1

Prologo

"This is a rocket science but you
don't need to be a rocket
scientist to use it"
The Economist, September 2007

The main actor of this book is *the Algorithm* so, in order to dig into the beauty and challenges that pertain with 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 for Algorithm is quite similar to that of *recipe, method, procedure, routine* except that the word Algorithm in Computer Science connotes something more precisely described. In fact many authoritative researchers have tried to pin down the term over the last 200 years [3] by proposing definitions which became more complicated and detailed nonetheless, hopefully in the minds of their proponents, more precise and elegant. As algorithm designers and engineers we will follow the definition provided by Donald Knuth at the end of the 60s [7, pag 4]: an Algorithm is *a finite, definite, effective procedure, with some output.* Although these five features may be intuitively clear and are widely accepted as requirements for a sequence-of-steps to be an Algorithm, they are so dense of significance that we need to look into them with some more detail, even because this investigation will surprisingly lead us to the scenario and challenges posed nowadays by algorithm design and engineering, and to the motivation underling these lectures.

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, pag. 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 onto the 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" for his *"mythical MIX...the world's first polyunsaturated computer"*. Today we know of many other programming languages such as `C/C++`, `Java`, `Python`, etc. etc. All of them specify a set of instructions that the programmer may use to describe the procedure underlying his/her algorithm 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 the algorithm's description will interpret it in a precise way: nothing will be left to personal mood!

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-definite 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"*.

Output: *"... quantities which have a specified relation to the inputs"*.

In this booklet we will not use a *formal* approach to algorithm description, because we wish to concentrate on the theoretically elegant and practically efficient ideas which underlie the algorithmic solution of some interesting problems, without being lost in the maze of programming technicalities. So in every lecture we will take an interesting *problem* coming out from a *real/useful application* and then propose *deeper and deeper* solutions of increasing sophistication and improved efficiency, taking care that this will not necessarily correspond to increasing the complexity of algorithm's description. Actually, problems were selected to admit *surprisingly* elegant solutions that can be described in few lines of code! So we will opt for the current practice of algorithm design and describe our algorithms either *colloquially* or by using *pseudo-code* that mimics the most famous `C` and `Java` languages. In any case we will not renounce to be as much rigorous as it needs an algorithm description to match the five features above.

Elegance will not be the only feature 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 named *worst case* because it concerns with the input that induces the worst behavior in time for the algorithm. Of course, the larger is n the larger is $T(n)$, which is therefore non decreasing 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 *model of Von Neumann* (aka Random Access Machine, *RAM* model). This model consists of a CPU and a memory of infinite size and constant-time access to each one 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: being it arithmetic, logical, or just a memory access (read/write). Here one postulates 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 $n \longrightarrow +\infty$, the faster is growing the time complexity over inputs of larger and larger size, the worse is its corresponding algorithm. 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. Therefore it is not surprising that most introductory books on Algorithms take the RAM model as a reference.

But in the last 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 have exploded in size ("$n \longrightarrow +\infty$" does not live only in the theory world!) because they are abundantly generated by many sources: such as DNA sequencing, bank transactions, mobile communications, Web navigation and searches, auctions, etc. etc.. The first change turned the RAM model into an unsatisfactory abstraction of modern PCs; whereas the second change made the design

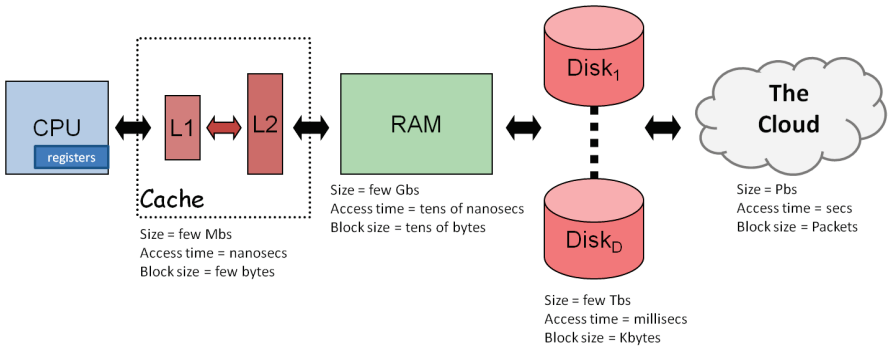

FIGURE 1.1: An example of memory hierarchy of a modern PC.

of *asymptotically good* algorithms ubiquitous and fruitful not only for dome-headed mathematicians but also for a much larger audience because of their impact on business [2], society [1] and science in general [4]. The net consequence was a revamped scientific interest in algorithmics and the spreading of the word "Algorithm" to even colloquial speeches!

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 their complexity analysis. Nowadays a modern PC consists of one or more CPUs (multi-core?) and a very complex hierarchy of memory levels, all with their own technological specialties (Figure 1.1): L1 and L2 caches, internal memory, one or more mechanical or SSDisks, 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 yet 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 by Tomas H. Cormen with his quote:

"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 the input/output subsystem to reduce the impact of the I/O-bottleneck onto the efficiency of applications managing large datasets; but, on the other hand, we are aware that the improvements achievable by means of a good algorithm design abundantly surpass the best expected technology advancements. Let us see the why with a simple example![1]

Assume to take three algorithms having increasing I/O-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 ith algorithm to process n input data (stored e.g. in n/B disk pages). Notice that the first two algorithms execute a *polynomial* number of I/Os (in the input size), whereas the last one executes an *exponential* number of I/Os. Moreover we note that the above complexities have a very simple (and thus unnatural) mathematical form because we want to simplify the calculations without impairing our final conclusions. Let us

[1] This is paraphrased from [8], now we talk about I/Os instead of steps.

now ask for how many data each of these algorithms is able to process in a fixed time-interval of size 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 t time, $\sqrt{t/c}$ data are processed by the second algorithm, and only $\log_2(t/c)$ data are processed by the third algorithm. 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 size or have distributions which favor efficient executions (think e.g. to the simplex algorithm). But, on the other hand, these examples are quite rare, and the much more stringent bounds on execution time satisfied by polynomial-time algorithms make them 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* search, whereas polynomial-time algorithms are generally made possible only through the gain of 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 to run the above algorithms with a better I/O-subsystem, say one that is k times faster, and ask: How many data can be managed by this new PC? To address this question we solve the previous equations with the time-interval set to the length $k \times t$, thus implicitly assuming to run the algorithms over the old PC but providing itself with k times more time. We get that the first algorithm perfectly scales by a factor k, the second algorithm scales by a factor $\sqrt{k}$, whereas the last algorithm scales *only of an additional term* $\log_2 k$. Noticeably the improvement induced by a k-times more powerful PC for an exponential-time algorithm is totally negligible even in the presence of impressive (and thus unnatural) technology advancements! Super-linear 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 PC induces a speed-up of a factor $\sqrt[\alpha]{k}$. Overall, it is not hazardous to state that the impact of a good algorithm is far beyond any optimistic forecasting for the performance of future (mechanical or SSD) disks.[2]

Given this *appetizer* about the "Power of Algorithms", let us now turn back to the problem of analyzing the performance of algorithms in modern PCs 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, 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 $A_j = A[j * b + 1, (j + 1) * b]$ for $j = 0, 1, 2, \ldots, n/b - 1$.[3] Then it sums all items in one block before moving to the next block that is s blocks apart on the right. Array A is considered as cyclic so that, when the next block lies out of A, the algorithm wraps around it starting again from its beginning. Clearly not all values of s allow to take into account all of A's blocks (and thus sum all of A's integers). Nevertheless we know that if s is co-prime with n/b then $[s \times i \mod (n/b)]$ generates 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 specialty of this parametrization is that by varying s and b we can sum according to different patterns of memory accesses: from the sequential scan indicated above (setting $s = b = 1$), to a block-wise

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

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

access (set a large b) and/or a random-wise access (set a large s). Of course, all algorithms $\mathcal{A}_{s,b}$ are equivalent from a computational point of view, since they read exactly n integers and thus take n steps; but from a practical point of view, they have different time performance which becomes more and more significant as the array size n grows. The reason is that, for a growing n, data will be spread over more and more memory levels, each one 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 that 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 algorithm design and analysis. In a previous example we already argued that the number of I/Os is a good estimator for the time complexity of an algorithm, given the large gap existing between disk- and internal-memory accesses. This is indeed what is captured by the so called *2-level memory model* (aka. disk-model, or external-memory model [11]) which abstracts the computer as composed by only *two memory levels*: the internal memory of 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 coming from 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 abstracts 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: (a) the number of accesses to disk pages (hereafter *I/O*s), (b) the internal running time (CPU time), and (c) the number of disk pages used by the algorithm as its working space. This 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 onto the disk(s) that makes each accessed disk-page as much useful as possible; the latter imposes to execute as much useful work as possible onto data fetched in internal memory, before they are written back to disk.

In the light of this new model, let us re-analyze the time complexity of algorithms $\mathcal{A}_{s,b}$ by taking into account I/Os, given that the CPU time is still n and the space occupancy is n/B pages. 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 by taking n/B I/Os, independently of the value of b. As s and b change the situation complicates, but by not much. Fix $s = 2$ and pick 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 algorithm $\mathcal{A}_{2,b}$ examines 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 to any s by writing a cost of sn/B I/Os, which correctly gives n/B in the simplest cases dealt with above. This formula provides a better approximation of the real time complexity of the algorithm, although it does not capture all features of the disk. In fact, it considers all I/Os as *equal*, independently of their distribution. This is clearly unprecise because on real disks the *sequential* I/Os are faster than the *random* I/Os.[4] Referring to the previous example, the algorithms $\mathcal{A}_{s,B}$ have still 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 larger and larger s. As a result, we can conclude that even the 2-level memory model is an approximation of the behavior of algorithms on

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

real computers, although it results sufficiently good that it has been widely adopted in the literature to evaluate their performance on massive datasets. So that, in order to be as much precise as possible, we will evaluate in these notes our algorithms by specifying not only the number of executed I/Os but also characterizing their *distribution* (random vs contiguous) over the disk.

At this point one could object that given the impressive technological advancements of the last years, the 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 be fit into it, thus reducing significantly the case of an I/O-fault. We will argue that an even small portion of data resident to disk makes the algorithm slower than expected, and so, 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 onto 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, without renouncing to a meaningful conclusion, we assume that $p(\epsilon)$ is the probability of an I/O-fault. As an example, if $p(\epsilon) = 1$ then the algorithm always accesses its data on disk (i.e. one of the ϵM items); if $p(\epsilon) = \frac{\epsilon}{1+\epsilon}$ then the algorithm has a fully-random behavior in accessing its input data (since, from above, it is $p(\epsilon) == \frac{\epsilon}{1+\epsilon} = \frac{\epsilon M}{(1+\epsilon)M} = \frac{\epsilon M}{n}$); finally, if $p(\epsilon) = 0$ then the algorithm has a working set smaller than the internal memory size, and thus it does not execute any I/Os. Overall $p(\epsilon)$ measures the *un-locality* of the memory references of the analyzed algorithm.

To complete the notation, let us indicate with c the time needed for 1 I/O— we have $c \approx 10^5 - 10^6$, see above— and we set a to be the fraction of steps that induce a memory access in the running algorithm (this is typically $30\% - 40\%$, according to [6]). Now we are ready to estimate the *average cost of the step* for an algorithm working in this scenario:

$$1 \times \mathcal{P}(\text{computation step}) + t_m \times \mathcal{P}(\text{memory-access step}),$$

where t_m is the average cost of a memory access. To compute 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 $t_m = 1 \times (1 - p(\epsilon)) + c \times p(\epsilon)$. Observing that $\mathcal{P}$(memory-access step) + $\mathcal{P}$(computation step) = 1, and plugging the fraction of memory accesses $\mathcal{P}$(memory access step) = a, we derive the final formula:

$$(1-a) \times 1 + a \times [1 \times (1 - p(\epsilon)) + c \times p(\epsilon)] = 1 + a \times (c-1) \times p(\epsilon) \geq 3 \times 10^4 \times p(\epsilon).$$

This formula clearly shows that, even for algorithms exploiting locality of references (i.e. a small $p(\epsilon)$), the slowdown may be significant and actually it turns out to be four order of magnitudes larger than what might be expected (i.e. $p(\epsilon)$). Just as an example, take an algorithm that exploits locality of references in its memory accesses, say 1 out of 1000 memory accesses is on disk (i.e. $p(\epsilon) = 0.001$). Then, its performance on a massive dataset that is stored on disk would be slowed down by a factor > 30 with respect to a computation executed completely in internal memory.

It goes without saying that this is just the tip of the iceberg, because the larger is the amount of data to be processed by an algorithm, the higher is the number of memory levels involved in the storage of these data and, hence, the more variegate are the types of "memory faults" (say cache-faults, memory-faults, etc.) to cope 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 on large input data.

Motivated by these premises, these notes will provide few examples of challenging problems which admit elegant algorithmic solutions whose efficiency is crucial to manage the large datasets

that occur in many real-world applications. 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. Too many times, as a theoretician, I got the observation that "your algorithm is far from being amenable to an efficient implementation!". By following the recent surge of investigations in *Algorithm Engineering* [10] (to be not confused with the "practice of Algorithms"), we will also dig into the deep computational features of some algorithms by resorting few other successful models of computations— mainly the streaming model [9] and the cache-oblivious model [5]. 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 to say, we will be unsuccessful in turning this "rocket science for non-boffins" into a "science for dummies" [2]. In fact lots of many more things have to fall into place for algorithms to work: top-IT companies (like Google, Yahoo, Microsoft, IBM, AT&T, Oracle, Facebook, Twitter, etc.) are perfectly aware of the difficulty to find people with the right skills for developing and refining "good" algorithms. This booklet will scratch just the surface of Algorithm Design and Engineering, with the main goal of spurring inspiration into your daily job as software designer or engineer.

References

[1] Person of the Year. *Time Magazine*, 168(27–28), December 2006.

[2] Business by numbers. *The Economist*, September 2007.

[3] Wikipedia's entry: "Algorithm characterizations", 2009. At `http://en.wikipedia.org/wiki/Algorithm_characterizations`

[4] Declan Butler. *2020 computing: Everything, everywhere*, volume 440, chapter 3, pages 402–405. Nature Publishing Group, March 2006.

[5] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, September 2006.

[7] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 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), 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*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.

[11] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.