Informatics in Education, 2003, Vol. 2, No. 2, 211–222
© 2003 *Institute of Mathematics and Informatics, Vilnius*

Seeking or Skipping Regularities? Novice Tendencies and the Role of Invariants

David GINAT
Computer Science Group, Science Education Department
Sharet Building, Tel-Aviv University
Tel-Aviv 69978, Israel
e-mail: ginat@post.tau.ac.il

Received: July 2003

Abstract. Every repetitive process encapsulates a regularity pattern, which may be expressed as an invariant assertion. Invariants embody implicit, insightful properties that characterize the execution of programming statements. Due to their implicit nature, invariants may be less apparent to algorithmic problem solvers. Yet, invariants are essential for designing correct and efficient algorithms. This paper illustrates the essential role of invariants, and examines whether novices tend to look for invariant properties during their algorithmic problem solving. The paper presents a study in which two novel algorithmic challenges were displayed to a group of motivated, novice students. Student solutions to these challenges demonstrate an operational reasoning approach, which does not capture the essence of the problems at hand, and yields non-satisfying results. Some solutions were incorrect, others were inefficient, and some had no convincing justification. These results, and the correct and efficient solutions to both challenges illuminate the importance of assertional reasoning and the fundamental role of invariants.

Key words: invariants, algorithmic problem solving, operational reasoning, assertional reasoning.

1. Introduction

Invariants are assertions that capture the essence of serial-program loops and concurrent-program commands (e.g., Hoare, 1969; Dijkstra, 1976; Gries, 1981; Pnueli, 1981; Chandy and Misra, 1989). They express regularities that are preserved during execution. These regularities encapsulate an implicit, "behind the scenes" perspective of programs, which is different from the explicit facet embodied by program statements. While program statements are operational, "how to do" instructions, invariants are assertional, "what it means" descriptions, inferred from characteristics of the instruction composites.

As invariants encapsulate an assertional, "behind the scenes" viewpoint, one may get the impression that they are not a necessity. Although Dijkstra, Gries, and others argue that program design should go hand-in-hand with its correctness, encapsulating invariant properties, it is not clear how many CS tutors indeed present invariants to their students. Many Introduction-to-Programming and Introduction-to-Algorithms textbooks do not explicitly mention invariants (e.g., Cormen *et al.*, 1990). Some computer science (CS) tutors that are well aware of invariants might skip their presentation, as they may see invariants as relevant only for formal program verification, with Hoare's logic (1969). What about students? Are they seeking regularities during problem solving? Are they aware of the notion of invariance? Can it help them in their solutions? Or, perhaps the notion of invariance may be skipped? In this paper we address these questions.

The objective of this paper is to two-fold: to reveal novice tendencies with regularities and invariance in solving algorithmic challenges, and to illuminate the role of invariants in algorithmic problem solving. We focus on algorithmic challenges with repeated operator invocations. In a broad sense, any algorithmic problem may be viewed as a problem with a finite set of operators to utilize. In a more specific sense, a particular operator may be specified for repeated use. This is the case for example in the Game task of IOI'96 (The International Olympiad in Informatics – a Game, 1996), where the operator is "line-end removal", and in the Median Task of IOI'2000 (Horvath and Verhoeff, 2002), where the operator returned the median of three distinct values.

We display here results from a study that we conducted with 27 rather talented high-school students, who were attending the advanced stage of the practice-competition activity of our national computer science Olympiad. The students completed (in their high-school classes) an Introduction-to-Programming course and a primary Data-Structures course. In order to be better prepared for our activity, they were asked to also learn searching, sorting, and graph algorithms from a textbook (in a suggested list of books).

The goal of the activity was to observe and discuss novice and expert solution approaches to various algorithmic challenges. In this paper we display the activity part that focused on the notion of invariance. This part involved two novel challenging tasks, which were meant to relate invariance to the fundamental topics of correctness and efficiency of algorithms. One task involved an emphasis on correctness, and the other – an emphasis on efficiency.

In the next section we display the first task – Triple-Switch, and in the section that follows the second task – Permutation-Reordering. In each section we first present the task and then gradually describe the student solutions, some of which were at an expert level. Following the two task-solution sections we discuss the findings of this study in a concluding section, and tie them to didactical aspects of the presentation and utilization of invariants.

2. Triple Switch

The first task involves a line of bits. It focused on the question whether a given arbitrary line can be transformed to a homogeneous line, by repeated invocations of an operator that changes line values.

The Triple-Switch Task

The operator Triple-Switch operates on any three adjacent bits in a cyclic line of N bits. A single operation of the operator switches the value of each of the three bits. For example, the operator's operation on the first three bits of the line 101001 will result in 010001. The operation of the operator on the last bit and the first two bits of the line 101001 will result in 011000.

Develop an algorithm for which the input is a given line on N arbitrary bits, and its output is a message notifying whether the input line can be transformed, by repeated invocations of Triple-Switch, to a *homogeneous line*, where all the bits are of the same value (i.e., all the bits are 1's, or all the bits are 0's).

Four students felt at loss, and offered no solution. They had a difficulty with the many possibilities of repeated operator applications. A couple of them tried repeated application of the operator on short lines, but had no constructive idea for a general solution. The rest of the students devised algorithmic solutions. We divide their solutions into three categories: "One-Pass Conversion", "Two-Pass Conversion", and "Invariant-Based Deduction". We describe each of the categories.

One-Pass Conversion

Five students had the idea of a **Flattening scheme**, where the input line is converted, from left to right, to the value of the 1st input bit. That is, if the 2nd bit equals the 1st, then it will not be touched. Otherwise the operator will be applied on the 2nd, 3rd, and 4th bits. This will guarantee that the 2nd bit will become equal to the 1st. The 3rd bit will be handled next, similarly to the 2nd bit (possibly yielding the operator's application on the 3rd, 4th, and 5th bits). This process will continue up to the bit $N-2$. It will guarantee that the bits $1, 2, 3, \ldots, N-2$ will all have the same value. If the remaining two bits (the bits $N-1$ and N) will also have that value in the end of the process, then the output will be the message "Yes" (transformation is possible). If one of the two remaining bits will differ from the first $N-2$ bits then the output will be "No".

The students demonstrated the above algorithmic idea on two inputs. The first input was that displayed in the example of the problem definition. The algorithm advances as follows:

$$101001 \rightarrow 110101 \rightarrow 111011 \rightarrow 111100.$$

Since the two rightmost bits are not equal to the first (left) four, the output will be "No". For a similar input line, of 7 bits, they demonstrated a case where the output is "Yes":

$$1010010 \rightarrow 1101010 \rightarrow 1110110 \rightarrow 1111000 \rightarrow 1111111.$$

The students were satisfied with their solution, and checked it on very few examples. When asked to explain why the solution is correct they reiterated the Flattening scheme and demonstrated it on their particular examples. No careful verification or further insight was displayed.

When the above solution scheme outputs "Yes" it indeed is due to the ability to transform. However, is the output "No" always correct? A simple example, very similar to the latter one, shows that the output "No" may be incorrect. For the input 1010011 the above scheme will yield 1111110 and the output "No", while the output should be "Yes" (since two additional operator invocations may yield 0000000). Thus, the above one-pass solution is incorrect.

Two-Pass Conversion

Twelve students noticed the latter difficulty, and observed that an additional "reverse" pass may yield the desired transformation in some cases. They divided the possible inputs into three disjoint cases according to the line length N: 1. N mod 3 = 1, 2. N mod 3 = 2, and 3. N mod 3 = 0.

They examined a refined Flattening scheme, in which the inequality between the two rightmost bits and the left $N-2$ bits may not inhibit proper transformation. In particular, for the first two cases above, where N is not a multiple of 3, the students noticed that proper transformation can always be obtained.

In the case where N mod 3 = 1, if the Flattening ends with the rightmost bit, or its adjacent bit (but not both bits) different from the other $N-2$ bits, then the line can become homogeneous by transforming all its bits, with an additional Flattening pass, to the value of the sole different bit (recall that the line is cyclic). If both of the rightmost bits are different from the other $N-2$ bits, then a single operator application may transform the line to the latter pattern, where only one bit is different from all the others, and therefore transformation can be achieved. The case where the line length characteristic is N mod 3 = 2 is similarly analyzed.

Thus, for an N that is not a multiple of 3, the output "Yes" should be immediately displayed. For the case where N mod 3 = 0, the students kept the one-pass Flattening scheme, and set the decision of "Yes"/"No" according the equality/inequality of the two rightmost bits to the other (identical) bits.

Some of these students felt confident with this solution, after trying it with various examples; but others felt uncertain. All these students could not provide a convincing argument that explains why the Flattening scheme indeed yields the correct output for the case where N is a multiple of 3. Their typical argument was that "if the one-pass flattening scheme does not yield a homogeneous line, then you will never be able to get rid of one or two bits that are different from all the others". This "operational" argumentation, which does not capture core task characteristics, is insufficient. Quite a few students felt it themselves, and were therefore unsure about the correctness of their solution.

Invariant-Based Deduction

Only six students were able to see beyond the "operational", Flattening scheme. These students carefully analyzed a single application of the operator, and noticed that the essence lies in the operator's property of affecting three *adjacent* bits. Upon analyzing the case where N is a multiple of 3, they "colored", or numbered, the bit locations in the line as follows: 1 2 3 1 2 3 1 2 3 . . . . This coloring guarantees that every application of the operator modifies one bit in a "1"- location, one bit in a "2"-location, and one bit in a "3"-location.

The operator switches the value of a bit on which it operates. Thus, an operator invocation implies a change of the parity of the number of 0's in the "1"-locations, and similarly with the parities of the number of 0's in the "2"-locations and in the "3"-locations. This implies the following invariant:

> **Parity-Equality Invariant.** The equality/inequality between the parities of the number of 0's in the "colored" ("1", "2", and "3") locations is preserved by the operator Triple-Switch.

In examining the short example 010001 in the problem definition, we notice that the parity of the number of 0's in the "1"-locations (the 1st and the 4th bits) is even, whereas the parity of the number of 0's in the "2"-locations is odd. According to the above invariant, these two parities will never be equal. The parities of the number of 0's in the "2"-locations equals, initially, that of the "3"-locations, and will always remain equal.

The above invariant provides the clue that we need. The three parities are equal in the homogeneous, all-bits-are-equal line, for a line with length N that is a multiple of 3 (in such a line the number of "1"-location bits, the number of "2"-location bits, and the number of "3"-location bits are all $N/3$). If these three parities are unequal in the initial line, they will always remain unequal, and no homogeneous line will be reached. Conversely, if these three parities are initially equal, then the Flattening process will yield a homogeneous line.

Concluding from the above, the initial line 010001 cannot be transformed to a homogeneous line, whereas the initial lines 110001 and 001001 can be transformed to a homogeneous line.

All in all, a very concise solution was obtained. We display it in the concluding scheme below. For computation convenience, the scheme involves comparisons of the parities of the number of 1 bits rather than the parities of the number of 0 bits.

> **Concluding Scheme:** If N, the line length, is not a multiple of 3, or it is a multiple of 3 and: the parity of the sum of the bits in the locations $\{1, 4, 7, \ldots\}$ equals the parities of the sums of the bits in the locations $\{2, 5, 8, \ldots\}$ and $\{3, 6, 9, \ldots\}$, then output "Yes"; otherwise output "No".

Notice that in the above scheme there is no invocation of the Triple-Switch operator. The core characteristic was recognized and specified in the Parity-Equality Invariant, using auxiliary coloring. The previously developed Flattening scheme, together with the invariant yielded the concluding scheme. Thus, this scheme was derived from both an operational component and an assertional component. Only the students who were able to combine both the operational and the assertional elements reached the concise and elegant Concluding Scheme and felt confident with their solution. We further elaborate on this aspect in the Discussion section.

3. Permutation Reordering

The task presented in the previous section involved emphasis on correctness. The task in this section involves emphasis on efficiency. The task focuses on the question of whether a given permutation can be transformed to another one, by repeated invocations of an operator that reorders permutation elements.

The Permutation-Reordering Task

The operator Move-a-Pair moves a pair of two adjacent elements in a permutation to another place in the permutation. Given two different permutations P_1 and P_2 of the integers 1 ... N, the goal is to transform P_1 to P_2 by repeatedly utilizing Move-a-Pair.

Develop an efficient algorithm for which the input is N, P_1 and P_2, and its output is a message notifying whether the desired goal can be achieved.

For example, for $N = 4$ and the permutations P_1: 1 3 2 4 and P_2: 2 4 1 3 the output will be "Yes" (since moving the pair 2 4 of P_1 to the first two positions will yield the transformation.) Had P_2 been: 1 2 3 4 the output would be "No".

Three students had no clue of how to approach the problem. The other students developed algorithmic solutions that differed in their insight and efficiency levels. Unsurprisingly, here too the students who gained limited insight offered a one-pass conversion scheme. We divide their solutions into three categories: "One-Pass Conversion", "General Counting", and "Concise Computation". We describe each of the categories below.

One-Pass Conversion

Fourteen students devised a **Flattening scheme**, similar to the one devised for the previous task. In this scheme, the elements of P_1 will be put in their destinations from left to right, according to their order in P_2. The element that has to be leftmost will be moved together with its right neighbor to the two leftmost locations in P_1, and the sub-line over which this pair should be "skipped" will be shifted to the right by 2 positions. Then, the element that needs to be second from left will be moved (with its right neighbor) to its final destination, etc. (Notice that if an element is the rightmost one in P_1, then two moves are required to put it in its destination.)

Eventually, the first $N - 2$ elements of P_1 will be properly ordered, but the two rightmost elements may not be necessarily ordered. For example, for P_1: 1 3 2 4 and P_2: 2 1 3 4, P_1 will be ordered as follows: 1 3 2 4 $\rightarrow$ 2 4 1 3 $\rightarrow$ 2 1 3 4. Had P_2 been: 2 1 4 3, the rightmost pair would be improperly ordered. The output will be dictated from the rightmost pair. If at the end of the Flattening process the rightmost pair will be ordered as it is in P_2, then the output will be "Yes", otherwise the output will be "No".

As in the previous task, here too, quite a few students were satisfied with their solution, and were convinced by its correctness, based on their experience with various examples. However, some felt that they have no convincing argument for proving that this scheme always yields the right output. When requested to justify their solution they claimed that "if in the end of the one-pass flattening process, the rightmost pair is improperly ordered, then there is no way to get rid of one unordered pair". This "operational" argumentation does not capture core task characteristics, and is insufficient. The complexity of this solution is O(N^2), as each move of a pair implies a shift of a sub-line with length in the order of N.

General Counting

Seven students went beyond the **Flattening scheme**. As the operator Move-a-Pair modifies relative ordering of elements in a permutation, they looked for “ordering characteristics”. One such characteristic may be tied to the number of ordered/unordered pairs of (not necessarily adjacent) elements. We regard any two elements in a permutation *ordered* if the smaller element is to the left of the larger one. One may measure the *number of unordered pairs in a permutation*. For examples, this number in the permutation 2 4 1 3 is 3 (since the pairs 1 2, 1 4, 3 4, are not in order).

The students noticed that the two elements moved by Move-a-Pair change their order with respect to each of the elements in the sub-line that is shifted due to the move. They identified the following two interesting patterns:

Change in the Num-of-Unorders: Let XY be the two elements moved by Move-a-Pair and Z an element in the sub-line shifted due to the move. The change in the number of unordered pairs due to moving XY “to the other side” of Z is $+2$, 0, or -2.

Num-of-Unorders Invariant: The total change in the number of unordered pairs resulting from an invocation of Move-a-Pair is an even number. Therefore, the parity of the number of unordered pairs is preserved by the operator.

The above invariant implies that a transformation is impossible if the parities of the “Num-of-Unorders” in P_1 and P_2 are different. If they are equal, however, then the Flattening scheme described earlier will result in a proper ordering of the rightmost pair, and therefore transformation is possible. This can be summarized in the following conclusion:

Ordering Pattern: P_1 can be transformed to P_2 if-and-only-if the parities of their number of unordered pairs are equal.

The invariant illuminated core characteristics that justified the “Yes”/”No” output of the operational, Flattening scheme. However, the invariant implies that there is no need to activate the Flattening scheme. Instead, one may compare the parities of the number of unordered pairs in P_1 and P_2. This is particularly relevant if the parities can be computed in less than O(N^2) time.

Simple counting of the number of unordered pairs in each permutation may be performed in O(N^2) time, by comparing the relative ordering of each pair of elements. Can one do better, time-wise? Some of the students turned to sorting algorithms. One particular algorithm that seemed relevant for them was Merge-Sort (Cormen *et al.*, 1990).

In Merge-Sort, it is possible to tell the exact number of reorders that occurs in every merging step. For example, in merging the *left* sub-vector 3 7 9 and the *right* sub-vector 4 8 10 to an ordered vector: first, 3 is chosen; then 4 is chosen and “skipped over” the 7 and the 9; then, 7 is chosen; then 8 is chosen and “skipped over” the 9; and finally the 9 and the 10 are chosen. This merging phase resulted in 3 reorders. And indeed, the number of unordered pairs in the pre-merged vector 3 7 9 4 8 10 is exactly 3 (due to the pairs 4 7, 4 9, 8 9).

Since the time complexity of Merge-Sort is O($N \log N$), the computation of the number of unordered pairs in each permutation can be done in time O($N \log N$), and this will be the complexity of our task solution – a significant improvement over the previous solutions.

The students who offered this solution were very proud of their improvement. Their solution does not require the operator's utilization, it is derived from an insightful invariant, and it is based on an efficient sorting scheme. However, their solution does not take advantage of the task characteristic that the two permutations involve the particular elements 1 ... N. Their counting of reorders is general, and can be applied for any N different elements. In addition, the Ordering Pattern specified above does not require the numbers of unordered pairs. Only parities are required. Perhaps one can do better by seeking only parities, and capitalizing on the characteristic that only the values 1 ... N are involved? Very few students addressed these observations.

Concise Computation

Three students noticed that when the N permuted values specifically involve the values 1 ... N, the ordering of any permutation in increasing order can be performed by *directly* putting each element in its destination in the permutation 1 2 3 ... N, as in this permutation each value i is in location i.

Thus, one may order a permutation by a **Direct-Placement scheme**, swapping the leftmost element i not yet in location i (its destination) with the element currently in location i. Each swap directly places an element in its destination in the ordered permutation 1 2 3 ... N. This scheme requires no more than N swaps, in one scan of the permutation P_1. For example, 2 4 1 3 will be ordered as follows:

2 4 1 3 $\rightarrow$ 4 2 1 3 $\rightarrow$ 3 2 1 4 $\rightarrow$ 1 2 3 4.

If the parity of the number of unordered pairs of a permutation can be derived from such ordering, one may order each of the permutations P_1 and P_2, and compare the parities. Examining a single swap in the Direct-Placement scheme yields valuable insight.

Let a permutation be ... X ... Y ... Z ... and the two swapped elements – X and Z. The change in the total number of unordered pairs due to reordering X Y and Y Z is: $+2$, 0, or -2. This is true for any Y between X and Z. To this change we have to add the reordering of the pair X Z. This reordering adds $+1$ or -1. Combining these observations implies a core invariant:

> **Direct-Placement Invariant:** Each swap in the Direct-Placement scheme swaps the *parity* (0 to 1, and 1 to 0) of the number of unordered pairs in a permutation.

The above invariant, together with the previously stated Ordering Pattern yields the following Concluding Scheme:

> **Concluding Scheme:** The *parity* of the number of unordered pairs of a permutation equals the *parity* of the number of reordering swaps in the Direct-Placement scheme. Therefore, if after ordering each of the permutations P_1 and P_2 with the Direct-Placement scheme, the parities of the number of reordering swaps are equal, then output "Yes"; otherwise output "No".

For the permutations P_1: 1 3 2 4 and P_2: 2 4 1 3 in the task description, the parities of the number of swaps are equal (1 swap for ordering $P_1, 3$ swaps for ordering P_2). Therefore, transformation using Move-a-Pair is possible. Had P_2 been 1 2 3 4, the parities would be different and the transformation impossible.

All in all, the Concluding Scheme requires O(N) time, an improvement over the previous schemes. It was derived from invoking both operational and assertional elements – the Flattening scheme, the Num-of-Unorders Invariant, the Direct-Placement scheme, and the Direct-Placement invariant.

4. Discussion

The student solutions to the two tasks display a tendency, by many, to solely follow an operational approach. In both tasks, a significant number of students devised an algorithmic scheme that they could not clearly justify. In the Triple-Switch task, a total of 17 out of the 27 students devised a Flattening scheme, for which they were unable to argue correctness. The five One-Pass Conversion students developed an erroneous solution. The twelve Two-Pass Conversion students designed the right solution, but quite a few of them were unsure about its correctness. In the Permutation-Reordering task, 14 out of the 27 students (all the One-Pass Conversion students) devised the rather inefficient Flattening scheme, for which they could also not argue correctness.

All these students tended to go directly for an algorithmic scheme, and their justification of that scheme was based on examination of various examples. Some, who examined too few examples (the One-Pass Conversion students in the Triple-Switch task), obtained an erroneous solution. Others, who examined further examples, were more careful, yielded better results, but felt that their insight is insufficient. In both tasks, less than half of the students looked for regularities, or invariant properties, which yield insight and illuminate core characteristics.

In analyzing the student solutions in line with Schoenfeld's problem solving framework (1985, 1992) of resources, heuristics, control, and beliefs, it seems that the students lacked resources and demonstrated limited monitoring and control ability. The notion of invariance is an essential resource, encapsulating an assertional point of view, invoked by experts during the design and analysis of algorithms. Experts' monitoring and control of their own problem solving processes involves assertional arguments that embody task and solution patterns. The students in this study seemed to lack awareness of assertional arguments in general and the notion of invariance in particular. They neither sought regularities for identifying task characteristics, nor did they utilize invariants for arguing and validating their solutions' correctness.

In the computer science community, the widespread manner of checking program validity is based on testing a program on diverse test cases, carefully chosen for examining a variety of "operational scenarios". Yet, this may sometimes not be enough for gaining satisfying assurance and sufficient confidence of correctness. The elegant and efficient solutions to both tasks in this study illustrated the importance of seeking and utilizing invariants. Only upon identifying invariant properties could one properly argue correctness and devise a concise algorithm.

It may not be surprising that novices tend to solely go for operational reasoning, as this is what they often see in textbooks. Many students seem to practice algorithmic problem solving with inclination to a trial-and-error approach, not only in the beginning of a design, but also upon verifying their final algorithm. They learn language constructs and algorithm design techniques, and often “get by” without regularities and invariants. However, one may not always “get by” without invariants, as could be seen in this study. The lack of turning to regularities and invariants may yield undesirable outcomes.

The notions of regularities and invariants add an important perspective to algorithm design, as could be seen in the two tasks presented here. Regularities and invariants may be essential for considerations of both correctness and efficiency. Although they require mathematical insight, which is not always straightforward, their illuminating role is fundamental.

The primary role of invariants was clearly apparent in this study. Yet, we do not argue for exclusive assertional reasoning. Invariants captured the essence in each task, but one still had to turn to operational reasoning and devise algorithmic solution schemes. The operational, Flattening idea in both tasks, the creative utilization of Merge-Sort, and the insightful Direct-Placement scheme in the Permutation-Reordering task were essential design patterns (Astrachan *et al.*, 1998) for the final solutions. The concluding schemes in both tasks combined both assertional and operational elements.

Operational reasoning is inherently present in the teaching of any course that involves programming and algorithms. Assertional reasoning is sometimes present only in a course of formal verification methods. This may result in the wrong impression that invariants are relevant only as unintuitive, formal means. We believe that CS tutors should combine both ways of reasoning in a natural, rather informal way in their teaching, as was illustrated in this paper. The illustrations displayed in this paper naturally combined both ways of reasoning and showed their mutual roles. Invariants were not displayed using formal logic, yet they were precisely stated. They concisely captured the core characteristics of each task, complementing the function of the operational, algorithmic schemes.

5. Conclusion

We displayed a study of student solutions to two algorithmic challenges, for which an assertional reasoning approach, involving regularities and invariants, was essential for reaching concise and efficient solutions. More than half of the students did not turn to regularities and invariants, but rather solely followed an operational reasoning approach, and yielded incorrect and inefficient results. It seemed that many were unaware of the important role of regularities and invariants, possibly due to the very limited number of demonstrations of invariants that they saw throughout their studies. The two challenges and the student solutions illustrate the essential role of invariants on the one hand and the unawareness of this role on the other hand. In concluding this study, we offer several pedagogical considerations for CS educators.

- Increase throughout the CS curriculum the number of programming and algorithmic challenges for which the recognition of regularities and invariants is a primary and convincing solution element.
- Let the students follow their own solution approaches and realize the limited outcomes and the need to go beyond operational reasoning.
- Elaborate on the fundamental and illuminating role of assertional reasoning and invariants, for capturing the essence, yielding a concise design, and arguing the design correctness.
- Show the importance of combining operational, design patterns and assertional, invariant patterns in successful algorithm design and analysis. Emphasize these elements' mutual roles.
- Underline problem solving heuristics that are utilized during the combined design. This may include generalization from simple cases, analogy invocations of familiar algorithmic schemes (e.g., Merge-Sort), and utilization of auxiliary elements (e.g., 1-2-3 "coloring").

Future work may offer further studies in line with the above pedagogical considerations. We believe that a convincing display of the complementing, yet mutual functions of both operational and assertional reasoning is a key element in broadening novices' perspectives. The elaboration of multiple perspectives in algorithmics, which evolves from this study, is a primary means in enhancing student competence in the novice-to-expert ladder.

References

Astrachan, O., G. Berry, L. Cox and G. Mitchener (1998). Design patterns: an essential component of CS curricula. In *Proceedings of the 28-th SIGCSE Technical Symposium on CS Education*, pp. 153–160.

Chandy, M.K., and J. Misra (1982). *Parallel Program Design – A Foundation*. Addison-Wesley.

Cormen, T.H., C.E. Leiserson and R.L. Rivest (1990). *Introduction to Algorithms*. MIT Press, Massachusetts.

Dijkstra, E.W. (1976). *A Discipline of Programming*. Prentice-Hall.

Game Task (1996). *The International Olympiad in Informatics*. `http://olympiads.win.tue.nl/ioi/ioi96/contest/ioi96g.html`

Gries, D. (1981). *The Science of Programming*. Springer-Verlag.

Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, **12**, 576–580.

Horvath, G., and T. Verhoeff (2002). Finding the median under IOI conditions. *Informatics in Education*, **1**.

Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science*, **13**, 45–60.

Schoenfeld, A. (1985). *Mathematical Problem Solving*. Academic Press.

Schoenfeld, A. (1992). Learning to think mathematically: problem solving, metacognition, and sense making in mathematics. In D.A. Grouws (Ed.), *Handbook of Research on Mathematics Teaching and Learning*. Macmillan, pp. 334–370.

D. Ginat is a faculty member in the Science Education Department at Tel-Aviv University. He obtained his PhD (Distributed Algorithms and Amortization Analysis, 1989) from the University of Maryland, at College Park, U.S.A. His research and teaching during the last 10 years focus on the development of algorithmic problem-solving skills, at all levels, from the very basic to an olympiad level.

Siekti griežtumo ar nekreipti į tai dėmesio? Invariantų vaidmuo mokant programuoti

David GINAT

Kiekvienas pasikartojantis procesas remiasi dėsningumu, kuris programavime paprastai apibūdinamas invarianto sąvoka. Invariantai įgyvendina implikatyvias (neišreikštines) savybes, kuriomis nusakomas programavimo taisyklių vykdymas. Dėl užslėptos invariantų prigimties jų taikymas sprendžiant algoritminius uždavinius nėra akivaizdus. Vis dėlto invariantai turi ypatingą reikšmę teisingų ir efektyvių algoritmų sudarymui, todėl jiems turėtų būti skiriamas ypatingas dėmesys, ypač ugdant jaunuosius programuotojus.

Šiame straipsnyje išryškinamas esminis invariantų vaidmuo. Čia aptariama, ar pradedantieji programuotojai yra linkę įžvelgti invariantų savybes spręsdami algoritminius uždavinius, ar apie tai negalvojama. Darbe supažindinama su tyrimu, kurio metu žingeidžių pradedančiųjų programuoti studentų grupė gavo dvi neįprastas algoritmines užduotis. Analizuojant studentų sprendimus buvo nustatyta, kad studentai labiau remiasi operaciniu mąstymu, kuris šiuo atveju, gilinantis į uždavinį, nėra pakankamas; taigi pasiekti patenkinamų rezultatų jiems nepavyko. Vieni sprendimai buvo neteisingi, kiti – neefektyvūs, o dar kiti net nebuvo motyvuotai pagrįsti. Šie rezultatai, kaip ir teisingi bei efektyvūs abiejų užduočių sprendimai, atskleidžia argumentuoto, teiginiais pagrįsto mąstymo svarbą bei parodo, koks svarbus vaidmuo tenka invariantams.