Informatics in Education, 2021, Vol. 20, No. 4, 567–582
© 2021 *Vilnius University, ETH Zürich*
DOI: 10.15388/infedu.2021.25

Abstraction, Declarative Observations and Algorithmic Problem Solving

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

Received: February 2021

Abstract. The notion of algorithm may be perceived in different levels of abstraction. In the lower levels it is an operational set of instructions. In higher levels it may be viewed as an object with properties, solving a problem with characteristics. Novices mostly relate to the lower levels. Yet, higher levels are very relevant for them as well. We unfold the importance of higher level abstractions for novices, by demonstrating the role of declarative observations of algorithmic problems, and the benefit of developing awareness of such observations in algorithmic problem solving. This is shown in a two-stage study, which first reveals the unfortunate lack of declarative observations, and then displays comparative results of experimental and control groups, which stems from different awareness and competence with declarative observations.

Keywords: abstraction, algorithmic problem solving.

1. Introduction

Consider the following algorithmic problem: "Given a long list of N different integers, output the *largest drop* in the list, which is the maximal difference between two (not necessarily adjacent) integers such that the larger is to the left of the smaller." For example, the largest drop in the list 3 0 4 7 2 1 5 is 6, due to the difference between the 7 and the 1. A simple problem for programming beginners? Not quite.

How do computer science (CS) novices approach this problem? In our experience, many novices offer erroneous, or inefficient cumbersome solutions to this problem. Some decide that the left-end of the largest drop must be the max list value (a hasty erroneous conjecture); others seek for each list integer its largest drop (an inefficient computation); and some invoke unnecessary subroutines for seeking "local maxima" and "local minima". Observant novices see that: *The left-end of the largest drop is larger than all the values to its left, and the right-end is smaller than all the value to its* *right*. This observation yields a simple on-the-fly solution (which repeatedly updates the max so far, and examines its differences from the next input values).

The latter observation expresses abstraction. It "captures" core entities on which to focus. And, it is ***declarative***, in the sense that it expresses a declarative characteristic of the posed problem which is not tied to a particular computation scheme. One may capitalize on this characteristic in devising a suitable operational solution. The incorrect and inefficient solutions express various implementations of "direct translation" (Hegarty *et al.*, 19995) of the problem text into naïve operational schemes, without "lifting" the point of view and extracting the essence.

The different problem solutions exemplify the gap between more and less abstract perspectives of an algorithmic problem. The declarative observation encapsulates a perception related to the ***problem-level*** of Perrenet *et al.*'s (2005) model of abstraction of the *concept of algorithm* (see next section). The other solutions express a perception of the ***program-level*** of the model, which focuses on the implementation of an operational computation. Knuth (in Hartmanis, 1994), Wing (2006) and others advocate thinking at both high and low abstraction levels. Yet, this is not necessarily what happens with novices, as exemplified above. In this paper we illuminate the lack of appropriate abstraction, and examine the outcome of an intervention aimed at addressing this deficiency.

In our experience, declarative observations are not the natural entities sought by algorithmic problem solvers. Algorithms (or computer programs) are operational – they are composed of schemes of "to do" statements. It may just be natural to go for the "how" of the computation rather than the underlying "that" of the problem. (A wider "how/that" perspective is in Ryle's (1946) "knowing how and knowing that".) But then, abstraction is avoided, and the underlying problem perspective may be missing. In order to address this difficulty, awareness of the importance of the higher problem level should be elaborated. This level encompasses the facet of declarative observations (among additional things). The skill of reaching such observations is cognitive. The awareness of their importance in one's problem solving process is metacognitive.

A recent study by Seth *et al.* (2016) argues for awareness of the fundamental role of the process of problem solving. Additional studies advocate the incorporation of metacognition in learning (e.g., Mani and Mazumder (2013)) and the awareness of one's location in her problem solving process (e.g., Loska *et al.*, 2016; Prather *et al.*, 2019). These studies address the various stages of the problem solving process (e.g., Bloom's (1956) stages, and Polya's (1945) phases). We underline a specific element of the problem solving process – the element of seeking the "that" of the posed problem before advancing to the "how" of its solution.

In what follows, we display a two-stage study of: 1. Illumination of the lack of a "that" phase among novices; and 2. Elaboration on its awareness. In the first stage, a first problem solving questionnaire was posed to an experimental group for examining (possibly missing) capitalizations on declarative observations. In the second stage, an intervention underlined and exemplified the role of declarative characteristics, and an additional questionnaire was posed to the experimental group. This questionnaire was also posed to a control group who studied the same materials but without underlining declarative characteristics.

The next section presents the background on abstraction for this study. Section 3 displays the methodology of the two stages. Section 4 displays the results of the first stage. Section 5 displays the results of the second stage. Section 6 shows statistical analyses of within subjects (of the experimental group) and between subjects (of the experimental and control groups). The final section discusses the findings, and advocates the essential role of declarative observations in developing algorithmic thinking already in early CS studies.

2. Background

Abstraction is fundamental in mathematics and computer science. It is underlined and advocated by primary computer scientists from the very beginning. Dijkstra (1972) regards abstraction as "one of the most vital activities of a competent programmer", in arguing the significance of generalizing a variety of recognized cases. Aho and Ulman (1972) regard computer science as the "science of abstraction", in which one of the core elements is modelling and representation. Knuth argues that natural computer scientists can move between levels of abstraction, seeing things 'in the large' and 'in the small' (in Hartmanis, 1994). Task forces for designing CS curricula repeatedly indicate abstraction as a fundamental CS notion (e.g., Denning *et al.*, 1988).

Fundamental design elements, including top down design, abstract data types (e.g., Dale and Walker, 1996), and design patterns (e.g., Gamma *et al.*, 1994) are inherently based on abstraction. They involve modelling, modularization, and zooming in and out upon examining and representing computational building blocks. Wing (2006) underlines these aspects of abstraction together with the characteristic of thinking at multiple levels of abstraction. Statter and Armoni (2016) offer the perspective of "changing the resolution", in the sense of finding the right degree of resolution for a problem solver. They also offer the point of view of "ignoring the how and focusing on the what", in the sense of using black boxes for which one does not need to know how an inner computation is conducted.

Statter and Armoni examined the teaching of abstract thinking to novices by developing verbal descriptions and quality documentation before writing programs. The basis for their analysis was Perrenet, Groot, and Kaasebrood's (PGK) four levels of abstraction of the concept of algorithm (2005):

1. **The execution level**, where an algorithm is interpreted as a specific run on specific input, in a specific concrete machine.
2. **The program level**, where an algorithm is an operational ("how") entity, described by a code of a specific programming language.
3. **The object level**, where an algorithm is viewed as an object, not associated with a particular language. The algorithm's properties such as invariance characteristics, termination, and complexity measures are examined.

4. **The problem level**, where an algorithm is viewed as a black box, solving a problem. The focus is on the problem features, including its solvability and intrinsic complexity, such as the lower bound for any of its algorithmic solutions.

We illustrate these levels in brief with the following algorithmic task: "Given a list of N weights of students, determine whether they form a homogenous group; where a *homogeneous group* is one in which the weight difference between every two students does not exceed 5 kg." **Level-1** of the hierarchy involves a solution for a particular input example. **Level-2** includes an algorithm for the general case. It may be efficient or inefficient. An inefficient solution may compare the weights of every two students. **Level-3** may relate to the complexity measures of the level-2 solutions (viewed as objects); e.g., $O(N^2)$ in the case of the inefficient solution. **Level-4** may involve, for example, the lowest time bound for any solution of the problem. We argue that this level may also involve an insightful declarative observation of the problem, such as: "A group is homogenous if and only if the difference between its max and its min does not exceed 5". (This observation yields a simple, elegant, and efficient O(N) solution.)

Quiet a few of Statter and Armoni's students demonstrated the program level (level-2) in the activities. Previous, as well as later studies repeatedly show that significant number of novices remain in the program level (e.g., Haberman, 2004; Haberman *et al.*, 2005; Ginat and Blau, 2017). In addition, studies of both introductory courses and more senior courses show that students refrain from using black boxes, and refrain from referring to them as "atomic" objects, even though the utilization of black boxes may considerably simplify the computation (e.g., Haberman and Ben-David Kolikant, 2001; Armoni *et al.*, 2006).

Ginat and Blau (2017) examined their findings through slightly different lenses, primarily related to the object of thought and the process-object duality. They adopted the first two of the following three interpretations of abstraction levels, displayed by Hazzan (2002), who borrowed it from the domain of mathematics education:

I. Abstraction level as the **quality of the relationships** between the object of thought and the thinking person.
II. Abstraction level as a reflection of the **process-object duality** (of first perceiving a term, or a concept by its computation process, and later (at a higher level) by its encapsulation into an object).
III. Abstraction level as the **degree of complexity** of the concept of thought.

The first interpretation relates, in algorithmics, to familiarity with design patterns (e.g., summation, max computation, searching schemes), as well as awareness of correctness, efficiency, and modularity. The second interpretation relates to the duality between the operational level and the conceptual, declarative level. The former is necessary for a solution implementation, and tied to level-2 of the PGK hierarchy, expressing the "how" of the computation. The latter is tied to level-3 of the hierarchy, viewing an algorithm as an object. It is also tied to level-4, through ("that") problem characteristics.

In what follows, we examine and analyze novices' algorithmic solutions and perceptions in light of both the PGK hierarchy and the interpretations I and II above, of abstraction levels.

3. Methodology

The study included two stages, and involved an experimental group of 29 participants and a control group of 31 participants. The participants of the experimental group solved an initial questionnaire of 3 algorithmic tasks, and then solved a second questionnaire of 3 tasks after intervention, in which declarative, "that" observations were underlined. The participants of the control group solved only the second questionnaire, after ordinary teaching of the same amount of time, in which the emphasis was on operative, "how" code-designs. We examined statistically both within-group and between-groups findings, and interviewed some of the participants.

Both groups of participants participated in an elective **algorithmic problem solving course** in our university. The groups studied the course in consecutive years, first the experimental group and then the control group. The course involved algorithmic tasks and some related mathematical tasks; and focused on elements of algorithmic and mathematical thinking. Apart from algorithmic aspects, the course involved cognitive and metacognitive notions, including abstraction, the use of heuristics, monitoring of one's problem solving process, awareness of underlying beliefs, and more. The course objective was to practice and comprehend problem solving, with participants who are already acquainted with the basics of CS and Math.

The students in the course had completed at least three semesters of CS/Math studies, including CS1 and CS2. The algorithmic tasks in the course involved sequence processing, mathematical games, and additional combinatorial computations. Since our focus was on problem solving, we only expected familiarity with 1D/2D arrays, elementary design patterns such as counting, searching and sorting, and the notions of correctness, efficiency, and modularity. Efficiency was discussed with respect to both time and space, with "soft" relation to the complexity measures of $O(N)$ and $O(N^2)$. The discrete mathematics needed was basic. All the **tasks** in the course were **shortly stated** (several lines) but usually **non-immediate**, in terms of the process of obtaining a sound solution. They involved hidden patterns, which were short to phrase, but sometimes challenging to recognize.

In the beginning of the course in both years of the study, the students were asked to solve a 75 min questionnaire, with 3 rather elementary algorithmic/programming tasks, in order to evaluate their initial knowledge and skills. The tasks involved devising algorithms for: counting the number of appearances of the second-max in a list, a variation of binary search, and a variation of bubble-sort. The average grade of the experimental group was 72 (out of 100), and that of the control group was 75.

In the second week of the first year, the questionnaire of stage-1 of the study was posed to the **experimental group**. In the following two classes, we solved the questionnaire with the experimental group students, while underlying the notion of declarative observations. We elaborated on the "that" perspective, and demonstrated and practiced this perspective in both class and homework assignments. Two weeks later we posed the stage-2 questionnaire.

The two-week **intervention** described above involved a few short guidelines.

- After understanding a posed problem, examine diverse I/O cases, and try to extract characteristics of the problem. The characteristics should be of the problem, and not of solutions.
- Observations should be phrased in a declarative manner, not in an operational one. Yet, they should be relevant for operational considerations and their justifications.

The intervention involved a set of worked out examples, that were carefully chosen, based on the approach of learning from examples (e.g., Zhu and Simon, 1987; Chi and Basok, 1989). They were of different levels of challenge. In addition, different examples involved different features, so that the examples offered width in terms of diverse problem characteristics. The questionnaire tasks of stage-1 were among the examples. These tasks involved sequence processing that required capitalization on patterns. One involved a reverse perspective; another involved "tag" recognition; and a third, more challenging, involved value collection with corresponding "memorization".

Accurate phrasing of observations was challenging to many students. In addition, sometimes observations were not very meaningful. Nevertheless, experience in seeking, recognizing, and phrasing was gained. And awareness of the relevant role of the "that" perspective was developed.

In the following year, we taught three classes to **the control group**, with the same task materials. The notion of declarative observations was not mentioned. Rather, class discussions about algorithmic solutions were conducted in a way that we regard as common, of examining mostly operational solution ideas offered by the students. The class discussions mostly involved an operational "how" language. These students were also given corresponding homework assignments. After three weeks into the course we posed to these control group students the stage-2 questionnaire.

Both groups of students cooperated with the described activities, and really enjoyed them. The challenges in the posed tasks rose interest and involvement. Students in both groups were enthusiastic, discussed the solutions among themselves, and appreciated elegant, concise, and simple observations.

The participants' solutions in the two stages were collected and analyzed, both quantitatively and qualitatively.

For stage-1, we summarized for each of the 3 tasks the number of the solutions that capitalized on "that" patterns (of declarative observations) and those that did not. We also extracted features of less desired "how" solutions that did not capitalize on "that" patterns.

For stage-2, we conducted two **comparative statistical calculations**, using the Generalized Estimating Equation (GEE) procedure (Hardin and Hilbe, 2013), to assess rates of success of the intervention, of within-group and between-groups. The former involved pre-intervention and post-intervention data of the experimental group, and the latter involved post-intervention data of the experimental group and data of the control group. The data in each case was binary, in the sense of capitalization, or no-capitalization on "that" patterns in the participants' solutions.

4. Findings of Stage-1

This section displays our findings and analysis of the experimental group solutions to the pre-intervention questionnaire, of 3 algorithmic problems (tasks). We mentioned features of these tasks in the previous section. Each task is displayed with its solutions and some students' sayings in interviews following their solutions. Then, we summarize the statistics of the student responses to the tasks, with respect to their demonstration of abstraction through capitalization on declarative observations.

The 29 students of this group were given 90 minutes for solving the tasks. They were allowed to provide their solutions either in pseudo-code, or in a programming language of their convenience. They were requested to pay attention to correctness and efficiency, and briefly write the ideas underlying their solutions. Four students, who demonstrated different levels of competence were interviewed about their solutions.

4.1. *a-b Substrings*

The first task was the following.

T1. a-b substrings. Given a string of N characters from the **abc**, output the number of substrings that start with an **a** and end with a **b**. Counted substrings may overlap.

Example: In the string **cdbaagbabbgbab** the number of **a-b** substrings is 15. (Notice that each of the first (left) two **a**'s appears in 5 **a-b** substrings.)

There were no erroneous solutions. Yet, the amount of inefficient solutions was large. More than half (18) of the solutions counted the number of substrings in a brute-force manner, by going back-and-forth over the given string, counting separately for each **a** the number of **b**'s that appear to its right. Those who offered this solution conducted a "direct translation" of the task specification into a naïve operational solution, and expressed level-2 of the PGK hierarchy. In a following interview, one of the students – P15 (participant #15) – was very explicit:

> "I do not see a different way to solve the task, as I need to count for each **a** the number of **b**'s that follow it. This is very clear from the task specification."

The remaining 11 students capitalized on the following:

*Each **b** "closes" a number of **a-b** substrings which is equal to the number of **a**'s that precede it.*

This declarative observation expresses a reversed point of view, different from the natural "forward" one. The declarative perspective relates to the 4-th PGK level, as it specifies a task characteristic. An on-the-fly solution of counting **a**'s and increasing the counter of **a-b** substrings each time a **b** is met offers a considerable improvement of the naïve solution offered by P15, both time-wise and space-wise (no need to keep the input in memory).

4.2. *Increasing Sub-sequences*

The second task involved counting of a different nature.

T2. Increasing sub-sequences. Given a long list of N different integers representing heights of a ridge, output the number of slopes that ascend from left to right (a slope involves 2 or more integers, starts from a lowest, bottom point and ends in a top).

Example: In the list 9 1 3 5 2 0 8 6 there are two such slopes – 1 3 5 and 0 8. (We do not regard the sub-lists 1 3 and 3 5 as relevant slopes, since the first does not reach the top, and the second does start from a lowest point.)

We were somewhat surprised to see that almost half (14) of the students "went over the list and collected slopes", mostly by invoking a sub-routine that follows each slope from bottom to top. Their algorithmic solutions were cumbersome and error-prone, due to inaccurate recognitions of bottoms and tops. In a following interview, one of these students (P23) said the following:

> "When I start processing a sub-sequence, I follow it until it ends; then, I prepare for the next sub-sequence. It is convenient to use a subroutine to collect sub-sequences."

Another student (P16) indicated some challenges she faced in her solution process:

> "I had some difficulties in handling very short sub-sequences, of length 2, as they end right after they begin. In addition, I was not sure how to proceed in descending slopes, again, especially if they are very short."

These students talked in operational "how" terms. They focused on the "natural" entities in the task specification – complete slopes, and did not realize that it is inappropriate here to focus on complete entities. Student P23 used a sub-routine, as kind of a black box, that in a sense expressed "pseudo abstraction". Although his solution was modular, it was not insightful. The sub-routine only complicated matters. These students' operational solutions relate to level-2 of the PGK hierarchy.

Some (3) students provided solutions that were difficult to understand. The rest (12) of the students were more observant and capitalized on the observation that:

The number of ascending slopes equals the number of beginnings of such slopes. And a beginning has a V-shape form of three integers – larger-smaller-larger.

These observations are declarative, describing "that" characteristics of the task. As such, they relate to the 4-th level of the PGK hierarchy. A simple, elegant, on-the-fly solution derives from these observations, based on seeking V shapes. The term (and drawing) "V shape" was used by P5 – one of the interviewed students. Four of the 12 students wrote descriptions that explicitly mention their declarative observations.

4.3. *Permutation Collection*

The third task was harder than the first two.

T3. Permutation collection. Given a permutation of 1..N, whose integers should be collected one-by-one, according to their values, output the number of times in which there will be a need to move to the left during the "collection process", which will start in the left end.

Example: For the permutation 4 5 1 3 6 2 the output should be 2, as first 1 and 2 will be collected, then the collection process will return left to collect the 3, and then return left to collect the 4, from which it will proceed right, and collect the 5 and 6.

We received many inefficient, or erroneous solutions. About a third (11) of the students just simulated the collection process, and counted the number of moves to the left. About the same amount of students (12) tried to extract characteristics from the "shape" of the given permutation, but got it wrong. Many conjectured that the number of unordered adjacencies yields the output. For example, in the task statement, there are 2 unordered adjacencies – 5 1 and 6 2. But, if we change the order of 1 and 3, we add an unordered adjacency without changing the necessary output. Some noticed the latter, and offered erroneous patches. In a sense, they turned to "pseudo that" observations, that were unsound (Ginat, 2007). One of these students (P16) justified her rationale with unfounded arguments.

The more observant (6) students capitalized on a sound observation that yielded a single read of the input from left to right, with a simple auxiliary memory.

An indication of an already read-value ***v*** *implies that: a* ***later read*** *of the value* ***v-1 means*** *a return to the left of the collection process.*

The operational implementation of the above observation requires an array A, of N binary cells, for keeping the indications of the observation. An elegant solution follows. For the example, in the task statement, the initial values of A will be: 0 0 0 0 0 0; after reading the input 4, A will become: 0 0 0 1 0 0; after reading the input 5 and 1, A will be: 1 0 0 1 1 0; and upon reading the input 3, A will be updated accordingly (a 1 will be put in its 3-rd bit), and the **counter** of the number of returns-to-the-left will be incremented by 1, due to the 1 already appearing in the 4-th bit. The correctness of the above idea may be justified by an invariant property that relates to the meaning of the 0 and 1 values and their locations in A.

The students who displayed this solution did not phrase the above observation, but did recognize it, and expressed levels 3 and 4 of the PGK hierarchy, with relation to the task characteristics and the array utilization.

Table 1 summarizes the solutions statistics of stage-1 tasks T1–T3. (In T2, three participants were excluded from the statistics, due to a vague solution.) We initially regarded T3 harder than T1 and T2. This will also be the case in the 3 tasks of the next stage – stage-2.

Table 1
Statistics of the experimental group in stage-1

	Demonstrated Abstraction
T1. a-b substrings	**37.9%**
T2. Increasing sub-sequences	**46.2%**
T3. Permutation collection	**20.7%**

5. Findings of Stage-2

This section displays our findings and analysis of the experimental group solutions and the control group solutions of the post-intervention questionnaire, of 3 algorithmic tasks (named T4, T5, T6). We regarded both questionnaire difficulty levels similar (including T6 harder than T4, T5, similarly to T3 vs T1, T2). We display each task followed by its solutions, and mention some written student comments on their solutions.

The 29 experimental-group students and the 31 control-group students were given 90 minutes for solving the 3 tasks. They were allowed to provide their solutions in pseudo-code or a programming language, and were requested to pay attention to correctness and efficiency, and briefly write the ideas underlying their solutions. Six students of the experimental group were interviewed about their view of the intervention experience and its outcome.

5.1. *c-c-c Substrings*

The first task involved string processing, somewhat different from that of stage-1.

T4. c-c-c substrings. Given a string of N characters from the **abc**, output the number of substrings that start with a **c**, end with a **c**, and between them there is exactly one (more) **c**.

Example: In the string **cdbccacbacacab** there are 4 such substrings. The leftmost of them is **cdbcc**, and the rightmost is **cbacac**.

Many students of the control group started writing a solution right after reading the problem, and 17 of them offered a brute-force solution, with no insight into the problem. The experimental group students spent more time trying to analyze the problem, but not all of them came up with the relevant observation – 11 of them also offered the brute-force solution. The rest of the students in both groups noticed the following observation:

*The number of **c-c-c** substrings equals the number of **c**'s in the string minus 2.*

These observant students realized that every c in the string, apart from the last two, starts exactly one c-c-c substring, since a c-c-c substring includes exactly 3 **c**'s. This declarative observation relates to level-4 of the PGK hierarchy of abstraction, while the brute-force solution expresses only level-2.

5.2. *Permutation Sorting*

The second task involved special sorting.

> **T5. Permutation sorting.** Given a permutation of the N integers 1..N, kept in an array A, sort the permutation into a decreasing order of values, so that it will be ordered in the form: N (N − 1) (N − 2) .. 2 1.

One should pay attention to efficiency here, and not hastily invoke some familiar scheme. Hasty tendency may lead to an inefficient solution. Some of the students in both groups – 12 in the control group 8 in the experimental group – employed a standard sorting scheme, usually Selection Sort, in which: N is <u>searched</u> and put in the left end, then N − 1 is <u>searched</u> and put next to it, and so on. This solution does not capitalize on the special problem characteristic that the values kept in A are a permutation of 1..N. An efficient solution capitalizes on this characteristic, and relates values to locations, based on the following observation:

> *The <u>leftmost</u> value in A that is not yet in its final destination may be put <u>directly</u> there by swapping it with the value that occupies its destination. (Notice that the final destination of value i is N − i + 1.)*

The rest of the students realized this declarative observation, and implemented a corresponding operational solution: the leftmost value in A will be swapped with the value that occupies its location; then, the new value in the leftmost location of A will be handled similarly, and so on, until N will "arrive" to the leftmost location of A; at this point the computation will proceed in the same way with the second-from-left location in A; and so on. Six students of the experimental group wrote an explanation of their algorithm behavior with invariants equal or similar to the one below:

> *Swaps will be performed without searching. After k swaps, at least k of the values will be in their final destination in* A.

The upper observation expresses the 4-th level of the PGK hierarchy, as it relates to a "that" characteristic of the task. The invariant observation expresses the 3-rd level of the hierarchy, as it captures the underlying property of the elegant computation described above.

5.3. *Interleaved 0-1*

The last task is harder than the previous two, and requires a different observation that relates to locations of values.

> **T6. Interleaved 0-1.** Given a list of 2N digits – N 0's and N 1's, arbitrarily ordered – output the <u>minimal number</u> of swaps between two digits that yield the interleaved list: 0 1 0 1 .. 0 1. A single swap may be performed between two digits that may or may not be adjacent. Note that only the number of swaps is required (and not the actual swaps).

Example: For the string 1 1 0 1 1 0 0 0 two swaps are required (for example, a swap between the first 1 and the last 0, and a swap between the 5-th and the 6-th digits).

This problem is more challenging than the previous one. Some of the students felt initially at loss. Their tendency was to "break" sub-sequences of equal digits, by swapping between 1's in a sequence of 1's and 0's in a sequence of 0's. More than two thirds (22) of the students of the control group offered solutions that are based on counting sub-sequences of 0's and 1's. Some of these solutions also related to lengths of these sub-sequences. Slightly less than half (14) of the students of the experimental group also followed this direction. Unfortunately, this direction involves heuristic conjectures that lead to erroneous solutions. As in the third task of stage-1 (T3), they turned to pseudo "that" observations, that were unsound.

The difficulty with the above solutions is that they are based on relative locations of adjacent values, while the key characteristic here is based on absolute locations. The following two observations relate to this characteristic:

All the 0's should be in odd locations, and all the 1's – in even locations.

If a 1 initially occupies a location of a 0, then there must be a 0 that occupies the location of this 1; and vice versa.

One swap may concurrently put in place 0 and 1 that are initially not in place. Thus, the total number of needed swaps equals the total number of 1's that are initially in odd locations. The above assertions imply minimality, as a single swap cannot "take care" of more than two digits at a time. A simple operational implementation follows.

The remaining students in both groups (9 of the control group and 15 of the experimental group) noticed and wrote the first observation above. A few in each group did not turn to the second observation, and offered cumbersome solutions. Still, these students reached sound declarative observations of a "that" nature, and related to abstractions of level 4 of the PGK hierarchy.

6. Within Group and Between Groups Comparisons

We conducted comparisons to assess success of the intervention – comparisons within the experimental group, and comparisons between the experimental and the control groups. We first display percentage comparisons in Tables 2 and 3 below, and then display Table 4 with results of statistical calculations. Finally, we display sayings of the experimental group students about their learning.

We remind the reader that the within group comparison refers only to the experimental group, who attended the stage-1, T1–T3 questionnaire and then attended the stage-2, T4–T6 questionnaire after intervention. The between groups comparison refers to both groups, who attended the T4–T6 questionnaire, after studying the same learning materials in different ways for the same amount of time.

In order to assess rates of success, we analyzed the above data using the General Estimating Equation (GEE) procedure, which allows to relate to the binary outcomes

Table 2

Experimental group demonstration of abstraction in the tasks of both stages.

Pre-intervention Abstraction		Post-intervention Abstraction	
T1 of stage-1:	**37.9%**	**T4** of stage-2:	**62.1%**
T2 of stage-1:	**46.2%**	**T5** of stage-2:	**72.4%**
T3 of stage-1:	**20.7%**	**T6** of stage-2:	**51.7%**

Table 3

Demonstration of abstraction of both groups in the tasks of stage-2.

	Control Group Abstraction	Experimental Group Abstraction
T4. c-c-c substrings	**45.2%**	**62.1%**
T5. Permutation sorting	**61.3%**	**72.4%**
T6. Interleaved 0–1	**29.0%**	**51.7%**

Table 4

GEE results for time, group, problem, and interaction effects

Type of Comparison	Tested Effects		
Pre vs. Post Intervention	**Time Effect:** Wald = 29.96, p <. 001	Problem Effect: Wald=9.19, p=.010	Interaction Effect: Wald = 2.67, p = .263
Experimental vs. Control	**Group Effect:** Wald = 3.04, p = .081	Problem Effect: Wald = 11.86, p = .003	Interaction Effect: Wald = 0.53, p = .796

Note. Wald refers to the Wald's χ^2 test for significant effect

(abstract/non-abstract) of the student solutions of the tasks (Hardin and Hilbe 2013). Table 4 displays the analysis results.

Table 4 displays sizes of four different effects – time effect, group effect, problem effect, and interaction effect. Effect sizes are qualitative measures of the magnitude of the experimental effect. Each effect is displayed with a Wald value and a p value. The *Wald value* relates to a measure of magnitude, and the *p value* indicates its "solidity". The lower the p value, the more solid is the effect.

The *time effect* above measures the improvement within the experimental group, between the pre intervention and the post intervention; the *group effect* above measures the achievement difference between the two groups in their responses to the T4–T6 questionnaire; the *problem effect* above measures the challenge-gap for the groups between the first two tasks and the third one; and the interaction effect measures the mutual influence between the time and problem effects of the first (top) comparison, and between the group and problem effects of the (bottom) second comparison.

Table 2 and Table 4 show that the intervention yielded a significant progress in the demonstration of abstraction by the experimental group students. This is particularly noticeable in the time effect of Table 4. Improvement between the stages was shown in all three tasks of the questionnaire, and especially in the third task, that was harder than the other two in both questionnaires. Table 3 and Table 4 show better performance in stage 2 of the experimental group compared to the control group. The group effect in Table 4 is not as solid as the time effect in the table, but still is significant. The problem effects in Table 4 reflect consistency in the difference between the third, harder task in each stage and the first two tasks. The interaction effects show very limited influence between the problem effects and the time effect and group effect, and enhance the validity of the analysis.

We interviewed six students of the experimental group on their feelings about the intervention. Some particular sayings were:

> "I liked the idea of examining the problem first, and not a right-away algorithm."
>
> "I would like to practice more. This was fun! Can you give us some more problems to solve? I felt that this experience made me a better problem solver."
>
> "It was difficult for me to phrase declarative observations. But although I struggled, and not succeeded much, I enjoyed it. The observations catch the essence. I like that."
>
> "I tried to find analogies to studied examples. I did not find exact ones, but still borrowed some ideas that helped me choose a direction."
>
> "I did not succeed in all the three tasks (of the second questionnaire), but I still felt that being aware of problem characteristics helped me in the solution process."

All in all, our impression was that seeking declarative observations and phrasing them were not easy to some of the experimental group students. But, many enjoyed the challenge, and their awareness of examining the problem carefully, in a very particular way of seeking observations yielded progress; not only in their competence, but in their view of the problem solving process.

7. Discussion

The study displayed here illustrate the important role of declarative observations already in early algorithmic thinking. The posed problems in both stages of the study were short and required basic computational structures and design patterns. Yet, their solutions demonstrated that operational implementation is insufficient even at the early stages of novice programming. One should be well acquainted with coding, but must also develop awareness and competence in problem solving abstraction. Such abstraction should not only involve top down design and modelling (which is practiced to some extent), but also concise extraction of problem characteristic.

We showed in our examination of student solutions the tendency of novices to brute-force and direct translation of problem specifications into coding, without capitalization on problem characteristics. The key to the suitable solutions was concise observations of various kinds. Sometimes they involved recognition of simple input features on which to focus (e.g., V shapes), sometimes they involved a reverse perspective, and sometimes they involved particular attention to value locations. The observations were declarative. They were based on I/O features of the given problems. They did not specify a way of computation, but only problem characteristics on which operational implementations may capitalize. The declarative phrasing expressed the abstract level which they encapsulated.

We related declarative observations to levels 3 and 4 (the object level and the problem level) of the PGK hierarchy of abstraction. These declarative observations widen, in a sense, the range of elements originally indicated for these levels (such as complexity measures, solvability, and problem equivalence). They illuminate patterns on which operational implementations may capitalize, and justifications may be based upon. Although we did not require justifications of solutions here, one could see the potential of using such observations (e.g., observations of the last problem, for arguing minimality).

The findings illuminate novice tendencies to "go for the how" at the beginning of the process of algorithmic problem solving, with very limited metacognitive awareness of their solution processes. Cognitive competence was limited as well. We displayed an intervention for raising both awareness and competence. The intervention involved emphasis of declarative observations, and underlining of the "that" perspective during the problem solving process.

The statistical findings of stage-2 of the study and the student sayings demonstrate the assets of the intervention. These students developed abstraction competence following their elaborated awareness of the relevant role of the "that" perspective. They also seemed to enjoy the process of which they became aware. It is our belief that tutors should encourage activities like the ones displayed in the intervention (including the questionnaires' tasks) in developing their students' awareness and competence in the problem solving process.

References

Aho, A.V. and Ullman, J.D. (1972). *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972.

Armoni, M., Gal-Ezer, J., Hazzan, O. (2006). Reductive thinking in computer science. *Computer Science Education*, 16(4), 281–301.

Bloom, B/S. (1956). *Taxonomy of Educational Objectives: The Classification of Education Goals. Cognitive Domain. Handbook 1*, Longman.

Chi, M.T.H. and Basok, M. (1989). Learning from examples via self-explanation. In: Resnick, L.B., *Knowing, Learning, and Instruction, Essays in Honor of Robert Glaser*. Lawrence Erlbaum Associates, 251–282.

Dale, N. and Walker, H.M. (1996). *Abstract Data Types: Specifications, Implementations, and Applications*. Jones and Bartlett Learning.

Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J., Young, P.R. (1988). Computing as a discipline. *Communications of the ACM*, 32(1), 9–23.

Dijkstra, E.W. (1972). The humble programmer. *Communications of the ACM*, 15(10), 859–866.

Dijkstra, E.W. (1989). A debate on teaching computer programming. *Communications of the ACM*, 32(12), 1397–1414.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Ginat, D., (2007). Hasty design, futile patching and the elaboration of rigor. *Proc of the 12th Conference on Innovation and Technology in Computer Science Education – ITiCSE*. ACM Press, 161–165.

Ginat, D. and Blau, Y. (2017). Multiple levels of abstraction in algorithmic problem solving. *Proc of the 48th Computer Science Education Symposium – SIGCSE*. ACM Press, 237–242.

Haberman, B. (2004). High-school students' attitudes regarding procedural abstraction. *Education and Information Technologies*, 9(2), 131–145.

Haberman, B., Averbuch, H., Ginat, D. (2005). Is it really an algorithm? The need for explicit discourse. *Proc of the 10th Conference on Innovation and Technology in Computer Science Education – ITiCSE*. ACM Press, 74–78.

Haberman, B. and Ben-David Kolikant, Y. (2001). Activating "black boxes" instead of opening "zippers" – a method for teaching novices basic CS concepts. *Proc of the 6th Conference on Innovation and Technology in Computer Science Education – ITiCSE*. ACM Press, 41–44.

Hardin, J.W. and Hilbe, J.M. (2013). *Generalized Estimating Equations*. Chapman and Hall.

Hartmanis, J. (1994). Turing award lecture on computational complexity and the nature of computer science. *Communications of the ACM*, 37(10), 37–43.

Hazzan, O. (2002). Reducing abstraction level when learning computability concepts. *Proc of the 7th Conference on Innovation and Technology in Computer Science Education – ITiCSE*. ACM Press, 156–160.

Hegarty, M., Mayer, R.E., Monk, C.A. (1995). Comprehension of arithmetic word problems: a comparison of successful and unsuccessful problem solvers. *Journal of Educational Psychology*, 87(1), 18–32.

Loksa, D., Ko, A.J., Jernigan, W., Oleson, A., Mendez, C.J., Burnett, M.M. (2016). Programming, problem solving, and self-awareness: effects of explicit guidance. *Proc of the 2016 CHI Conference on Human Factors in Computer Systems*. ACM Press, 1449–1461.

Mani, M. and Mazumder, Q. (2013). Incorporating metacognition into learning. *Proc of the 44th Computer Science Education Symposium – SIGCSE*. ACM Press, 53–58.

Perrenet, J., Groot, J.F., Kaasebrood, E. (2005). Exploring students' understanding of the concept of algorithm: levels of abstraction. *ACM SIGCSE Bulletin*, 37(3), 64–68.

Polya, G. (1945). *How to Solve it*. Princeton University Press.

Prather, J., Pettit, R., Becker, B.A., Denny, P., Loksa, D., Peters, A., Albrecht, Z., Masci, K. (2019). First thing first: providing metacognitive scaffolding for interpreting problem prompts. *Proc of the 50th Computer Science Education Symposium – SIGCSE*. ACM Press, 531–537.

Ryle, G. (1946). Knowing how and knowing that. *Proceedings of the Aristotelian Society*, 46, 1–16.

Sheth, S, Murphy, C., Ross, K.A., Shasha, D. (2016). A course on programming and problem solving. *Proc of the 47th Computer Science Education Symposium – SIGCSE*. ACM Press, 323–328.

Statter, D. and Armoni, M. (2016). Teaching abstract thinking in introduction to computer science for 7-th graders. *Proc of the 11th Workshop in Primary and Secondary Computing Education*. ACM Press, 80–83.

Wing, J. (2006). Computational Thinking. *Communications of the ACM*. 49(3), 33–35.

Zhu, X. and Simon, H.A. (1987). Learning mathematics from examples and by doing. *Cognition and Instruction*, 4(3), 137–166.

D. Ginat – is the head of the Computer Science Group in the Science Education Department at Tel-Aviv University. His PhD is in the Computer Science domains of distributed algorithms and amortized analysis. His current research is in Computer Science and Mathematics Education, with particular focus on various aspects of problem solving and learning from errors.