Design and Analysis of Algorithms Reconsidered

Anany Levitin
Department of Computing Sciences
Villanova University
Villanova, PA 19085, USA
anany.levitin@villanova.edu

Abstract

The paper elucidates two views (models) of algorithmic problem solving. The first one is static; it is based on the identification of several principal *dimensions* of algorithmic problem solving. The second one is dynamic, i.e., it catalogs main steps in the *process* of solving a problem with a computer. The models are used to identify several important issues in teaching design and analysis of algorithms and to suggest ways of rectifying the shortcomings identified.

1 Introduction

Once the notion of algorithm came to be recognized as the cornerstone of computer science, a course on design and analysis of algorithms became a standard requirement in computer science curricula. Judging by the contents of widely used textbooks, such a course follows one of two alternatives in presenting the subject by grouping algorithms either by problem types (e.g., [10, 11]) or by underlying design techniques (e.g., [3, 5, 8]). (There are also textbooks such as [4] that try to straddle the two approaches.) Each of these two options has its obvious strengths and weaknesses [1]; for example, the second one is arguably more conducive to teaching design techniques. As for the analysis side, all the textbooks follow the well-established framework of asymptotic analysis of time-, and to a much lesser degree, space- efficiency.

We will contend in this paper that, no matter which approach is used, the established treatment of design and analysis of algorithms has serious deficiencies and limitations. We will point out these deficiencies and limitations — and suggest ways to alleviate or eliminate them — in Section 4 of the paper after outlining two views of the algorithmic problem solving in Sections 2 and 3, respectively.

2 A Static View of Algorithmic Problem Solving

One can easily identify a few major aspects (or dimensions) of algorithmic problem solving. They are:

- computational means (the principal choices of interest here are sequential vs. parallel computers; more exotic options include a human and such abstract devices as the Turing machine)

- exact vs. approximate solving[1]

- deterministic vs. probabilistic paradigm

- data structure(s)

- algorithm design technique

There are three dimensions that are applicable to an algorithm in the analysis, as opposed to design, stage:
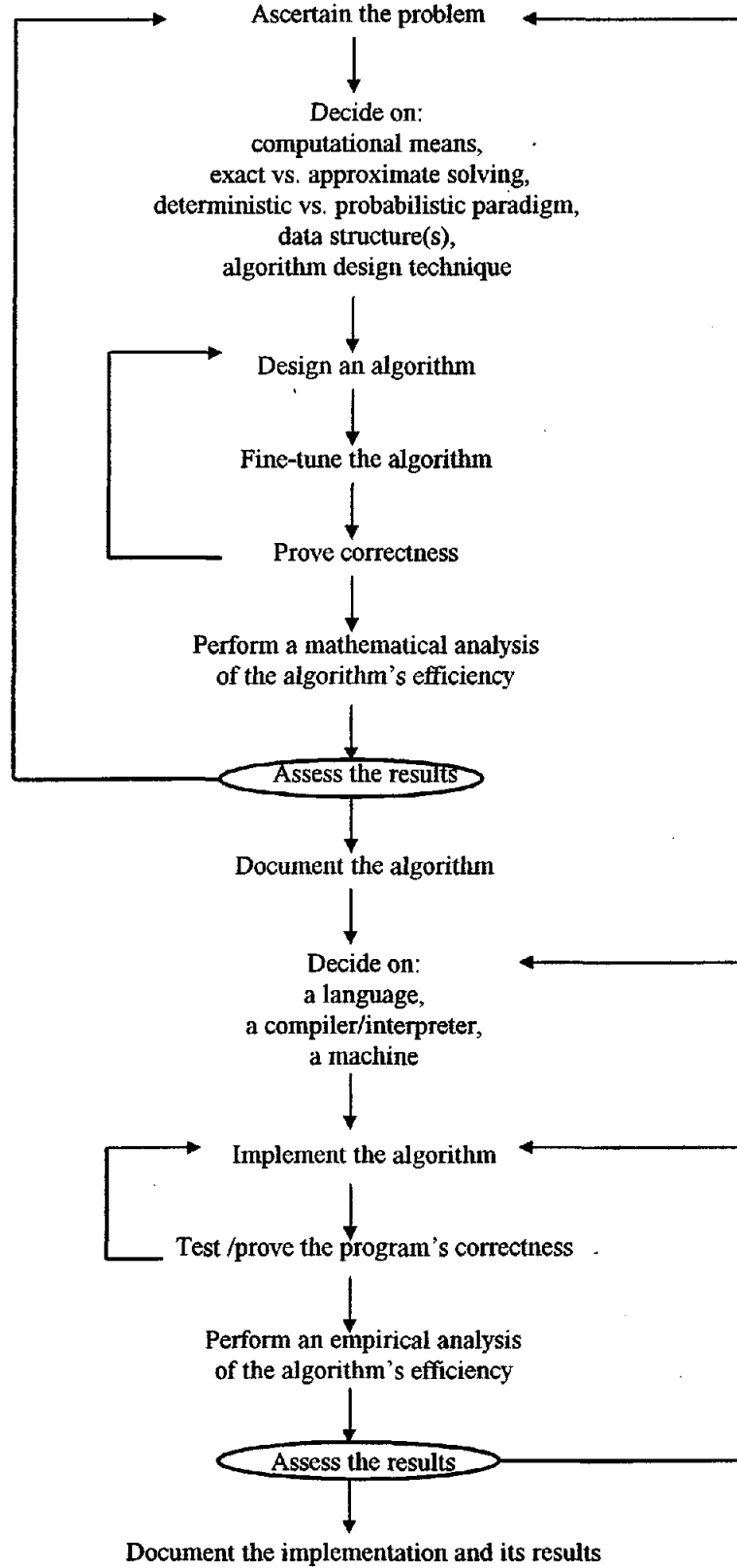
- correctness (and a degree of error for an approximation algorithm)

- efficiency (time and space)

- clarity (we will consider "simplicity" to be synonymous with clarity").

Finally, there is one more dimension of algorithmic problem solving: optimality.

---

[1] There are several reasons why one might be interested in solving a problem approximately. First, there are many important problems (e.g., solving nonlinear equations) that cannot be solved exactly by any algorithm. Second, available algorithms for solving a problem exactly can be unacceptably slow (e.g., *NP*-hard problems). Third, one could be interested in finding an approximate solution to a problem even if an exact solution can be found efficiently (e.g., one might want to solve the string-searching problem approximately in order to identify possible misspellings of a given word).

3 A Dynamic View of Algorithmic Problem Solving

be modeled by the diagram below:

A dynamic view of solving a problem with a computer can

Ascertain the problem

Decide on:
computational means,
exact vs. approximate solving,
deterministic vs. probabilistic paradigm,
data structure(s),
algorithm design technique

Design an algorithm

Fine-tune the algorithm

Prove correctness

Perform a mathematical analysis
of the algorithm's efficiency

Assess the results

Document the algorithm

Decide on:
a language,
a compiler/interpreter,
a machine

Implement the algorithm

Test /prove the program's correctness

Perform an empirical analysis
of the algorithm's efficiency

Assess the results

Document the implementation and its results

17

4 Current Treatment and Possible Improvements

- Interpreting algorithms as solutions to problems is, of course, intrinsic to the notion of algorithm. However, in our opinion, this perspective needs to be emphasized more strongly than is typically done. (In fact, one can argue that algorithmic problem solving is a broader topic than design and analysis of algorithms. An inspection of the list of dimensions of the static model and of the steps of the dynamic model would seem to support such a conclusion.)

  Emphasis on problem solving can be strengthened in several ways. First, organizing a course around design techniques, which can be treated as problem solving strategies, would be more appropriate than organizing the course around problem types. Second, all the aspects of algorithmic problem solving implied by the static and dynamic models should be covered. (We will elaborate on this point in more detail below.) Third, some general ideas about problem solving — along the lines of Polya's classic [9] — might enhance the course. Finally, more examples, especially from real-life applications, ought to be presented. (Skiena's book [12] can be very useful here.)

- The *process* of algorithmic problem solving is not usually discussed in algorithm textbooks. In particular, the implementation part of the process is almost always ignored, probably under the assumption that it has been covered in preceding programming courses.

  Though the dynamic (process) model above simply lists the common steps of algorithmic problem solving, it has two benefits. First, it reminds students of the "obvious" steps of problem solving with a computer. Of course, as any model, this one does not perfectly reflect reality. For example, for the sake of simplicity, we have decided against inclusion of the fine-tuning step in the programming stage of the process. We strongly advocate not the specifics of the dynamic model above but rather the fact that such a model needs to be discussed in an algorithm course.

  Second, and most important, the model emphasizes the point that getting a good algorithmic solution is an *iterative process*, which typically requires repeated rework along the way. Regrettably, most textbooks fail to make this point. Fortunately, some interesting and poignant examples have been published elsewhere (see, e.g., [2], [13]).

  As for the many issues arising in the implementation stage of the process (see the entire bottom half of the dynamic model's diagram), they deserve at least a brief review, in our opinion. A recently published book by Kernigan and Pike [6] contains a lot of up-to-date information on these issues.

- Algorithm design is taught either by default by simply exposing students to well-known algorithms or by grouping algorithms along a few general design techniques whose traditional classification has very serious shortcomings.

  There is an unfortunate dichotomy between our knowledge about general principles of algorithm design versus their analysis. While the latter has been firmly established for quite some time now, the former has been almost completely neglected by computer science researchers. At the same time, it is easy to make the case that teaching algorithm design should have at least the same, if not a higher, priority than algorithm analysis. (After all, one needs to have an algorithm first before it can be analyzed!)

  There is some anecdotal evidence that issues of algorithm design have been starting to gain more attention. Thus, updates of several textbooks organized by design techniques have been published recently [3, 5, 8], while the textbooks organized by problem types usually include nowadays a chapter reviewing major design strategies. These developments are obviously most welcomed; however, no matter which of the two exposition options is pursued, it is imperative that a better taxonomy of design techniques is followed.

  The commonly used list of design techniques consists of five strategies: divide-and-conquer, greedy approach, dynamic programming, backtracking, and branch-and-bound. Among several shortcomings of this taxonomy, the most glaring one is its inability to classify many important algorithms (e.g., Euclid's algorithm, heapsort, hashing, Gaussian elimination). A much more versatile classification scheme has been recently proposed by Levitin [7]. It has a hierarchical structure that, on its highest level, divides the strategies into more general and less general ones. The first group consists of brute force, divide-and-conquer, decrease-and-conquer, and transform-and-conquer; the second one includes local search techniques (with greedy algorithms and improvement methods as special cases), dynamic programming, and state-space-tree techniques (which include backtracking and branch-and-bound).

- Fine-tuning of an algorithm is not discussed, at least systematically.

  By fine-tuning, we mean improving the implementation details of an algorithm's idea. Of course, many such

improvements should, by necessity, be intimately related to the peculiarities of the algorithm in question. Still, a variety of standard "tricks" are well-known (see, e.g., [2]). Of particular interest and importance are, of course, adjustments that seek to optimize algorithms' innermost loops. In addition to some practical value as tools for improving algorithms' running time, their discussion goes to the heart of the asymptotic efficiency framework and the nature of multiplicative (hidden) constants.

Algorithm analysis is mostly limited to the well-established framework of time- (and, to a much lesser degree, space-) efficiency analysis. No other dimension is considered systematically in analyzing algorithms.

But the American Heritage Dictionary defines "analysis" as "the separation of an intellectual or substantial whole into its constituent parts for *individual study*" (the emphasis is added). Accordingly, the static model above explicates several dimensions of algorithmic problem solving, with efficiency being just one of them. Granted, efficiency is the dimension that we can analyze mathematically. But this fact should not prevent us from discussing the other dimensions as well.

The computational model, determinism vs. randomization, and the underlying data structure(s) can be, of course, identified immediately for an algorithm under study. But a discussion of making an alternative choice for one or more of these dimensions can be quite beneficial. (What will happen if the data structure underlying the algorithm changes? Can the algorithm benefit from introducing a randomization step? How can the algorithm be implemented on a parallel computer?) Whether the algorithm always yields an exact solution may or may not be trivial for a student to answer. And the question about the design technique the algorithm is based upon can have the double benefit of checking students' understanding of both the algorithm in question and the general design techniques. (A better taxonomy of algorithm design techniques can significantly expand the range of algorithms for which such a question has a meaningful answer.)

Finally, clarity is probably the most controversial dimension of an algorithm. Researchers in software engineering have been trying for years to come to grips with the closely related notion of complexity or comprehensibility of a computer program. In particular, several metrics seeking to quantify it have been proposed, running the gamut from metrics based on the program's size to those seeking to quantify complexity of the program's control flow. Unfortunately, no measures of this elusive quality have been agreed upon; therefore it is difficult to insist on a systematic discussion of clarity for each and every algorithm. Still, avoiding this dimension altogether is hardly productive either: for example, the principal attraction of brute-force algorithms is arguably due to their simplicity. Of course, the pursuit of simplicity/clarity may contradict the quest for efficiency. This provides a good opportunity to discuss the inevitability of occasional tradeoffs, which are a landmark of any design activity.

Finally, on the subject of tradeoffs, the important space-for-speed one is barely mentioned in modern algorithm textbooks. This omission ought to be rectified.

- Empirical analysis of efficiency is usually mentioned only in passing, if at all. In particular, a clear distinction between timing (measuring the overall running time) and profiling (identifying bottlenecks) is not always made. Means of empirical data presentation (tabular and graphical) and methods of their analysis are not usually reviewed.

In our view, empirical analysis deserves more attention than it currently receives in algorithm textbooks. (In order to stress the importance of empirical analysis, we have included it as a standard step in our dynamic model above.) Though its limitations vis-à-vis the mathematical analysis of efficiency are self-evident, so are its unique strengths. Besides, it provides a valuable opportunity to show students the empirical side of computer science (and to do so in the midst of a rather theoretical course). In addition, it allows an instructor to expose students to working with empirical data — an important skill that many computer science students, in our experience, sorely lack. Programming projects in particular seem to present a natural and convenient way for implementing this task.
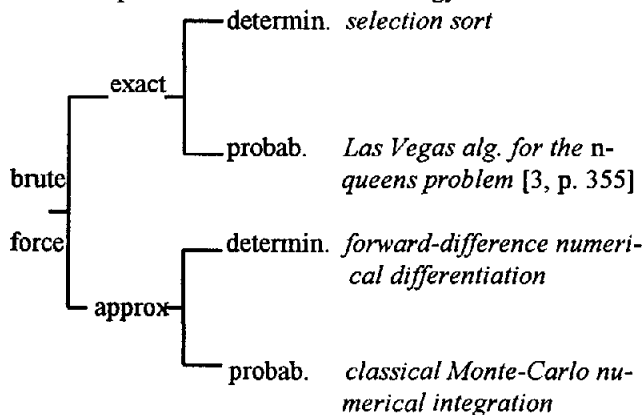
- Numerical analysis algorithms, especially from the area of continuous mathematics, are typically ignored (with the FFT algorithm being a notable exception).

In the early days of computing, numerical analysis was, of course, *the* computer science course. Then it became one of the required courses and now seems to have reached the ignominious status of an unpopular elective in many computer science programs. Though this diminishing interest clearly reflects the decreasing relative importance of numerical computations for the industry and research, the fact that a student may get a computer science degree with no exposure to numerical analysis is highly regrettable.

19

An algorithm course seems to be a natural vehicle for rectifying this situation. Besides, the inclusion of numerical analysis algorithms can serve two other purposes. First, the majority of problems from this domain cannot be solved exactly by any algorithm, thus giving the strongest motivation possible for approximate problem solving. Second, such algorithms can serve as useful examples of the applicability of general design techniques to the realm of approximate problem solving (e.g., the bisection method and the method of false position as examples of decrease-and-conquer).

- Approximation and probabilistic algorithms are introduced with little if any regard for the general design techniques used in the introduction of other algorithms.

This treatment obscures the understanding of approximate and probabilistic problem solving as two different, orthogonal dimensions. One way to emphasize this point-of-view is to give (or ask students for) examples demonstrating all possible combinations of a design technique and these two dimensions. As an illustration, the following tree-like diagram gives such examples for the brute-force strategy:

```
                    ┌── determin.  selection sort
            ┌─ exact ┤
            │        └── probab.   Las Vegas alg. for the n-
 brute      │                      queens problem [3, p. 355]
 force ─────┤
            │        ┌── determin. forward-difference numeri-
            └─ approx ┤                cal differentiation
                     └── probab.   classical Monte-Carlo nu-
                                       merical integration
```

- Finally, the exciting area of algorithm visualization and animation has still not found its way into popular textbooks.

This is both regrettable and surprising. It is regrettable because this area is undeniably attractive to students. It is surprising because the overwhelming interest in visual programming in general and Java in particular should have made algorithm visualization and animation a quite attractive component of algorithms courses. With the juggernaut of the Web, which has a number of sites with attractive algorithm animations, this neglect should disappear in the near future.

5 Conclusion

Two views of algorithmic problem solving have been explicated in the paper. The static view simply lists the main dimensions of algorithmic problem solving; the dynamic view indicates the principal steps in the process of solving problems with a computer. Though mundane, they raise several important issues about standard ways of presenting the subject in courses on design and analysis of algorithms. The paper has pointed out these issues and made specific suggestions about ways of addressing them. Given the importance of the topic, the author hopes that the paper will initiate a discussion about the issues raised in the paper.

References

[1] Baeza-Yates, R. Teaching Algoirthms. *SIGACT News, 26* (December 1995), 51-59.

[2] Bentley, J. *Programming Pearls*. Addison-Wesley, 1986.

[3] Brassard, G. and Bratley, P. *Fundamental of Algorithmics*, Prentice-Hall, 1996.

[4] Cormen, T. et al. *Introduction To Algorithms*. MIT, 1992.

[5] Horowitz, et al. *Computer Algorithms*. Computer Science Press, 1996.

[6] Kernigan, B. and Pike, R. *The Practice of Programming*. Addison-Wesley, 1999.

[7] Levitin, A. Do we teach the right algorithm design techniques? in *Proc. SIGCSE '99* (March 1999), 179-183.

[8] Neapolitan, R.and Naimipour, K. *Foundations of Algorithms*. Jones and Bartlett, 2nd ed., 1997.

[9] Polya, G. *How To Solve It*. Princeton Univ. Press, 1957.

[10] Rawlins, J. *Compared To What?: an Introduction to the Analysis of Algorithms*. Comp. Sc. Press, 1992.

[11] Sedgewick, R. *Algorithms*. Addison-Wesley, 1988.

[12] Skiena, S. *The Algorithm Design Manual*. Springer Verlag, 1997.

[13] Vandervooerde, D. The maximal rectangle problem. *Dr. Dobb's Journal, 23* (April 1998), 28-32.