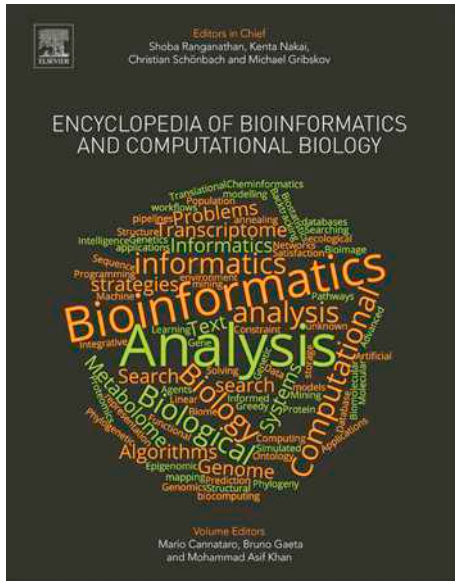
Algorithms Foundations

Nadia P Pisanti



Nadia Pisanti (2019) Algorithms Foundations. In: Guenther, R. and Steel, D. (eds.), Encyclopedia of Bioinformatics and Computational Biology, vol. 1, pp. 1–4. Oxford: Elsevier.

Algorithms Foundations

Nadia Pisanti, University of Pisa, Pisa, Italy

Introduction

Biology offers a huge amount and variety of data to be processed. Such data has to be stored, analysed, compared, searched, classified, etcetera, feeding with new challenges many fields of computer science. Among them, algorithmics plays a special role in the analysis of biological sequences, structures, and networks. Indeed, especially due to the flood of data coming from sequencing projects as well as from its down-stream analysis, the size of digital biological data to be studied requires the design of very efficient algorithms. Moreover, biology has become, probably more than any other fundamental science, a great source of new algorithmic problems asking for accurate solutions. Nowadays, biologists more and more need to work with *in silico* data, and therefore it is important for them to understand why and how an algorithm works, in order to be confident in its results. The goal of this chapter is to give an overview of fundamentals of algorithms design and evaluation to a non-computer scientist.

Algorithms and Their Complexity

Computationally speaking, a *problem* is defined by an input/output relation: we are given an input, and we want to return as output a well defined solution which is a function of the input satisfying some property.

An *algorithm* is a computational procedure (described by means of an unambiguous sequence of instructions) that has to be excuted in order to solve a computational problem. An algorithm solving a given problem is correct if it outputs the right result for every possible input. The algorithm has to be described according to the entity which will execute it: if this is a computer, then the algorithm will have to be written in a programming language.

Example: Sorting Problem
INPUT: A sequence S of n numbers $< a_1, a_2, \ldots, a_n >$.
OUTPUT: A permutation $< a_1', a_2', \ldots, a_n' >$ of S such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Given a problem, there can be many algorithms that correctly solve it, but in general they will not all be equally efficient. The efficiency of an algorithm is a function of its input size.

For example, a solution for the sorting problem would be to generate all possible permutations of S and, per each one of them, check whether this is sorted. With this procedure, one needs to be lucky to find the right sorting fast, as there is an exponential (in n) number of such permutations and in the average case, as well as in the worst case, this algorithm would require a number of elementary operations (such as write a value in a memory cell, comparing two values, swapping two values, etcetera) which is exponential in the input size n. In this case, since the worst case cannot be excluded, we say that the algorithm has an exponential *time complexity*. In computer science, exponential algorithms are considered *intractable*. An algorithm is, instead, *tractable*, if its complexity function is polynomial in the input size. The *complexity of a problem* is that of the most efficient algorithm that solves it. Fortunately, the sorting problem is tractable, as there exist tractable solutions that we will describe later.

In order to evaluate the running time of an algorithm independently from the specific hardware on which it is executed, this is computed in terms of the amount of simple operations to which it is assigned an unitary cost or, however, a cost which is constant with respect to the input size. A constant running time is a negligible cost, as it does not grow when the input size does; moreover, a constant factor summed up with a higher degree polynomial in n is also negligible; furthermore, even a constant factor multiplying a higher polynomial is considered negligible in running time analysis. What counts is the growth factor with respect to the input size, i.e. the *asymptotic* complexity T(n) as the input size n grows. In computational complexity theory, this is formalized using the *big-O* notation that excludes both coefficients and lower order terms: the asymptotic time complexity T(n) of an algorithm is in O(f(n)) if there exist n_0 and $c > 0$ such that $T(n) \leq c\, f(n)$ for all $n \geq n_0$. For example, an algorithm that scans an input of size n a constant number of times, and then performs a constant number of some other operations, takes O(n) time, and is said to have linear time complexity. An algorithm that takes linear time only in the worst case is also said to be in O(n), because the big-O notation represents an upper bound. There is also an asymptotic complexity notation $\Omega(f(n))$ for the lower bound: $T(n) = \Omega(f(n))$ whenever $f(n) = O(T(n))$. A third notation $\Theta(f(n))$ denotes asymptotic equivalence: we write $T(n) = \Theta(f(n))$ if both $T(n) = O(f(n))$ and $f(n) = O(T(n))$ hold. For example, an algorithm that *always* performs a linear scan of the input, and not just in the worst case, has time complexity in $\Theta(n)$. Finally, an algorithm which needs to at least read, hence scan, the whole input of size n (and possibily also perform more costly tasks), has time complexity in $\Omega(n)$.

Time complexity is not the only cost parameter of an algorithm: *space complexity* is also relevant to evaluate its efficiency. For space complexity, computer scientists do not mean the size of the program describing an algorithm, but rather the data structures this actually keeps in memory during its execution. Like for time complexity, the concern is about how much memory the execution takes in the worst case and with respect to the input size. For example, an algorithm solving the sorting problem without

requiring any additional data structure (besides possibly a constant number of constant-size variables), would have linear space complexity. Also the exponential time complexity algorithm we described above has linear space complexity: at each step, it suffices to keep in memory only one permutation of S, as those previously attempted can be discarded. This observation offers an example of why, often, time complexity is of more concern than space complexity. The reason is not that space is less relevant than time, but rather that space complexity is in practice a lower bound of (and thus smaller than) time complexity: if an algorithm has to write and/or read a certain amount of data, then it forcely has to perform at least that amount of elementary steps (Cormen *et al.*, 2009; Jones and Pevzner, 2004).

Iterative Algorithms

An *iterative algorithm* is an algorithm which repeates a same sequence of actions several times; the number of such times does not need to be known a priori, but it has to be finite. In programming languages, there are basically two kinds of iterative commands: the **for** command repeats the actions a number of times which is computed, or anyhow known, before the iteractions begin; the **while** command, instead, performs the actions as long as a certain given condition is satisfied, and the number of times this will occur is not known a priori. What we call here an *action* is a command which can be, on its turn, again iterative. The cost of an iterative command is the cost of its actions multiplied by the number of iterations.

From now on, in this article we will describe an algorithm by means of the so-called *pseudocode*: an informal description of a real computer program, which is a mixture of natural language and keywords representing commands that are typical of programming languages. To this purpose, before exhibiting an example of an iterative algorithm for the sorting problem, we introduce the syntax of a fundamental elementary command: the assignment "x←E", whose effect is to set the value of an expression E to the variable x, and whose time cost is constant, provided that computing the value of E, which can contain on its turn variables as well as calls of functions, is also constant. We will assume that the input sequence S of the sorting problem is given as an array: an array is a data structure of known fixed length that contains elements of the same type (in this case numbers). The i-th element of array S is denoted by S[i], and reading or writing S[i] takes constant time. Also swapping two values of the array takes constant time, and we will denote this as a single command in our pseudocode, even if in practice it will be implemented by a few operations that use a third temporary variable. What follows is the pseudocode of an algorithm that solves the sorting problem in polynomial time.

```
INSERTION-SORT(S,n)
  for i=1 to n−1 do
      j←i
      while (j>0 and S[j−1]>S[j])
          swap S[j] and S[j−1]
          j←j−1
      end while
  end for
```

INSERTION-SORT takes in input the array S and its size n. It works iteratively by inserting into the partially sorted S the elements one after the other. The array is indexed from 0 to n − 1, and a **for** command performs actions for each i in the interval [1, n − 1] so that at the end of iteration i, the left end of the array up to its i-th position is sorted. This is realized by means of another iterative command, nested into the first one, that uses a second index j that starts from i, compares S[j] (the new element) with its predecessor, and possibly swaps them so that S[j] moves down towards its right position; then j is decreased and the task is repeated until S[j] has reached its correct position; this inner iterative command is a **while** command because this task has to be performed as long as the predecessor of S[j] is larger than it.

Example: Let us consider S=[3,2,7,1]. Recall that arrays are indexed from position 0 (that is, S[0]=3, S[1]=1, and so on). INSERTION-SORT for i=1 sets j=1 as well, and then executes the while because j=1>0 and S[0]>S[1]: these two values are swapped and j becomes 0 so that the while command ends with S=[2,3,7,1]. Then a new **for** iteration starts with i=2 (notice that at this time, correctly, S is sorted up to S[1]), and S[2] is taken into account; this time the **while** command is entered with j=2 and its condition is not satisfied (as S[2]>S[1]) so that the **while** immediately ends without changing S: the first three values of S are already sorted. Finally, the last **for** iteration with i=4 will execute the **while** three times (that is, n − 1) swapping 1 with 7, then with 3, and finally with 2, leading to S=[1,2,3,7] which is the correct output.

INSERTION-SORT takes at least linear time (that is, its time complexity is in $\Omega(n)$) because all elements of S must be read, and indeed the **for** command is executed $\Theta(n)$ times: one per each array position from the second to the last. The invariant is that at the beginning of each such iteration, the array is sorted up to position S[$i-1$], and then the new value at S[i] is processed. Each iteration of the **for**, besides the constant time (hence negligible) assignment j←i, executes the **while** command. This latter checks its condition (in constant time) and, if the newly read element S[j] is greater than, or equal to, S[$j-1$] (which is the largest of the so far sorted array), then it does nothing; else, it swaps S[j] and S[$j-1$], decreases j, checks again the condition, and possibly repeates these actions, as long as either S[j] finds its place after a smaller value, or it becomes the new first element of S as it is the smallest found so far. Therefore, the actions of the **while** command are never executed if the array is already sorted. This is the best case time complexity of INSERTION-SORT: linear in the input size n. The worst case is, instead, when the input array is sorted in

the reverse order: in this case, at each iteration i, the **while** command has to perform exactly i swaps to let S[*j*] move down to the first position. Therefore, in this case, iteration i of the **for** takes i steps, and there are $n-1$ such iterations for each $1 \le i \le n-1$. Hence, the worst case running time is

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

As for space complexity, INSERTION-SORT works within the input array plus a constant number of temporary variables, and hence it has linear space complexity. Being n also a lower bound (the whole array must be stored), in this case the space complexity is optimal.

The algorithm we just described is an example of iterative algorithm that realises a quite intuitive sorting strategy; indeed, often this algorithm is explained as the way we would sort playing cards in one hand by using the other hand to iteratively insert each new card in its correct position. Iteration is powerful enough to achieve, for our sorting problem, a polynomial time – although almost trivial – solution; the time complexity of INSERTION-SORT cannot however be proved to be optimal as the lower bound for the sorting problem is not n^2, but rather $n \cdot \log_2 n$ (result not proved here). In order to achieve $O(n \cdot \log_2 n)$ time complexity we need an even more powerful paradigm that we will introduce in next section.

Recursive Algorithms

A *recursive algorithm* is an algorithm which, among its commands, recursively calls itself on smaller instances: it splits the main problem into subproblems, recursively solves them and combines their solutions in order to build up the solution of the original problem. There is a fascinating mathematical foundation, that goes back to the arithmetic of Peano, and even further back to induction theory, for the conditions that guarantee correctness of a recursive algorithm. We will omit details of this involved mathematical framework. Surprisingly enough, for a computer this apparently very complex paradigm, is easy to implement by means of a simple data structure (the *stack*).

In order to show how powerful induction is, we will use again our Sorting Problem running example. Namely, we describe here the recursive MERGE-SORT algorithm which achieves $\Theta(n \cdot \log_2 n)$ time complexity, and is thus optimal. Basically, the algorithm MERGE-SORT splits the array into two halves, sorts them (by means of two recursive calls on as many sub-arrays of size n/2 each), and then merges the outcomes into a whole sorted array. The two recursive calls, on their turn, will recursively split again into subarrays of size n/4, and so on, until the base case (the already sorted sub-array of size 1) is reached. The merging procedure will be implemented by the function MERGE (pseudocode not shown) which takes in input the array and the starting and ending positions of its portions that contain the two contiguous sub-arrays to be merged. Recalling that the two half-arrays to be merged are sorted, MERGE simply uses two indices along them sliding from left to right, and, at each step: makes a comparison, writes the smallest, and increases the index of the sub-array which contained it. This is done until when both sub-arrays have been entirely written into the result.

```
MERGE-SORT(S,p,r)
  if p<r then
      q ← ⌊(p+r)/2⌋
      MERGE-SORT(S,p,q)
      MERGE-SORT(S,q+1,r)
      MERGE(S,p,q,r)
  end if
```

Given the need of calling the algorithm on different array fragments, the input parameters, besides S itself, will be the starting and ending position of the portion of array to be sorted. Therefore, the first call will be MERGE-SORT(S,0,n − 1). Then the index q which splits S in two halves is computed, and the two so found subarrays are sorted by means of as many recursive calls; the two resulting sorted arrays of size n/2 are then fused by MERGE into the final result. The correctness of the recursion follows from the fact that the recursive call is done on a half-long array, and from the termination condition "p < r": if this holds, then the recursion goes on; else (p = r) there is nothing to do as the array has length 1 and it is sorted. Notice, indeed, that if S is not empty, then $p > r$ can never hold as q is computed such that $p \le q < r$.

The algorithm MERGE-SORT has linear (hence optimal) space complexity as it only uses S itself plus a contant number of variables. The time complexity T(n) of MERGE-SORT can be defined by the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & \rightarrow n = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \rightarrow n > 1 \end{cases}$$

because, with an input of size n, MERGE-SORT calls itself twice on arrays of size n/2, and then calls MERGE which takes, as we showed above, $\Theta(n)$ time.

We now show by induction on n that $T(n) = \Theta(n \cdot \log_2 n)$. The base case is simple: if $n = 1$ then S is already sorted and correctly MERGE-SORT does nothing and ends in $\Theta(1)$ time. If $n > 1$, assuming that $T(n') = \Theta(n' \cdot \log_2 n')$ holds for $n' < n$, then we have

$T(n) = 2 \cdot n/2 \cdot \log_2(n/2) + n = n(\log_2 n - \log_2 2 + 1)$, which is in $\Theta(n \cdot \log n)$. It follows that MERGE-SORT has optimal time complexity.

Closing Remarks

In this article we gave an overview of algorithms and their complexity, as well as of the complexity of a computational problem and how the latter should be stated. We also described two fundamental paradigms in algorithms design: iteration and recursion. We used as running example a specific problem (sorting an array of numbers) to exemplify definitions, describe algorithms using different strategies, and learning how to compute their complexity.

See also: Information Retrieval in Life Sciences

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to Algorithms, third ed. Boston, MA: MIT Press.
Jones, N.C., Pevzner, P.A., 2004. An Introduction to Bioinformatics Algorithms. Boston, MA: MIT Press.

Further Reading

Mäkinen, V, Belazzougui, D, Cunial, F, 2015. Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge: Cambridge University Press.

Biographical Sketch

Nadia Pisanti graduated cum laude in Computer Science at the University of Pisa in 1996. In 1998 she obtained a DEA degree at the University of Paris Est, and in 2002 a PhD in Informatics at the University of Pisa. She has been visiting fellow at the Pasteur Institute in Paris, ERCIM fellow at INRIA Rhone Alpes, research fellow at the University of Pisa, and CNRS post-doc at the University of Paris 13. Since 2006 she is with the Department of Computer Science of the University of Pisa. During the academic year 2012–2013 she was on sabbatical leave at Leiden University, and during that time she has been visiting fellow at CWI Amsterdam. Since 2015, she is part of the international INRIA team ERABLE. Her research interests fall in the field of Computational Biology and, in particular, in the design and application of efficient algorithms for the analysis of genomic data.