Algerian Democratic and Popular Republic Ministry of Higher Education and Scientific Research

University of May 8, 1945 – Guelma -

Faculty of Mathematics, Computer Science and Sciences matter

Department of Computer Science

Course handout

2nd year of degree

Option :Computer science

Algorithmics and Complexity

By: Dr. Chohra Chemseddine

Academic year 2023/2024

Table of Contents

Foreword

This course is aimed at second-year undergraduate students in computer science. It is intended to give them the necessary foundations to understand the concept of algorithmic complexity, a central and fundamental subject in your computer engineering curriculum. Algorithmics, as a discipline, is the art of designing algorithms, i.e. well-defined sequences of instructions to solve a problem or accomplish a given task. It is the basis of many computer applications that we use every day. From finding optimized routes on a map to recommending products online, to sorting our emails, algorithms are omnipresent and play a major role in the performance and efficiency of the software we use. One of the crucial aspects that we will cover in this course is the concept of algorithmic complexity. The complexity of an algorithm measures its performance in terms of the resources it consumes, such as execution time and memory space. Understanding and analyzing algorithmic complexity is essential to assess the quality of an algorithm and its ability to efficiently process increasingly large data. We will therefore study the different methods of complexity analysis, the notations used, and how to assess the relevance of an algorithm according to the specific constraints of each situation.

This course is divided into four (4) chapters, of which here is a brief overview:

1. Algorithmic complexity: algorithmic reminders:This chapter is dedicated to recalling the basic notions of algorithms, namely algorithms, data structures, data types, etc. These notions have already been covered in the first year (the algorithm modules and data structures 1 and 2), but we will recall them here by emphasizing the aspects that interest us in this course. We will also introduce in this chapter the notion of algorithmic complexity, by giving several examples of algorithms and analyzing their complexity.

2. Sorting algorithms:Sorting is a common operation in data processing, in this chapter we will explore different sorting techniques, from the most

simple to the most sophisticated. We will learn how to assess their respective complexity and choose the sorting best suited to each use case.

3. The trees:Trees are data structures that are widely used in computer science, particularly in databases, file systems, and search algorithms. In this chapter, we will study binary trees, binary search trees, balanced search trees, etc. We will see how to use them to solve real-world problems and how to evaluate their complexity.
4. The graphs:Finally, we will discuss graphs, which are nonlinear data structures that can be used to model many real-world situations, such as social networks, computer networks, electrical circuits, etc. We will explore graph traversal algorithms, pathfinding algorithms, and other classical problems associated with graphs. We will also study shortest path algorithms, which are widely used in navigation systems and mapping applications.

All the examples and exercises in this course are written in C language to facilitate implementation and testing, and also to be in line with the algorithm modules seen in the first year. However, the concepts and techniques that we will study are applicable to any other programming language.

We hope that this course will be an enriching and stimulating experience for you. By providing you with in-depth knowledge in this field, we aim to equip you with essential tools to solve complex problems and meet the challenges of modern computing. We encourage you to actively participate in discussions, ask questions, and work in groups for a better understanding of the concepts presented. Mastering algorithms is a fundamental skill that will serve you throughout your journey as a computer engineer and beyond.

Prerequisites:This course assumes that you have already completed the algorithms and data structures modules 1 and 2 in the first year, and that you have a good command of C programming.

Chapter 1

Algorithmic complexity

1.1 Introduction

In this chapter, we will explore the qualities and characteristics of algorithms. We highlight the criteria for judging the quality of an algorithm and the importance of writing efficient and robust algorithms. Then we discuss the crucial notion of algorithmic complexity. The complexity of an algorithm measures the amount of resources it consumes as a function of the size of the input. In other words, it evaluates the execution time and the memory required by the algorithm to process data. We will study the different forms of complexity, in particular time complexity (execution time) and space complexity (memory consumption). Understanding algorithmic complexity is essential for evaluating the performance of an algorithm, anticipating its limits and choosing the best approach to solve a given problem. We present the rules for calculating complexity in the best, worst and average cases.

To better assimilate the concepts discussed, we will illustrate each notion with concrete examples. You will have the opportunity to see how the qualities of an algorithm translate into code, how to calculate its complexity, and how to interpret the results obtained. These practical examples will help you better visualize the theoretical concepts and apply them in real situations.

1.2 Qualities and characteristics of an algorithm

Algorithms are a way of solving problems. They are used in many fields, including mathematics, computer science, economics, etc. They are omnipresent in our daily lives, when you use a GPS to find the shortest path between two points, use a search engine to find information on the Internet, or use facial recognition software to unlock your phone.

rusting your smartphone, in fact you are using an algorithm to solve a problem. Several algorithms can be used to solve the same problem. Some are more efficient than others, that is, they consume fewer resources (time and memory) to solve the problem. It is therefore important to know how to evaluate the quality of an algorithm and choose the most efficient one to solve a given problem. In this section, we will see the characteristics that allow us to judge the quality and efficiency of an algorithm. The ones presented below may be more or less important depending on the context in which the algorithm is used:

1. The precision:The algorithm must give correct and accurate results for all possible input cases, solving the specified problem according to the specifications. It must not contain logical errors and must produce expected results.
2. Simplicity and clarity:A good algorithm should be simple and easy to understand. It should be written in a clear and readable manner, with easy-to-follow steps. A good algorithm structure makes it easier to maintain and understand the code when needed.
3. Robustness:The algorithm must be able to handle unexpected cases and manage exceptional situations without failing abruptly or providing incorrect results. It must be resilient to incorrect or unexpected data.
4. Modularity:A modular algorithm is composed of several independent modules that can be reused in other algorithms. It is easier to maintain and debug a modular algorithm. In addition, it is easier to reuse it in other programs. Designing a modular algorithm allows for great scalability and ease of maintenance because it is designed to allow modifications and extensions and even adaptations to solve similar problems.
5. Efficiency :This is the most important characteristic in the context of this chapter. An efficient algorithm is one that consumes fewer resources (time and memory) to solve a problem. It must be fast and economical. It must be able to process large amounts of data in a reasonable time. The efficiency of an algorithm is measured by its algorithmic complexity.

Other features can be added to this list, such as portability, parallelism, etc. But for the purposes of this course, we are mainly interested in algorithmic complexity.

1.3 Definition of algorithmic complexity

Algorithmic complexity is a measure of the amount of resources an algorithm consumes to solve a problem. It evaluates the execution time and the memory required by the algorithm to process data. Algorithmic complexity is an important concept in computer science. It allows us to evaluate the performance of an algorithm, anticipate its limits and choose the best approach to solve a given problem. It also allows us to compare several algorithms for the same problem and choose the most efficient one. Algorithmic complexity is studied in the context of algorithm analysis. Complexity is measured according to the size of the input. The size of the input is the number of data to be processed by the algorithm. It can be expressed in number of elements, number of bits, number of characters, etc. For example, the size of the input of a sorting algorithm is the number of elements to be sorted. There are two types of algorithmic complexity: time complexity and space complexity.

1.3.1 Time complexity

The time complexity of an algorithm is the measure of the execution time of the algorithm as a function of the size of the input. It is expressed as the number of elementary operations performed by the algorithm. The elementary operations can be arithmetic operations, logical operations, input/output operations, etc. The execution time of an algorithm depends on the machine on which it is executed, the size of the input and the quality of the implementation of the algorithm. It is therefore difficult to measure the exact execution time of an algorithm. This is why we measure the execution time as a function of the size of the input. We can thus compare several algorithms for the same problem and choose the most efficient one.

1.3.2 Spatial complexity

The space complexity of an algorithm is the measure of the memory required by the algorithm to process data given the size of the input. It is expressed in number of memory units (bits, bytes, etc.). Space complexity is also an important factor to consider when designing algorithms, especially when working on embedded systems or systems with limited memory resources. An algorithm with too high a space complexity can lead to performance issues, including excessive memory consumption and memory overflow issues. It is therefore essential to find a balance between execution time efficiency and memory space usage when designing optimized algorithms.

Nowadays, time complexity is the most used to measure algorithmic complexity. That is why, in this course, we are mainly interested in time complexity. Space complexity is also important, but it is less used except in the case of embedded systems or systems with limited memory resources. In the rest of this course, we will simply write "complexity" to designate time complexity, if we do not specify otherwise.

It is also important to note that algorithmic complexity can depend either on the size of the input, or on the value of the data, or both. For example, calculating the sum ofnfirst integers is a problem whose complexity depends on the value of neven if the memory size ofnis constant. This detail will not be explained in every example in this course, but it is important to keep it in mind when analyzing the complexity of an algorithm.

1.4 Complexity calculations

Calculating the complexity of an algorithm involves analyzing how the running time varies as a function of the size of the input. To do this, one must first identify the elementary operations performed by the algorithm. Then, one must count the number of elementary operations performed by the algorithm as a function of the size of the input (usually denotedn).Finally, we must express the number of elementary operations as a function ofnand simplify the expression obtained. The result obtained is the complexity of the algorithm. The "Big O" notation is the most used to express the complexity of an algorithm. It allows to define an upper bound on the execution time of the algorithm (in the worst case). For an algorithm that does$c \cdot n$elementary operations for a size inputn,Andcis a constant, the complexity is noted$O(n)$.The constant cis usually omitted, as it has no impact on complexity. Likewise for terms of lower degrees in the case of a polynomial expression. For example, if an algorithm does$c \cdot n^2 + d \cdot n + e$elementary operations for a size inputn,we write that the complexity is$O(n^2)$. There are other notations to express complexity, such as Θ notation,Ω.These notations are less used, but they allow to define lower bounds and average cases on the execution time of the algorithm. In this course, we will focus mainly on the "Big O" notation, the other notations will be briefly discussed in the section1.5.5.

1.4.1 Elementary calculation of complexity

Basic complexity calculation is used to estimate the amount of execution time or memory space used by each operation as a function of the size of the input.

Each instruction or step in the algorithm is examined and assigned a cost in terms of time or space for that operation. Once each fundamental operation has been identified, it is determined how many times each operation is performed based on the size of the input. For example, if a loop is executed$n$times, then the cost of the operation inside this loop will be multiplied by "n". Then, by combining the costs of all the fundamental operations, we obtain the total complexity of the algorithm in terms of execution time or memory space depending on the size of the input.

Basic complexity calculus allows to perform a fine-grained analysis of the algorithm's performance and to understand which parts of the algorithm contribute the most to its overall complexity. It also allows to identify critical points that might require optimization to improve the algorithm's efficiency. It is important to note that basic complexity calculus is a theoretical approach that provides an estimate of the complexity as a function of the input size. In practice, the actual performance of the algorithm may be influenced by other factors such as hardware architecture, programming language, compiler optimizations, etc. However, the analysis of fundamental operations remains an essential tool to understand the overall behavior of the algorithm and to compare different algorithmic solutions.

1.4.2 Rules for calculating "worst case" complexity

When calculating the "worst case" complexity of an algorithm, we are interested in the maximum performance of the algorithm, that is, the execution time or memory space required when the input is the most unfavorable. For a given task$x$,to be carried out on a size input$n$,we will note$T_x(n) = O(f(n))$to say that the execution time of the algorithm is bounded by a function$f(n)$when the input is of size$n$.Here are the rules for calculating "worst case" complexity:

1. Basic operations:An elementary operation is an operation that takes a constant time to execute. For example, an arithmetic operation, a logical operation, an input/output operation, etc. The execution time of an elementary operation is independent of the size of the input. Therefore, the complexity of an elementary operation is$O(1)$.
2. Sequence of instructions:if a sequence of instructions$S$is composed of$k$ instructions$S_1, S_2, \ldots, S_k$, then the complexity of the sequence$S$is the sum of the instruction complexities$S_1, S_2, \ldots, S_k$. In other words, we can write:

$$T_S(n) = \sum_{i=1}^{k} T_{S_i}(n) \quad (1.1)$$

3.Choice :if an instructionSis composed of several instructionsS_1,S_2, ...,S_k
which are executed based on a condition (ifOrswitch),then the complexity of the instructionSdepends on the instruction$S_i$having the highest complexity. We also write:

$$T_S(n) = \max_{1 \le i \le k} T_{S_i}(n) \quad (1.2)$$

4.Loop :if an instructionSis a loop that executesntimes at most, and that the body of the loop has a complexity$T_B(n)$,then the complexity of the instruction Sis given by the formula:

$$T_S(n) = n \cdot T_B(n) \quad (1.3)$$

This reasoning is valid for loopsforas well as for the loopswhile. In the case of a loopwhile,care must be taken to ensure that the loop exit condition is met. Otherwise, the loop will run indefinitely and the algorithm will never terminate. In this case, the complexity of the loop is O(∞).

5.Nested loops:if an instructionSis composed ofknested loops, and that the body of the innermost loop has a complexity$T_B(n)$,then the complexity of the instructionSis given by the formula:

$$T_S(n) = \prod_{i=1}^{k} n_k \cdot T_B(\eta) \quad (1.4)$$

Or$n_k$is the number of iterations of the loopk.

6.Function call:if an instructionSis a function callfwhich has a complexity$T_f(n)$,then the complexity of the instructionSis the same as that of the functionf.

7.Recursion:In the case of recursive functions, it is necessary to express the complexity of the recursion as a function of the size of the input and to calculate the total number of recursive calls in the worst case. To do this, it is necessary to define a recurrence relation that expresses the complexity as a function of the size of the input. Then, it is necessary to solve the recurrence relation to obtain an explicit expression of the complexity as a function of the size of the input. Finally, it is necessary to simplify the expression obtained and determine the complexity of the recursion.

1.5 Examples of complexity calculations

In this section, we will illustrate the rules of complexity calculation with concrete examples. We will calculate the complexity of several simple algorithms. We will also see how the qualities of an algorithm translate into code, how to calculate its complexity, and how to interpret the results obtained. These practical examples will help you better visualize the theoretical concepts and apply them in real-world situations.

1.5.1 Linear complexity

Linear complexity is a type complexityO(n).It is used to express the complexity of an algorithm whose running time is proportional to the size of the input. For example, if an algorithm doesc·nelementary operations for a size inputn,then the complexity of this algorithm isO(n).Linear search is an example of an algorithm with linear complexity. The linear search algorithm is used to search for an element in an array. It traverses the array element by element and compares each element with the searched value. If the element is found, the algorithm stops and returns the position of the element. Otherwise, it returns a special value (for example, -1) to indicate that the element was not found. Here is an example of implementing a linear search function in C language:

```
1  intlinear_search(int*array ,intsize ,intvalue) {
2      for(inti = 0; i < size; i++) {
3          if(array[i] == value) {
4              returni;
5          }
6      }
7      return-1;
8  }
```

Algorithm 1.1 – Linear search

The elementary operation that interests us in this algorithm is the comparison between two integers. This operation is performed in the instructionifon line 3. It is executed at each iteration of the loopfor.The loopforis executedntimes for a size entryn.The loop may possibly stop before traversing the entire array if the element is found, but in the worst case, if the element is not found, the loop will be executed ntimes. So the complexity of the functionlinear_searchEastO(n), because she does her bestnelementary operations for a size inputn.

1.5.2 Constant complexity

Constant complexity is a type complexity$O(1)$.It is used to express the complexity of an algorithm whose running time is independent of the size of the input. For example, if an algorithm does$c$elementary operations for a size input$n$,with$c$a constant, then the complexity of this algorithm is$O(1)$.Finding an element in a sorted array is an example of an algorithm with constant complexity. We will take as an example the calculation of the sum of the$n$first integers with$n$a positive integer. At first glance, one might think that an algorithm that solves this problem must iterate through all integers from 1 to$n$and add them as shown below:

```
int sum_n(int n) {
    int sum = 0;
    for(int i = 1; i <= n; i++) {
        sum + = i;
    }
    return sum;
}
```

Algorithm 1.2 – Sum of the first n integers - Linear complexity

This algorithm does$n$elementary operations for a size input$n$.So its complexity is $O(n)$.However, there is a more efficient solution to solve the same problem. Indeed, the sum of the$n$first integers can be calculated using the formula for the sum of$n$ first elements of an arithmetic sequence having as first termha_1=1and for good reason$r = 1$.The sum of the$n$first integers is given by the following formula:

$$S_n = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} \qquad (1.5)$$

This formula is used to calculate the sum of the$n$first integers by performing three elementary operations. Here is an implementation of the functionsum_nwho uses the formula1.5:

```
int sum_n(int n) {
    return(n * (n + 1)) / 2;
}
```

Algorithm 1.3 – Sum of the first n integers - Constant complexity

This function always performs three elementary operations regardless of the value of$n$. So its complexity is$O(1)$.

It is important to note that constant complexity does not necessarily mean that an algorithm is always faster for all input sizes. It simply means that the running time or memory consumption does not depend on the input size. Constant complexity is generally desirable for operations that must be very fast and not depend on the size of the data. However, in some cases, algorithms with higher complexity may be more efficient for large input sizes. The choice of algorithm depends on the context and the specific needs of each problem to be solved.

1.5.3 Logarithmic complexity

Logarithmic complexity is a type complexity $O(\log n)$. It is used to express the complexity of an algorithm whose running time is proportional to the logarithm of the input size. For example, if an algorithm does $c \cdot \log n$ elementary operations for a size input n, then the complexity of this algorithm is $O(\log n)$. An example of an algorithm with logarithmic complexity is calculating the power of a number. As with the previous example, one might think that an algorithm that calculates x^n with n a positive integer must perform n multiplications as shown below:

```
int pow_n(int x, int n) {
    int result = 1;
    for(int i = 0; i < n; i++) {
        result *= x;
    }
    return result;
}
```

Algorithm 1.4 – Power of a number - Linear complexity

However, there is a more efficient solution to solve the same problem. In fact, the power of a number x to the power n can be calculated using the following recurrence formula:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot (x^{(n-1)/2})^2 & \text{if } n \text{ is odd} \\ (x^{n/2})^2 & \text{if } n \text{ is even} \end{cases} \tag{1.6}$$

This formula is used to calculate the power of a number. x to the power n by dividing the problem into smaller subproblems. The complexity of this algorithm is logarithmique, because at each step the problem is divided by two. Here is an implementation of the functionpow_nwho uses the formula1.6:

```
 1  intpow_n(intx,intn) {
 2      if(n == 0) {
 3          return1;
 4      }else if(n % 2 == 0) {
 5          inty = pow_n(x, n / 2); returny *
 6          y;
 7      }   else{
 8          inty = pow_n(x, (n - 1) / 2); returnx * y *
 9          y;
10      }
11  }
```

Algorithm 1.5 – Power of a number - Logarithmic complexity

The elementary operation of the functionpow_nis multiplication. This operation is performed on line 6 and line 9. The functionpow_nis recursive. It calls itself with a smaller parameter. The complexity is given by the following recursion relation:

$$T(n) = \begin{cases} 0 & \text{if} n = 0 \\ T(n/2) + 2 & \text{if} n \text{ is odd} \\ T(n/2) + 1 & \text{if} n \text{ is even} \end{cases} \tag{1.7}$$

It can be easily demonstrated by recurrence that$T(n) = O(\log_n n)$.But another way to look at it is to notice that the parameternis divided by two at each recursive call, in other words, the parameter loses a bit at each call. So the number of recursive calls is proportional to the number of bits in the value ofn,which implies that the complexity of the functionpow_nEast$O(\log_2 n)$,because she does her best$2 \cdot \log_2 n$ elementary operations in the case where all calls are made with odd values.

1.5.4 Quadratic complexity

Quadratic complexity is a type complexity$O(n^2)$. It is used to express the complexity of an algorithm whose running time is proportional to the square of the input size. For example, if an algorithm does$c \cdot n^2$elementary operations for a size inputn,then the complexity of this algorithm is$O(n^2)$. An example of an algorithm with quadratic complexity is searching for a pair of elements in an array whose sum is equal to a given value. This algorithm iterates through the array and compares each element with all other elements in the array. If the sum of two elements is equal to the given value, the algorithm stops and returns 1 (true). Otherwise, it returns 0 (false) to indicate that the pair was not found. Here is an example of implementing a function to find a pair of elements in an array whose sum is equal to a given value in C language:

```
1   intpair_sum(int*array ,intsize ,intvalue) {
2       for(inti = 0; i < size; i++) {
3           for(intj = 0; j < size; j++) {
4               if(array[i] + array[j] == value) {
5                   return1;
6               }
7           }
8       }
9       return0;
10  }
```

Algorithm 1.6 – Finding a pair of elements whose sum is equal to a given value

The elementary operation that interests us in this algorithm is the comparison between two integers. This operation is performed in the instructionifat line 4. It is executed at each iteration of the loopforinternal. The loopforinternal is executed$n$ times for a size entry$n$.The loopforexternal is also executed$n$times for a size entry$n$. So the complexity of the functionpair_sumEast$O(n^2)$, because it does the maximumn^2elementary operations for a size input$n$.

It is possible to improve the complexity of this algorithm by first sorting the array with an efficient sorting algorithm (quicksort, mergesort,etc.). Then, we can use a two-way search to find the pair of elements (the technique twopointers).This algorithm will be presented in the chapter2as one of the applications of sorting algorithms.

Quadratic complexity is a special case of polynomial complexity. An algorithm whose complexity is polynomial is an algorithm whose complexity is of the form$O(n^k)$with$k$a positive integer. Polynomial complexity is used to express the complexity of an algorithm whose running time is proportional to a power of the input size. For example, if an algorithm does$c \cdot n^k$elementary operations for a size input$n$,then the complexity of this algorithm is$O(n^k)$.

1.5.5 Complexity at best, worst and average

Best-case, worst-case, and average complexity are notations used to express the complexity of an algorithm in particular cases. Best-case complexity is the complexity of the algorithm in the best case. That is, when the input is most favorable. It is denoted Ω.Worst-case complexity is the complexity of the algorithm in the worst case. That is, when the input is the most unfavorable. It is denoted O.The average complexity is the complexity of the algorithm in the average case. That is, when the input is random. In other words, it is the complexity observed on a large number of random inputs. The best-case complexity is always less than or equal to the average complexity, which is itself less than or equal to the worst-case complexity. Consider as an example the linear search algorithm implemented in the algorithm1.1.

— In the best case, the element sought is at the first position of the array. In this case, the algorithm stops after a single iteration and returns the position of the element. So the best-case complexity of this algorithm is $\Omega(1)$.

— The worst-case complexity of this algorithm is $O(n)$ as we have already explained in the section1.5.1.

— The average complexity is a bit more complicated to calculate. It depends on the distribution of the data in the array as well as the probability of finding the desired element. If we assume that the data is uniformly distributed in the array, and that the desired element certainly belongs to the array, then we have $T(n) = \frac{n}{2}$(the average of 1 has n).If we assume that the sought element belongs to the array with a probability p,then the average complexity is given by the formula

$$T(n) = p\cdot\frac{n}{2} + (1-p)\cdot n \tag{1.8}$$

In case $p = 1$,We have $T(n) = \frac{n}{2}$, and in the event that $p = 0$,We have $T(n) = n$.So the complexity on average is between $\Omega(1)$ And $O(n)$.

1.5.6 Exponential complexity

Exponential complexity is a complexity of type $O(c^n)$. It is used to express the complexity of an algorithm whose running time is proportional to an exponential of the input size. For example, if an algorithm does c^n elementary operations for a size input n,then the complexity of this algorithm is $O(c^n)$. An example of an algorithm with exponential complexity is the exhaustive search for a solution to a problem (encryption key, password, etc.). This algorithm generates all possible solutions and checks if the solution is correct. If the solution is correct, the algorithm stops and returns the solution. Otherwise, it continues generating solutions. In the example of finding a password, if we assume that the password consists of$n$characters, and that each character can take$c$different values, then the total number of possible solutions isc^n. So the complexity of this algorithm is$O(c^n)$.

Algorithms with exponential (or factorial) complexity are very slow and not efficient. They are very rarely used in real applications to solve small or medium-sized problems. However, they can be used to solve large problems if no other more efficient solution is known to solve the problem. Problems that have no known efficient solution are called NP-complete problems, but their study is beyond the scope of this course.

1.6 Conclusion

In this chapter, we introduced the basic concepts of algorithmic complexity. We saw how to measure the complexity of an algorithm based on the size of the input. We also saw how to calculate the complexity of an algorithm using the rules of complexity calculation. We illustrated these rules with concrete examples. In the next chapter, we will focus on sorting algorithms to demonstrate that for a given problem, there are several possible algorithms, and that some algorithms are more efficient than others to solve the same problem.