1 Introduction

This is rocket science but you don’t have
to be a rocket scientist to use it.
Jack Noonan, CEO of SPSS

The main actor in this book is the *algorithm*, so in order to dig into the beauty and challenges that pertain to its ideation and design, we need to start from one of its many possible definitions. The Oxford English Dictionary reports that an algorithm is, informally, *“a process, or set of rules, usually one expressed in algebraic notation, now used esp. in computing, machine translation and linguistics.”* The modern meaning of *algorithm* is quite similar to that of *recipe*, *method*, *procedure*, or *routine*, but in computer science the word connotes something more precisely described. In fact many authoritative researchers have tried to pin down the term over the past 200 years by proposing definitions that have become more complicated and detailed but, in the minds of their proponents, more precise and elegant.[1] As algorithm designers and engineers we will follow the definition provided by Donald Knuth at the end of the 1960s [7]: an algorithm is *a finite, definite, effective procedure, with some output.* Although these features may be intuitively clear and are widely accepted as requirements for a sequence of steps to be an algorithm, they are so dense in significance that we need to look at them in more detail; this will lead us to the scenarios and challenges posed nowadays by algorithm design and engineering, and to the motivation behind this book.

- Finite: “An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number.” Clearly, the term “reasonable” is related to the *efficiency* of the algorithm: Knuth [7] states that “In practice, we not only want algorithms, we want good algorithms.” The “goodness” of an algorithm is related to the use that the algorithm makes of some precious *computational resources*, such as: time, space, communication, I/Os, energy, or just simplicity and elegance, which both impact on its coding, debugging and maintenance costs.
- Definite: “Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.” Knuth made an effort in this direction by detailing what he called the “machine language”

[1] See “algorithm characterizations” at https://en.wikipedia.org/wiki/Algorithm_characterizations.

for his “mythical MIX... the world’s first polyunsaturated computer.” Today there are many other programming languages, such as C/C++, Java, Python, and so on. They all specify a set of instructions that programmers may use to describe the procedure[s] underlying their algorithm[s] in an unambiguous way: “unambiguity” here is granted by the formal semantics that researchers have attached to each of these instructions. This eventually means that anyone reading that algorithm’s description will interpret it in a precise way: nothing will be left to personal choice.

- Effective: “all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil.” Therefore the notion of “step” invoked in the previous item implies that one has to dig into a complete and deep understanding of the problem to be solved, and then into logical well-defined structuring of a step-by-step solution.
- Procedure: “the sequence of specific steps arranged in a logical order.”
- Input: “quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects.” Therefore the behavior of the algorithm is not unique, but it depends on the “sets of objects” given as input to be processed.
- Output: “quantities which have a specified relation to the inputs” given by the problem at hand, and constitute the answer returned by the algorithm for those inputs.

In this book we will not use a formal approach to algorithm description, because we wish to concentrate on the theoretically elegant and practically efficient ideas that underlie the algorithmic solution of some interesting problems, without being lost in the maze of programming technicalities. So, in every chapter, we will take an interesting problem that emerges from a practical/useful application and then propose solutions of increasing sophistication and improved efficiency, taking care that this will not necessarily lead to increasing the complexity of the algorithm’s description. Actually, problems were selected to admit surprisingly elegant solutions that can be described in a few lines of code. So we will opt for the current practice of algorithm design and describe our algorithms either colloquially or by using *pseudocode* that mimics, the most well known languages. In all cases the algorithm descriptions will be as rigorous as they need to be to match Knuth’s six features.

Elegance will not be the only goal of our algorithm design, of course; we will also aim for *efficiency*, which commonly relates to the *time/space complexity* of the algorithm. Traditionally, time complexity has been evaluated as a function of the input size n by counting the (maximum) number of steps, say $T(n)$, an algorithm takes to complete its computation over an input of n items. Since the maximum is taken over all inputs of that size, the time complexity is termed *worst case* because it concerns the input that induces the worst behavior in time for the algorithm. Of course, the larger n is, the larger $T(n)$ is, which is therefore nondecreasing and positive. In a similar way we can define the (worst-case) space complexity of an algorithm as the maximum number of memory cells it uses for its computation over an input of size n.

This approach to the *design* and *analysis* of algorithms assumes a very simple model of computation, known as the *Von Neumann model* (aka random access machine, or *RAM* model). This model consists of a CPU and a memory of infinite size, and constant-time access to each of its cells. Here we argue that every step takes a fixed amount of time on a PC, which is the same for any operation, be it arithmetic, logical, or just a memory access (read/write). Hence we postulate that it is enough to *count* the number of steps executed by the algorithm in order to have an "accurate" estimate of its execution time on a real PC. Two algorithms can then be compared according to the *asymptotic behavior* of their time-complexity functions as $n \to +\infty$; the faster the time complexity grows over inputs of increasing size, the worse the corresponding algorithm is judged to be. The robustness of this approach has been debated for a long time but, eventually, the RAM model dominated the algorithmic scene for decades (and is still dominating it) because of its simplicity, which impacts on algorithm design and evaluation, and its ability to estimate the algorithm performance "quite accurately" on (old) PCs and small input sizes. Therefore it is not surprising that most introductory books on algorithms deploy the RAM model to evaluate their performance [6].

But in the past ten years things have changed significantly, thus highlighting the need for a shift in algorithm design and analysis. Two main changes occurred: the architecture of modern PCs became more and more sophisticated (not just one CPU and one monolithic memory), and input data has exploded in size ("$n \to +\infty$" does not only belong in the theoretical world), because it is abundantly generated by many sources, such as DNA sequencing, bank transactions, mobile communications, web navigation and searches, auctions, and so on. The first change turned the RAM model into an unsatisfactory abstraction of modern PCs, whereas the second change made the design of asymptotically good algorithms ubiquitous and fruitful not only for theoreticians but also for a much larger professional audience because of their impact on business [2], society [1], and science in general [3]. The net consequence was a revamped scientific interest in algorithmics and the spread of the word "algorithm" to even colloquial speech.

In order to make algorithms effective in this new scenario, researchers needed new models of computation able to abstract in a better way the features of modern computers and applications and, in turn, to derive more accurate estimates of algorithm performance from the analysis of their complexity. Nowadays a modern PC consists of one or more CPUs (multi-cores, GPUs, TPUs, etc.) and a very complex hierarchy of memory levels, all with their own technological peculiarities (see Figure 1.1): L1 and L2 caches, internal memory, one or more mechanical or solid-state disks, and possibly other (hierarchical) memories of multiple hosts distributed over a (possibly geographic) network, the so-called "cloud." Each of these memory levels has its own cost, capacity, latency, bandwidth, and access method. The closer a memory level is to the CPU, the smaller, the faster, and the more expensive it is. Currently, nanoseconds suffice to access the caches, whereas milliseconds are needed to fetch data from disks (aka I/O). This is the so-called *I/O bottleneck*, which amounts to the astonishing factor of $10^5 - 10^6$, nicely illustrated in a quote attributed to Thomas H. Cormen:

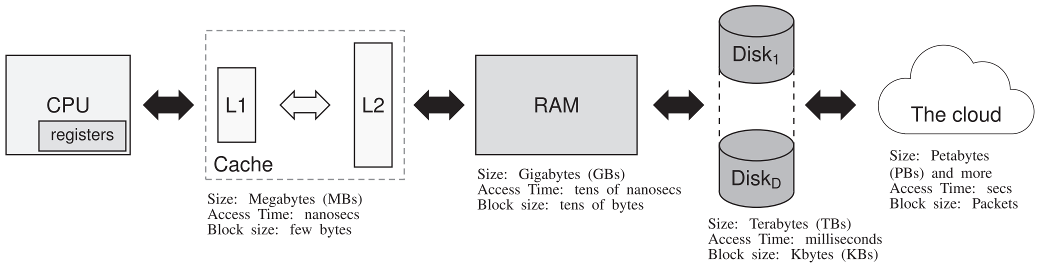

Figure 1.1 An example of memory hierarchy in a modern PC.

"The difference in speed between modern CPU and (mechanical) disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk."

Engineering research is trying nowadays to improve input/output subsystems to reduce the impact of the I/O bottleneck on the efficiency of applications managing large datasets; on the other hand, however, the improvements achievable by means of good algorithm design and engineering abundantly surpass the best expected technology advancements. Let us see the why, with a simple example.[2]

Consider three algorithms which have increasing I/O complexity (and thus, time complexity): $C_1(n) = n$, $C_2(n) = n^2$, and $C_3(n) = 2^n$. Here $C_i(n)$ denotes the number of disk accesses executed by the i-th algorithm to process n input data. Notice that the first two algorithms execute a *polynomial* number of I/Os (in the input size n), whereas the last one executes an *exponential* number of I/Os in n. Note that these I/O complexities have a very simple (and thus unrealistic) mathematical form, because we want to simplify the calculations without impairing our final conclusions. Let us now ask how much data each of these algorithms is able to process in a fixed time interval of length t, given that each I/O takes c time. The answer is obtained by solving the equation $C_i(n) \times c = t$ with respect to n: so we get t/c data are processed by the first algorithm in time t, $\sqrt{t/c}$ data are processed by the second algorithm, and only $\log_2(t/c)$ data are processed by the third algorithm in time t. These values are already impressive by themselves, and provide a robust understanding of why polynomial-time algorithms are called *efficient*, whereas exponential-time algorithms are called *inefficient*: a large change in the length t of the time interval induces just a tiny change in the amount of data that exponential-time algorithms can process. Of course, this distinction admits many exceptions when the problem instances have limited input size or have distributions that favor efficient executions. But, on the other hand, these examples are quite rare, and the much more stringent bounds on execution time satisfied by polynomial-time algorithms mean that they are considered *provably* efficient and the preferred way to solve problems. Algorithmically speaking, most exponential-time algorithms are merely implementations of the approach based on exhaustive searches, whereas

[2] This is paraphrased from [8]; here we talk about I/Os instead of steps.

polynomial-time algorithms are generally made possible only through gaining some deeper insight into the structure of a problem. So polynomial-time algorithms are the right choice from many points of view.

Let us now assume that we run these algorithms with a better I/O-subsystem, say one that is k times faster, and ask: How much data can be managed by this new computer? To address this question we solve the previous equations with the time interval set to the length $k \times t$, thus implicitly assuming that the algorithms are executed with k times more available running time than the previous computer. We find that the first algorithm perfectly scales by a factor of k, the second algorithm scales by a factor of $\sqrt{k}$, whereas the last algorithm scales *only by an additive term* $\log_2 k$. We can see that the improvement induced by a k-times more powerful computer for an exponential-time algorithm is totally negligible even in the presence of impressive (and thus unrealistic) technology advancements. Super-linear time algorithms, like the second one, are positively affected by technology advancements, but their performance improvement decreases as the degree of the polynomial-time complexity grows: more precisely, if $C(n) = n^\alpha$ then a k-times more powerful computer induces an increase in speed by a factor of $\sqrt[\alpha]{k}$. Overall, it is safe to say that the impact of a good algorithm is far beyond any optimistic forecasting for the performance of future (mechanical or solid-state) disks.[3]

Given this appetizer on the "power" of algorithm design and engineering, let us now turn back to the problem of analyzing the performance of algorithms in modern computers by considering the following simple example: compute the sum of the integers stored in an array $A[1, n]$. The simplest idea is to scan A and accumulate in a temporary variable the sum of the scanned integers. This algorithm executes n sums between two integers, accesses each integer in A once, and thus takes n steps. Let us now generalize this approach by considering a family of algorithms, denoted by $\mathcal{A}_{s,b}$, which differentiate themselves according to the pattern of accesses to A's elements, as driven by the parameters s and b. In particular, $\mathcal{A}_{s,b}$ looks at array A as logically divided into blocks of b elements each, say $A_j = A[j \times b + 1, (j+1) \times b]$ for $j = 0, 1, 2, \ldots, n/b - 1$.[4] Then it sums all items in one block A_j before moving to the next block A_{j+s}, which occurs s blocks farther on the right. Array A is considered cyclic so that, when the next block lies out of A, the algorithm wraps around it, starting again from its beginning: hence, the index of the next block is actually defined as $(j + s) \bmod (n/b)$.[5] Clearly, not all values of s allow us to take into account all of A's blocks (and thus sum all of A's integers). And in fact we know that if s is coprime with n/b then the sequence of visited-block indexes, that is, $j = s \times i \bmod (n/b)$ for $i = 0, 1, \ldots, n/b - 1$, is a permutation of the integers $\{0, 1, \ldots, n/b - 1\}$, and thus $\mathcal{A}_{s,b}$ touches all blocks in A and hence sums all of its integers. But the peculiarity of this parametrization is that by varying s and b we can sum A's integers according to different patterns of memory accesses:

[3] See [11] for an extended treatment of this subject.

[4] For the sake of presentation we assume that n and b are powers of two, so b divides n.

[5] The modulo (mod) function is defined as follows: given two positive integers x and $m > 1$, $x \bmod m$ is the remainder of the division of x by m.

from the sequential scan we have described (setting $s = b = 1$), to sequential-wise blocked access (setting a larger b), or to random-wise blocked access (setting a larger s). Nicely enough, all algorithms $\mathcal{A}_{s,b}$ are equivalent from a computational point of view, because they read and sum exactly n integers and thus take exactly n steps; but from a practical point of view, they have different time performance which becomes more and more different as the array size n grows. The reason for this is that, for a growing n, data will be spread over more and more memory levels, each with its own capacity, latency, bandwidth and access method. So the "equivalence in efficiency" derived by adopting the RAM model, and counting the number of steps executed by $\mathcal{A}_{s,b}$, is not an accurate estimate of the real time required by the algorithms to sum A's elements.

We need a different model that grasps the essence of real computers and is simple enough to not jeopardize the algorithm design and analysis. In a previous example we argued that the number of I/Os is a good estimator for the time complexity of an algorithm, given the large gap between disk- and internal-memory performance. This is indeed captured by the *2-level memory model* (aka disk model, or external-memory model [11]), which abstracts the computer as comprising only *two memory levels*: the internal memory of (bounded) size M, and the (unbounded) disk memory which operates by reading/writing data via blocks of size B (called *disk pages*). Sometimes the model consists of D disks, each of unbounded size, so that each I/O reads or writes a total of $D \times B$ items stored in D pages, each one residing on a different disk. For the sake of clarity we remark that the two-level view must not suggest to the reader that this model is restricted to abstract disk-based computations; in fact, we are actually free to choose any two levels of the memory hierarchy, with their M and B parameters properly set. The algorithm performance is evaluated in this model by counting: (i) the number of accesses to disk pages (hereafter *I/Os*), (ii) the running time (CPU time), and (iii) the number of disk pages used by the algorithm as its working space. This also suggests two golden rules for the design of "good" algorithms operating on large datasets: they must exploit *spatial locality* and *temporal locality*. The former imposes a data organization in the disk(s) that makes each accessed disk page as useful as possible; the latter requires as much useful work as possible over the data fetched in internal memory, before it is written back to disk.

In the light of this new model, let us reanalyze the time complexity of algorithms $\mathcal{A}_{s,b}$ by taking into account I/Os, given that the CPU time is n and the space occupancy is n/B disk pages independently of s and b. We start from the simplest settings for s and b in order to gain some intuitions about the general formulas. The case $s = 1$ is obvious: algorithms $\mathcal{A}_{1,b}$ scan A rightward, summing the items one block at a time, by taking n/B I/Os independently of the value of b. As s and b change, the situation gets complicated, but by not much. As an example, fix $s = 2$ and select some $b < B$ that, for simplicity, is assumed to divide the block-size B. Every block of size B consists of B/b smaller (logical) blocks of size b, and the algorithms $\mathcal{A}_{2,b}$ examine only half of them because of the jump $s = 2$. This actually means that each B-sized page is half utilized in the summing process, thus inducing a total of $2n/B$ I/Os. It is then not difficult to generalize this formula by writing a cost of $\min\{s, B/b\} \times (n/B)$ I/Os, which correctly gives n/b for the case of large jumps over array A. This formula provides a better approximation of the real time complexity of the algorithms $\mathcal{A}_{s,b}$, although it does not capture all features of the disk: all I/Os are evaluated as equal, independently of their distribution. This is clearly not precise, because on real disks *sequential* I/Os are faster than *random* I/Os.[6] As such, referring to the previous example, all algorithms $\mathcal{A}_{s,B}$ have the same I/O complexity n/B, independently of s, although their behavior is rather different if executed on a (mechanical) disk, because of the disk seeks induced by increasing s. Therefore, we can conclude that even the two-level memory model provides an approximation of the behavior of algorithms on real computers, although its results are sufficiently good that it has been widely adopted in the literature to evaluate algorithm performance on massive datasets. So in order to be as precise as possible, we will evaluate algorithms in these pages not only by specifying the number of executed I/Os but also by characterizing their *distribution* (random vs. sequential) over the disk.

At this point one could object that given the impressive technological advancements of recent years, internal-memory size M is so large that most of the working set of an algorithm (roughly speaking, the set of pages it will reference in the near future) can fit into it, thus reducing significantly the number of I/O faults. We will argue that even a small portion of data resident in disk makes the algorithm slower than expected, so that data organization cannot be neglected even in these extremely favorable situations. Let us see why, by means of a "back of the envelope" calculation.

Assume that the input size $n = (1 + \epsilon)M$ is larger than the internal-memory size of a factor $\epsilon > 0$. The question is how much ϵ impacts on the average cost of an algorithm step, given that it may access a datum located either in internal memory or on disk. To simplify our analysis, while still obtaining a meaningful conclusion, we assume that $p(\epsilon)$ is the probability of an I/O fault: hence, if $p(\epsilon) = 1$, the algorithm always accesses data on disk; if $p(\epsilon) = 0$, the algorithm has a working set smaller than the internal-memory size, and thus it always accesses data in internal memory; finally, $p(\epsilon) = \frac{\epsilon M}{(1+\epsilon)M} = \frac{\epsilon}{1+\epsilon}$ when the algorithm has a fully random behavior in accessing its input data. In other words, we can look at $p(\epsilon)$ as a measure of the non-locality of the memory references of the analyzed algorithm.

To complete the notation, let us indicate with c the time cost of one I/O with respect to one internal-memory access (we have in practice $c \approx 10^5 - 10^6$, see above), with f the fraction of steps that induce a memory access in the running algorithm (this is typically 30%–40%, according to [5]), with t_m the average time cost of such memory accesses and the cost of a computation step or an internal-memory access set as 1. To derive t_m we have to distinguish two cases: an in-memory access (occurring with probability $1 - p(\epsilon)$) or a disk access (occurring with probability $p(\epsilon)$). So we have $t_m = 1 \times (1 - p(\epsilon)) + c \times p(\epsilon)$.

[6] Conversely, this difference will be almost negligible in an (electronic) memory, such as DRAM or modern solid-state disks, where the distribution of the memory accesses does not significantly impact on the throughput of the memory/SSD.

Now we are ready to estimate the *average time cost of a step* for an algorithm working in this scenario: it is $1 \times (1 - f) + t_m \times f$, since $1 - f$ is the fraction of computing steps and f is the fraction of memory accesses (both in internal memory and on disk). By plugging in the value computed for t_m, we can lower bound that cost by $3 \times 10^4 \times p(\epsilon)$. This formula clearly shows that, even for algorithms exploiting locality of references (i.e. a small $p(\epsilon)$), the slowdown may be significant, resulting in four orders of magnitude larger than what might be expected (i.e. $p(\epsilon)$). As an example, take an algorithm that forces locality of references into its memory accesses: say 1 out of 1000 memory accesses go to data stored on disk (i.e. $p(\epsilon) = 0.001$). Then, its performance gets slowed down by a factor larger than 30 in comparison with the case in which its computation would be fully executed in internal memory.

It goes without saying that this is just the tip of the iceberg, because the larger the amount of data to be processed by an algorithm, the higher is the number of memory levels involved in the storage of this data and, hence, the more varied are the types of "memory faults" that need to be coped with for achieving efficiency. The overall message is that neglecting questions pertaining to the cost of memory references in a hierarchical-memory system may prevent the use of an algorithm for large input data.

Motivated by these premises, this book will provide a few examples of challenging problems that admit elegant algorithmic solutions whose efficiency is crucial to manage the large datasets that occur in many real-world applications. Details of the algorithm design will be accompanied by several comments on the difficulties that underlie the engineering of those algorithms: how to turn a "theoretically efficient" algorithm into a "practically efficient" code. In fact, too many times, as a theoretician, I was told that "your algorithm is far from being amenable to an efficient implementation!" Furthermore, by following the recent surge of investigations in *algorithm engineering* [10] (not to be confused with the "practice of algorithms"), we will also dig into the deep computational features of some algorithms by resorting to a few other successful models of computation – mainly the streaming model [9] and the cache-oblivious model [4]. These models will allow us to capture and highlight some interesting issues of the underlying computation, such as disk passes (streaming model), and universal scalability (cache-oblivious model). We will try our best to describe all these issues in their simplest terms but, nonetheless, we will be unsuccessful in turning this "rocket science for non-boffins" into a "science for dummies" [2]. In fact many more things have to fall into place for algorithms to work: top IT companies (like Amazon, Facebook, Google, IBM, Microsoft, Oracle, Spotify, Twitter, etc.) are perfectly aware of the difficulty of finding people with the right skills for designing and engineering "good" algorithms. This book will only scratch the surface of algorithm design and engineering, with the main goal of inspiring you in your daily job as a software designer and engineer.

References

[1] Person of the Year. *Time Magazine*, 168:27, December 2006.
[2] Business by numbers. *The Economist*, September 2007.

[3] Declan Butler. *2020 computing: Everything, everywhere*. *Nature*, 440(7083): 402–5, 2006.

[4] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 264–9, 2016.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2006.

[6] Ming-Yang Kao. *Encyclopedia of Algorithms*. Springer, 2016.

[7] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Vol. 1. Addison-Wesley, 1973.

[8] Fabrizio Luccio. *La struttura degli algoritmi*. Boringhieri, 1982.

[9] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2): 117–236, 2005.

[10] Peter Sanders. Algorithm engineering – an attempt at a definition. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science, 5760, Springer, 321–3, 2009.

[11] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2): 209–71, 2001.