the average case analysis may seem more appealing.

But first, it is important to know that one cannot talk about expectation/average without a probability distribution on the inputs. This implies on the one hand that the distribution of the data given as input must be known, something which is very delicate to predict or estimate in practice. How to anticipate for example the lists that will be given to a sorting algorithm?

One sometimes makes the hypothesis that the inputs have same probability (when this makes sense, as in the case where one wants to sort n numbers between 1 and n) but this is often very arbitrary, and not totally justifiable.

On the other hand, as we will see, computing the average case complexity is often more delicate to deal with than worst case analysis.

2 Complexity of a problem

One can also talk about the *complexity of a problem* which provides a way to talk about the optimality of an algorithm to solve a given problem.

One fixes a problem $\mathscr{P}$, for example, the problem of sorting a list of integers. Let $Alg(\mathscr{P})$ be the class of all algorithms that solves $\mathscr{P}$: an algorithm $\mathscr{A}$ of $Alg(\mathscr{P})$ is an algorithm that answers to the specification of the problem $\mathscr{P}$: For every input d, it produces a correct answer $\mathscr{A}(d)$.

The complexity of the problem $\mathscr{P}$ is defined as the infinimum[1] of the complexity of the algorithms of $Alg(\mathscr{P})$. Consequently, an algorithm $\mathscr{A}$ is *optimal* if its complexity is equal to the optimal complexity of $Alg(\mathscr{P})$: That is to say, there is no other algorithm $B \in Alg(\mathscr{P})$ with a smaller complexity. We write $\mu(\mathscr{P}, n)$ for the complexity of some optimal algorithm[2] on inputs of size n .

In other words, we do not only make the inputs of size n vary, but also the algorithm. One considers the best algorithm that solves the problem. The best being the one with the best complexity in terms of previous definition, and hence in the worst case. This is hence the complexity of the best algorithm in the worst case.

The interest of this definition is to be able to state that some algorithm is optimal: That is to say, that an algorithm is such that any other correct algorithm would be less efficient by definition.

3 Example : Computing the maximum

We will illustrate the previous discussion with an example: The problem of computing the maximum. The problem is the following: We are given a list of non-negative integers $e_1, e_2, \cdots, e_n$, with $n \geq 1$, and we want to output $M = \max_{1 \leq i \leq n} e_i$, that is the maximum of these integers.

3.1 Complexity of a first algorithm

Assuming that the input is in an array, the following Java function solves the problem:

```
static int max(int T[]) {
    int l = T.length-1;
    int M =  T[l];
    l = l-1;
    while (l ≥ 0) {
     if (M < T[l])  M = T[l];
     l = l-1;
    }
    return M;
  }
```

Assume that our elementary measure is the number of comparisons. We make 2 comparisons per iteration of the loop, which is executed $n-1$ times, plus 1 last of type $l \geq 0$ when l has the value 0. We therefore make $\mu(\mathscr{A}, n) = 2n-1$ comparisons

[1] If it exists.

[2] Assuming it exists. It may not exist.

for this algorithm $\mathscr{A}$, where n is the size of the input, that is to say the number of integers in the list $e_1, e_2, \cdots, e_n$. This number is independent of the input d, and hence $\mu(\mathscr{A}, n) = 2n - 1$.

In contrast, if our elementary measure μ is the number of assignments, we analyze the complexity as follows: we make 3 assignments before the **while** loop. Each iteration of the loop does either 1 or 2 assignments according to the result of the test M<T[l]. We hence have for an input d of size n, $n+2 \leq \mu(\mathscr{A}, d) \leq 2n+1$: The minimal value is reached for a list that has its maximum in its last element, and the maximum value for a list of n different numbers sorted in decreasing order. So here $\mu(\mathscr{A}, d)$ depends on the input d. The worst case complexity is hence $\mu(\mathscr{A}, n) = 2n$.

3.2 Complexity of a second algorithm

If the input is in an array, defined for example by:

```
class List {
  int val ;      // The element
  List next ;    // La suite

  List (int  val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

the following function solves the problem.

```
static int max(List a) {
  int M = a.val;
  for (a = a.next; a ≠ null;  a = a.next) {
    if (a.val > M)
      M = a.val;}
  return M;
}
```

Assume that our elementary measure is the number of comparisons between integers (we are not counting the comparisons between variables of type "reference" on the type List). We make one comparison per iteration of the loop, that is executed $n-1$-times, so $n-1$ comparisons in total.

The complexity $\mu(\mathscr{A}, n)$ of this algorithm $\mathscr{A}$ on the inputs of size n is hence $n-1$.

3.3 Complexity of the problem

One can wonder if it is possible to do better, and solve the problems with less than $n-1$ comparisons: The answer is no, under the condition that one restricts to algorithms that work only with comparisons[3]. Indeed, this algorithm is optimal in terms of number of comparisons.

[3] If the inputs are integers, and arithmetic is authorized, it may be possible to decrease the number of comparisons. We will not discuss this type of algorithms here.

Consider the class $\mathscr{C}$ of algorithms that solve the problem of finding the maximum of n elements by using as decision criteria the comparisons between elements, with the above hypothesis.

Let us start by stating the following property:

Lemma 1 *Any algorithm $\mathscr{A}$ of $\mathscr{C}$ is such that any element distinct from the maximum is compared at least once to an element greater then itself.*

Proof: Indeed, let i_0 be the index of the maximum M returned by the algorithm on a list $L = e_1 e_2 \cdots e_n$, that is $e_{i_0} = M = \max_{1 \leq i \leq n} e_i$. We reason by contradiction: Let $j_0 \neq i_0$ such that e_{j_0} is not compared to any element greater than itself. Then the element e_{j_0} has then not been compared to the maximum e_{i_0} .

Consider the list $L' = e_1 e_2 \cdots e_{j_0-1} M+1 e_{j_0+1} \cdots e_n$ obtained from L by replacing the element of index j_0 by $M+1$.

The algorithm $\mathscr{A}$ will do exactly the same comparisons on L and on L', without comparing $L'[j_0]$ with $L'[i_0]$ and hence will return $L'[i_0]$, so an incorrect result. We reach a contradiction which proves the property. □

It follows from this lemma that it is not possible to find the maximum of n elements with less than $n-1$ comparisons between integers. In other words, the complexity of the problem $\mathscr{P}$ of computing the maximum on the inputs of size n is $\mu(\mathscr{P}, n) = n-1$.

The previous algorithm works with $n-1$ such comparisons and is thus optimal for this measures of complexity.

3.4 Average case complexity of the algorithm

If our elementary measure μ is the number of assignments inside the **for** loop, one sees that the complexity depends on the input.

To evaluate its average case complexity, one needs to make some hypothesis on the distribution of inputs. Suppose that the lists given as inputs are permutations of $\{1, 2, \cdots, n\}$, and that the $n!$ permutations all have same probability.

One can prove [Sedgewick and Flajolet, 1996, Froidevaux et al., 1993] that the average case complexity on inputs of size n for this elementary measure μ is then H_n, the nth harmonic number: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. The number H_n is of order $\log n$ when n tends to infinity.

However, the computation is rather technical and would be laborious in the framework of this course.

Let us simplify the discussion, and let us focus on an even simpler problem: Instead of finding the maximum in the list $e_1, e_2, \ldots, e_n$, with $n \geq 1$, suppose we are given a list of integers of $\{1, 2, \ldots, k\}$ and some integer $1 \leq v \leq k$, and we want to determine if there is some index $1 \leq i \leq n$ with $e_i = v$.

The following algorithm solves the problem:

```
static boolean find(int[] T, int v) {
    for (int i = 0; i < T.length; i++)
        if (T[i] == v)
```

```
                return true;
        return false;
    }
```

Its worst case complexity in terms of elementary instructions is linear in n, since the loop is executed n times in the worst case.

Observe that the lists given as inputs are functions from $\{1,2,\ldots,n\}$ to $\{1,2,\ldots,k\}$, that we will call *array*. Suppose that each of these arrays has the same probability to be the input.

Observe that there are k^n arrays. Among those, $(k-1)^n$ do not contain the element v and in that case, the algorithm performs exactly n iterations. In the contrary case, the integer is in the array, and its first occurrence is then i with probability

$$\frac{(k-1)^{i-1}}{k^i}$$

and the algorithms stops after i iterations.

In total, we have a average case complexity of

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i$$

But for all x we have

$$\sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(to establish this result, it is sufficient to take derivative of $\sum_{i=1}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) and hence

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1 + \frac{n}{k})\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right)$$

4 Asymptotics

4.1 Asymptotic complexity

As we have seen in the previous example, a precise and complete study of the complexity of a problem can be very fastidious, and often hard. This is why the focus in computer science is often on the order of growth of the (asymptotic) complexity when the size n of the inputs becomes very big. Such an analysis is often quite representative of the performance of the algorithm, even if of course, talking about asymptotics up to some constants has its limits.

4.2 Landau notations

As it is the custom in computer science, one often reasons on the order of growth using the $\mathcal{O}(.)$ notation. We recall the following notations:

Definition 1 (Notation $\mathcal{O}(.)$) *Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$. We write $f(n) = \mathcal{O}\left(g(n)\right)$ if there exist integers c and n_0 such that for all $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Intuitively, this means that f is lower than g up to some multiplicative constant, for sufficiently big input instances.

In a similar way, one defines:

Definition 2 (Notations o, Ω, Θ) *Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$.*

- *We write $f(n) = o(g(n))$ if for all positive real numbers c there exists an integer n_0 such that for all $n \geq n_0$,*

$$f(n) \leq cg(n).$$

- *We write $f(n) = \Omega(g(n))$ if there exist integers c and n_0 such that for all $n \geq n_0$,*

$$cg(n) \leq f(n).$$

  (we have in this case $g = \mathcal{O}\left((f)\right)$)

- *We write $f(n) = \Theta(g(n))$ when $f(n) = \mathcal{O}\left(g(n)\right)$ and $f(n) = \Omega(g(n))$ hold.*

Exercise 1 *(solution on page 239) Let f and g be two functions such that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is a number $c > 0$. Prove that $f(n) = \Theta(g(n))$.

Exercise 2 *Prove:*

- *If $f = \mathcal{O}\left(g\right)$ and $g = \mathcal{O}\left(h\right)$ then $f = \mathcal{O}\left(h\right)$.*
- *If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.*
- *If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.*

Exercise 3 *(solution on page 239) Give some examples of algorithms of respective complexity (in terms of number of instructions):*

- *linear, that is* $O(n)$
- $O(n \log n)$
- *cubic, that is* $O(n^3)$
- *non-polynomial*

Exercise 4 *(solution on page 240) Suppose that one has algorithms with the complexities listed below (assuming that this corresponds to some exact times). How much are these algorithms slowed down when* (*a*) *the size of the input is doubled* (*b*) *the size of the input is increased by* 1.

1. n^2
2. n^3
3. $100n^2$
4. $n \log n$
5. 2^n

5 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest to read the first chapters of [Kleinberg and Tardos, 2005], or of the course INF421 (old version) of École Polytechnique.

Bibliography The text of this chapter is taken from a text that we wrote for the lecture notes of INF412. It is inspired by the introduction of the textbook [Kleinberg and Tardos, 2005]. The analysis of the computation of the maximum and its variations is based on the book [Froidevaux et al., 1993].

Index

Bibliography

[Froidevaux et al., 1993] Froidevaux, C., Gaudel, M., and Soria, M. (1993). *Types de donnees et algorithmes*. Ediscience International.

[Kleinberg and Tardos, 2005] Kleinberg, J. and Tardos, E. (2005). *Algorithm design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Sedgewick and Flajolet, 1996] Sedgewick, R. and Flajolet, P. (1996). *Introduction à l'analyse d'algorithmes*. International Thomson Publishing, FRANCE.