The Problem of Examination Questions in Algorithmics

P.A. de Marneffe
Service d'Informatique (Algorithmique)
Institut Montefiore Université de Liège
B4000 Sart Tilman (Liège) Belgium
PA.deMarneffe@ulg.ac.be

1. ABSTRACT

Algorithmics is a problem solving activity. Examination questions must reflect this nature of the domain. They must lead to open answers, but the specific criteria used in grading these answers must be clearly understood by the students. In this paper, we explain which criteria we use in the context of a course on algorithmics given to first year students in Informatics and in Engineering. Our experience shows that the teaching of very important topics in Computing Science can never be automated.

2. ALGORITHMICS AND PROBLEM SOLVING

Algorithmics is a problem solving activity. But, in algorithmics, instead of solving a particular "one-shot" problem, we have to produce a way of systematically solving instances of problems belonging to a given class. Moreover, the systematic way (i.e. the algorithm) of solving any instance must be executable by a mechanism devoid of intelligence.

Algorithmics must be clearly disconnected from programming languages : the principles which form the hard core of the discipline are language-independent. There is no need to tie the domain to a specific programming language as it is frequently the case in published textbooks. Students must understand from the outset that they are learning general principles which are applicable to a multitude of present or future programming languages.

ITiCSE '98 Dublin, Ireland

3. CORRECTNESS ARGUMENTATION AND EFFICIENCY

As the designed algorithm must solve each instance of a problem in a given class, there must be an argumentation explaining that it is the case. The usual term "algorithm correctness" may be misleading; the students must understand that they have to produce a convincing argument proving that the algorithm they have written is correct. This argument must be set in a such way that it is comprehensible by other persons who share some common background with the designer of the algorithm. It helps if the presentation of the argumentation follows a method known and mastered by others : that allows criticisms.

Besides correctness, the efficiency of the designed algorithm is an important point. Students must understand that efficiency is predictable without any recourse to an actual run on an actual equipment. With the help of a purely mental model of a computer, it is possible to decide whether some design decision will increase the efficiency of the algorithm or not. The fact that a specified algorithm has an actual computing time which is logarithmic, linear, quadratic or cubic can be derived from a close study of the algorithm text. The efficiency of an actual computation is not simply dependent on the actual hardware implementation; usually, the decisions taken during the design of the algorithm will set limits upon the attainable efficiency on an actual equipment.

4. OUTLINE OF OUR LECTURES ON ALGORITHMICS

The audience is composed of university students in Informatics (first year) and in Engineering (second year). In order to apply the principles stated above, we use a programming meta-language, insisting on the use of loop invariants in the documentation of the correctness argumentation, and introducing several efficiency paradigms as guidelines for the design of algorithms.

4.1 Meta-Language

The meta-language is the "guarded command programming language", as described by E.W. Dijkstra in [1]. In this language, besides the assignment command, we have two types of commands : the alternative command and the repetitive one. This language is used for the description of

the several steps in the design of an algorithm. There is no implementation of the language, but it is explained how the repetitive and alternative commands can be "translated" into current (imperative) programming languages. In the programming laboratory, Pascal is used as implementation language, but it is made clear that, by suitable translation rules, any other imperative language can be used. The goal is to make a clear distinction between two tasks : on one hand, the design of an algorithm (in the meta-language), and on the other hand, the use of an actual programming language for the implementation of an already designed algorithm.

4.2 Correctness Argumentation

Emphasis is being placed on the use of loop invariants for the argumentation of the correctness. In nearly all textbooks, loop invariant is presented as an important method for correctness of algorithm, but it is hardly used (even in the same textbooks !). Usually, loop invariants are expounded in conjunction with the use of first-order logic formulas. There is no obligation to do so : the loop invariant may be expressed as a general predicate (even stated in natural language) or as a schematic graphical representation of the stated situation at the end of each iteration of the loop body (for instance, a schematic representation of the properties of the elements in sections of an array). The important point is the understanding that the general condition stated as the invariant of the loop must be true just before the first iteration and after each iteration, and that after the end of the repetitive command, the logical conjunction of the invariant and the negation of the loop boolean expression is true. For termination of the loop, we use t -function (as in [1]).

The respect of the rules of use of loop invariants is the common base between the algorithm designer and outside observers who have to be convinced by the argumentation of correctness. It is stated that these outsiders are expecting that the loop invariants are provided as a backbone of the correctness argumentation. (In practice, these outsiders are the lecturer and his assistants.)

The presentation of the design of the algorithm follows the general principle used in "literate programming" ([2]) : a sequence of program chunks which can be fitted together in order to compose the complete algorithm. A short explaining paragraph in natural language (with - if required - loop invariant in a suitable representation format) must accompany each program chunk.

4.3 Efficiency Paradigms

In order to act as guidelines in the design of an algorithm, two efficiency paradigms are introduced :

a) "divide and conquer", which is the basis of so many algorithms with a logarithmic complexity;

b) "to turn iteration to the best account", which occurs if an iteration can be made faster by using values computed by previous iterations; for instance, building a table of squares by using the fact that : $(n+1)^2 = (n^2+2n+1)$. This efficiency paradigm leads to the introduction of data structures needed for keeping some of the values computed by previous iterations.

4.4 Examples of Algorithms

The application of the three principles, (meta-language, correctness, efficiency), is exemplified by the study of classical algorithms : finding the integer square root of an integer, linear and dichotomic search in a table, sorting (bubblesort, heapsort, quicksort).

5. GRADING THE STUDENT ABILITY TO DESIGN ALGORITHMS

In order to grade the student ability to design algorithms, we use algorithmic problems based on a kind of pattern matching in two-dimensional array. The purpose of the problems is to detect the elements of the array which verify a given property. The property involves group of array elements; the number of elements belonging to one group is a parameter in the problem. A example of such a problem is given in the next section.

5.1 Example of an Algorithmic Problem

Let an array B[0..M-1, 0..N-1] with M rows and N columns, (0<M and 0<N). The elements of the array are integers.

Let an element B[i, j] and p an integer such that (p >=1).

By definition, B[i, j] is origin of a *peak(p)* if the following three conditions hold for the specified elements of the array:

1) The elements B[i, j], B[i-p, j+p] and B[i, j+2*p] belong to the array B.

2) B[i, j] is origin of an oblique composed of (p+1) elements for which the condition

($\underline{\mathbf{A}}$ k : 0<=k<p : B[i-k, j+k] <= B[i-(k+1), j+(k+1)]) is true.

3) B[i, j+2*p] is origin of an oblique composed of (p+1) elements for which the condition

($\underline{\mathbf{A}}$ k : 0<=k<p : B[i-k, j+2*p -k] <= B[i-(k+1), j+2*p - (k+1)]) is true.

From the definition of a *peak(p)*, it results that, considering all the possible values for p (p >=1), an element B[i, j] can be the origin of zero, one or many *peak(p)*.

Let an integer value *pval(i,j)* defined as follows :

a) if B[i, j] is not the origin of any *peak(p)*, *pval(i,j)* is equal to zero;

b) if B[i, j] is the origin of one or many *peak(p)*, *pval(i,j)* is equal to the maximum value of p among all the *peak(p)* the origin of which is B[i, j].

By definition, an element B[i, j] is called a T-element if the following two conditions hold :

1) *pval(i,j)* >=2

2) (*pval(i,j+1)* >= 1) **and** (*pval(i,j+1)* <= *pval(i,j)* -1)

Design an algorithm which :

1) determines the maximum value of the *pval(i,j)* for a given array B (the result must be in integer variable **pmax**) and the number of elements in B such that *pval(i,j)* is equal to the value of **pmax** (the result must be in integer variable **nbel**).

2) builds a linked list (access pointer **psk**) which contains one cell for each element B[i, j] which is a T-element such that *pval(i,j)* = **pmax** ; the cell stores the indices (i, j) of B[i, j] and the value of *pval(i,j+1)*.

If there is no T-element in the array B, the list is empty.

5.2 Discussion about this Example

There are many answers for the exam question (given in section 4.1). The students have room for exercising their creativity in the design of one algorithm among all the possible ones. They know that the answer will be graded according to the following criteria :

a) the algorithm must be expounded in a systematic manner (program chunks with suitable justifications), and the algorithm must be correct, with a convincing argumentation;

b) among the possible algorithms, a distinction is made between the simple ones and the advanced ones; a simple algorithm is an algorithm designed without application of an efficiency paradigm; an advanced algorithm tries to turn iterations to the best account and to exploit some properties in the problem statement in order to reduce the computation load.

The students know that a simple algorithm is marked 15/20 at most, advanced algorithm will be marked in the bracket (16/20 to 20/20) in proportion to its efficiency and its originality in the solution. Incorrect advanced algorithms are marked below correct simple algorithms; correctness takes precedence over efficiency.

For instance, for the given problem, a simple algorithm will compute *pval(i,j)* for each B[i,j], starting from scratch for each element. An advanced algorithm could use auxiliary arrays in order to store the length (in number of elements) of the increasing sections of the obliques; besides that, by scanning the array row by row starting from row (M-1), it is possible to reduce the number of array elements which are examined by the algorithm in function of the current value of **pmax**, (similarly, some columns can be skipped). An auxiliary array containing the pval*(i,j)* values can be built explicitly and used for the building of the linked list, or a current linked list can be maintained and updated during the scanning. Elaborated solutions can be devised by scanning the obliques from the element B[i-*p*, j+*p*], etc.

The problem stated in the example may seem quite abstract; this abstractness is a deliberate choice; the students must concentrate on the two aspects that we consider as essential for the design of algorithms : correctness and efficiency. Moreover, problem solving abilities are involved in the way the student decides to tackle the design of the algorithm. The designed algorithm must follow a plan, and the structure of this plan must be based on some rationale. We insist on the exposition of the reasons behind the several decisions taken during the design.

6. INTEGRATING TECHNOLOGY INTO EDUCATION

Our experience shows that there exist limits to the integration of technology into computing science education. The kind of examination questions we use is not suitable for automatic correction or automatic grading by computer.

We believe that in very intellectual domains, such as the design of algorithms, the students must have the opportunity to submit their own answers which may not belong to a set of known solutions. In scientific domains, topics beyond the pure training in skills cannot be automated. All knowledge with durable contents is far beyond pure training. In this case, the introduction of technology is limited to the use of some basic tools such as text editors, compilers.

7. CONCLUSIONS

We have outline the basic structure of a course on algorithmics geared to the acquisition of what we call "algorithmic abilities" : expounding the correctness of a designed algorithm and taking decisions for improving its efficiency. Problem statements are selected in order to induce students to practice these algorithmic abilities in a creative manner. The way the answer is marked follows a simple pattern which is well-understood by the students : correctness comes first (they may decide to develop a simple program and get good marks), efficiency is second (but it may lead to better marks). The main lesson we try to convey to the students is that designing an algorithm is not producing lines of code in a specified programming language, but is a very intellectual activity where the rationale behind the final product must be well-argumented and carefully expounded.

8. REFERENCES

[1] Dijkstra, E.W.; A discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J. 1976; 217 p.

[2] Knuth, D.E.; Literate Programming, The Computer Journal, Vol. 27 (1984), pp. 97-111.