Efficiency of algorithms for programming beginners

Conference Paper *in* ACM SIGCSE Bulletin · March 1996

DOI: 10.1145/236462.236551 · Source: DBLP

1 author:

David Ginat
Tel Aviv University
115 PUBLICATIONS **485** CITATIONS

SEE PROFILE

EFFICIENCY OF ALGORITHMS FOR PROGRAMMING BEGINNERS

David Ginat
Science Teaching Department
Weizmann Institute of Science
Rehovot, Israel
email: ntginat@weizmann.weizmann.ac.il

ABSTRACT

Introductory computer science courses often present the concept of algorithm efficiency in a rather late stage, through searching and sorting problems. We believe that the concept of efficiency can be presented much earlier. In this paper we present a novel approach in which efficiency of algorithms is presented early, gradually and intuitively. We link our approach to cognitive consequences of programming instruction and illustrate it with three problems.
We implemented our approach with 10th and 11th grade high-school students. In the paper, we describe our experience with posing the three problems to the students. Student solutions varied considerably, reflecting different levels of insight into the problems. The various solutions led to fruitful class discussions, thus widening the students' repertoire of templates, and enhancing student realization as to the importance of analysis and planning.

1. INTRODUCTION

The goal of most introductory programming courses is to teach the basic knowledge and skills of program design. This includes well-defined language features as well as general problem solving skills such as analysis and planning. Some courses include in their curriculum a primary introduction to the core computer science topics - program correctness and algorithm efficiency.

Introduction to program correctness in rather early stages of programming is becoming more common [5,1,12], although the debate regarding the extent of formal methods is still open [4]. Introduction to algorithm efficiency in early stages of programming is less common.

SIGCSE '96 2/96 Philadelphia, PA USA

Efficiency of algorithms is usually introduced in a rather late chapter of introductory courses [2,3,9], if at all. When introduced, it is often presented through searching and sorting algorithms, after the students have already seen a wide range of loop constructs and array uses. Efficiency is then introduced either on an intuitive level or, formally, through complexity measures and "Big O" notation.

However, convincing the students that there are less efficient and more efficient ways to go about doing things can be done much earlier in the introductory course. In fact, efficiency can be first introduced shortly after loops are presented, and can accompany the studied material thereafter. By this, a deeper computer science perspective can be enhanced already from early stages of programming.

In this paper, we present an approach of early, gradual, and intuitive introduction of algorithm efficiency, and describe our initial experience with introducing it to pre-college students.

In section 2, we briefly describe our approach, linking it to cognitive consequences of programming instruction. In sections 3, 4, and 5 we present three problems (one in each section) that represent our approach, and describe our experience with posing the problems to 10th and 11th grade high-school students.

2. EARLY INTRODUCTION OF EFFICIENCY

We first mention several relevant studies regarding the cognitive consequences of programming instruction. Linn [6] divides design skills to knowledge of repertoire of templates and to procedural skills. Templates are stereotypic patterns of code, and procedural skills include planning a solution using templates, testing the plan, and possibly reformulating it. Spohrer and Soloway [11] propose that expert programmers organize their knowledge in plans that indicate the steps necessary to perform computing actions. Linn, Soloway, and others indicate that experts spend a great deal of time planning problem solutions, and among novice programmers, the more successful are those who can construct a well formulated program plan.

Linn and Clancy [7], suggest that introductory courses may reinforce the emphasis on syntax and low-level details rather than recognition and reuse of patterns. They advocate discussions regarding the decisions made while designing programs. They encourage the generation and evaluation of alternative designs, thus widening the repertoire of templates and improving procedural skills. Schoenfeld [10] demonstrates similar principles for teaching mathematical problem solving. He emphasizes the importance of analysis and reanalysis during pattern exploration and plan composition.

In our approach, introduction of algorithm efficiency is combined with widening the repertoire of templates, devising plans, evaluating alternative designs, and performing analysis and reanalysis. The programs developed, and examined for efficiency considerations, include templates that are employed as entities in future programs for commonly encountered tasks. During program design, students perform problem analysis and plan formulation. Different designs offer constructive discussions on efficiency considerations, enabling capitalization on useful ideas and suppression of unproductive ideas. In particular, students can realize the importance of analysis and planning, in the sense that better insight into a task yields a better program.

We concentrate on the concept of efficiency with respect to running-time (and not space) of programs. Soon after students learn to write loops, they are referred to count the number of iterations made by their constructed loops. Initially, examined programs for counting the number of loop iterations are programs like "compute a sum" and "perform division by subsequent substractions".

Students first count the number of iterations for specific input examples, and then generalize by formulating an expression based on the input size. The expressions for the examined programs are rather simple expressions like $N, N^2, N/2, N/M$, and $\sqrt{N}$.

The problem through which the concept of efficiency is first introduced is "check whether a given number N is a prime number". The students are shown the loop where the remainder of dividing N by i, the loop counter, is checked. At first, the bound selected for the loop counter is N. Then, the bound is reduced to N/2. Finally, the bound is reduced to $\sqrt{N}$ (if $N=x \cdot y$, then either x or y is no bigger than $\sqrt{N}$). This problem offers a rather simple illustration of bound reduction and the notion of best-case (N is even) and worst-case (N is prime).

In our approach, efficiency considerations are introduced gradually, through increasingly-more-subtle problems. By this, we enhance the development of computer-science thinking and seeing along the course.

We implemented our approach with 10th and 11th grade high-school students, using the programming language Pascal. The students' mathematical ability varied, and many were of average ability.

In the following sections we present three problems of increasing difficulty that characterize our approach. The first two problems are sequence generation problems, and the third is a search problem. We posed the three problems to the students at different stages of their introductory course. We did not expect all students to come up with the most efficient solution. On the contrary, we expected various solutions that would reflect different levels of insight. And so it was! We describe the various student solutions, that led to constructive discussions in class.

3. SEQUENCE OF MULTIPLES

The following problem was posed to the students shortly after loops and efficiency were introduced.

> *Sequence of Multiples.* Given two positive integers, x and y where $y > x$, output in decreasing order, all the multiples of x that are smaller or equal to y. For example, for the input 100 550, the output should be: 500 400 300 200 100.

Although this problem may look trivial even to programming beginners, we received various solutions that varied considerably "efficiency-wise", according to students' insight into the problem.

The first type of solutions were "brute-force" solutions of students who had little mathematical insight into the problem. The typical loop in these solutions was:

```
for i:=y downto x do
   if (i mod x) = 0 then write (i)
```

The second type of solutions were solutions that reflected more insight, but lacked some computation competence. These solutions included a loop with proper decrements, for generating the desired sequence. But, the computation of the first element of the sequence was not done by an expression, but rather with an additional loop.

```
i := y;
while (i mod x) <> 0 do
    i := i-1;
while (i >= x) do begin
    write (i);
    i := i-x
end
```

The students that solved the problem this way realized that the extra loop degrades efficiency, but they had a difficulty computing the first element using an expression.

The third type of solutions were variants of the efficient solution:

```
e := y - (y mod x);
for i := (e div x) downto 1 do begin
   write (e);
   e := e-x
end
```

The different answers were evaluated and discussed in class. The students calculated the number of loop iterations performed by each solution and realized the difference between the expressions: y-x - for the first solution, y mod x + y div x - for the second solution, and y div x - for the third solution.
In addition, they executed the various solutions and observed the gaps in response times between the various executions for inputs like 2000 19500.

4. THE MIDGETS AND THE LIGHT-BULBS

The following problem was posed to the students after they had some experience with loops, and initial experience with arrays.

> *The Midgets and the Light-Bulbs.* N Midgets are turning on-and-off a line of N light-bulbs. The first midget turns on all the N light-bulbs, the second turns off all the light-bulbs that are in even positions, the third switches the state (i.e., "on" to "off" and "off" to "on") of all the light-bulbs that their positions are a multiple of 3, the fourth. switches the state of all the light-bulbs that their position is a multiple of 4, and so on. The task is to output the indices (positions) of all the light-bulbs that will be in the state "on" at the end of the process.

One way of performing the task is by implementing the process described in the problem, using an N element array, and updating the array values according to the midgets' activity. This solution to the problem does not reflect any insight into the problem, and is rather inefficient, both time-wise and space-wise.

The students were directed to analyze the problem and identify a pattern that characterizes the indices of the light-bulbs that will be in state "on" at the end of the process. In a class discussion, the students reached the conclusion that a light-bulb will be in the state "on" at the end of the process, if its index is divisible by an odd number of divisors (e.g., the light-bulb in the 4-th position will be "on", since 4 is divisible by 1, 2, and 4). This conclusion provided an initial insight into the problem.

At this stage, the students were instructed to look for further characteristics, and to design a solution without using an array. As in the previous problem, here again, we received various solutions that reflected various levels of insight into the problem.

The first type of solutions were solutions of students who did not do any further analysis, and implemented the conclusion reached in class using the following nested loop:

```
for i:=1 to N do begin
   sum := 0;
   for j:=1 to i do
      if (i mod j) = 0 then sum := sum+1;
   if (sum mod 2) = 1 then write (i)
end
```

These students did not make the connection to the efficiency considerations of a previously solved problem: "checking for primality" (in which the bound of the loop was the square-root of the number checked for primality). Other students have made the connection and tried, at first, to design a nested loop in which the inner loop's bound is $[\sqrt{i}]$. While designing the inner loop, they realized that the value of the variable sum has to be incremented by 2 each time it is incremented, except for the time where $\sqrt{i}$ is an integer (when j, $j < \sqrt{i}$, divides N, both j and N/j have to be counted, and sum should be incremented by 2; when $j = \sqrt{i} = N/j$, sum should be incremented only by 1). This led many to the observation that sum remains even during the inner loop, and becomes odd only if $\sqrt{i}$ is an integer. Thus, the i's that should be printed are those whose square-roots are integers.

Following the additional insight, they designed a non-nested loop. However, some did not realize that the required i's can be efficiently generated from their square-roots. These students developed the following loop:

```
for i:=1 to N do
   if sqrt (i) = trunc (sqrt (i)) then write (i)
```

Those who did realize that the required i's can be efficiently generated from their square-roots, designed the efficient solution:

```
for i:=1 to trunc (sqrt (N)) do
   write (i*i)
```

As in the previous problem, the students calculated the number of loop iterations in each solution ($1/2\ N^2$ - for the first, N - for the second, and $\sqrt{N}$ - for the third), and further observed the efficiency differences by executing the various solutions for the value 1000 of N.

5. THE TOWER AND THE GLASS BALLS

The last problem we present is a search problem. It was introduced when students were already familiar with linear search, but before they had seen binary search.

> *The Tower and the Glass Balls.* Given a tower of N floors, one wants to find the lowest floor from which a glass ball will break once it falls. If there is only one such ball, then the search for the "breaking floor" must be linear. That is: throw the ball from the first floor, then from the second, and so on, until the ball will break. But, there are two identical balls. Thus, one ball can be used for a "coarse-grain" search until it breaks. Then, the other ball can be used for a "fine-grain" search. Develop an efficient algorithm for performing the search with minimal number of throws.

The problem was posed for discussion in groups, in an attempt to raise fruitful ideas for an efficient search principle. The ideas that were suggested can be divided into two approaches. One approach was "first, advance the first ball in a large interval, like N/2; then, a smaller interval, like N/4, and so on". The other approach was "advance the first ball in small intervals, so that if it breaks, the search interval of the second ball will be small"; e.g., "throw the first ball from the 4th floor, then from the 8th floor, and so on".

During the group discussions, representatives of the two approaches debated with one another, and students realized that there are advantages and disadvantages to each approach. In particular, students realized that the worst-case of the first approach is the case where the "breaking-floor" is the floor from which the first ball is thrown at first (N/2). And the worst-case of the second approach is the case where the "breaking-floor" is the floor from which the first ball is thrown last (N).

Following the group discussions and some guidance from the teacher, the students realized that the most efficient approach is a combination of the two approaches suggested. That is, "keep the interval between two consecutive throws of the first ball not too small, but not too large". And in a further insight: "so that the max-possible number of throws of the first ball, will be equal to the max-possible number of throws of the second ball".

Concluding from the above intuitive arguments, the first ball should be thrown in intervals of $\sqrt{N}$ floors. This guarantees that it will not be thrown more than $\sqrt{N}$ times, and once it breaks, the second ball will not be thrown more than $\sqrt{N}$ times. By this, a worse case of $2\sqrt{N}$ throws is obtained.

The students reached the quadratic-search principle of the efficient algorithm only with the teacher's assistance. However, the fruitful discussions along the way, examining the advantages and disadvantages of their two approaches, were most constructive in further realization of efficiency considerations.

6. CONCLUSION

Although introductory computer science courses present algorithm efficiency in a rather late stage, we believe that the concept of efficiency can be presented in an early stage and can accompany the studied material thereafter. In this paper we presented a novel approach of early, gradual, and intuitive introduction of algorithm efficiency, and illustrated it with three problems.

We implemented our approach with 10th and 11th grade high-school students. In the paper, we described our experience with presenting the three problems to the students. The student solutions to the problems varied considerably, reflecting different levels of insight into the problems. The various solutions led to productive discussions in class regarding efficiency considerations. Students widened their repertoire of templates, evaluated alternative designs, and realized the importance of analysis and planning, in particular in the sense that better insight yields a better solution.

Finally, the second and the third problems are colorful problems that involve physical objects. Problems that involve connecting the natural environment with more formal systems are always an asset in enriching the students' intuition [8]. In particular, when the students are pre-college students.

ACKNOWLEDGEMENT

The author acknowledges Hanna Mahlev for her assistance in following and analyzing the students' problem solving activities.

REFERENCES

[1] Arnow, D., Teaching programming to liberal arts students using loop invariants, *Proc of the 24-th Tech Symp on CS Education,* March 1993.

[2] Cooper D., *Oh! Pascal! third edition,* W.W. Norton & Company, 1993.

[3] Decker R., and Hirshfield S., *The object Concept, an Introduction to Computer Programming Using*

C++. PWS, 1995.

[4] Dijkstra, E.W., et al, A debate on teaching computer Science, *Comm ACM vol 32,* December 1989.

[5] Gries, D., *The Science of Programming,* Springer-Verlay, 1981.

[6] Linn, M.C., The cognitive consequences of programming instruction in classrooms, *Educational Researcher,* May 1985.

[7] Linn, M.C., and Clancy, M.J., The case for case studies of programming problems, *Comm ACM vol 35,* March 1992.

[8] Nesher, P., Microworlds in mathematical education: a pedagogical realism. *Knowing, Learning, and Instruction,* Resnick, L.B., editor, Lawrence Erlbaum Associates, 1989.

[9] Savitch W., *Problem Solving with C++, The Object of Programming,* Addison-Wesley, 1996.

[10] Schoenfeld, A., Learning to think mathematically: problem solving, metacognition, and sense making in mathematics, *Handbook of Research on Mathematics Teaching and Learning,* Macmillan, 1992.

[11] Spohrer, J.C. and Soloway, E., Novice mistakes: are the folk wisdoms correct?, *Comm ACM vol 29,* 1986.

[12] Tam, W., Teaching loop invariants to beginners by example, *Proc of the 23-rd Tech Symp on CS Education*, March 1992.