Olympiads in Informatics, 2011, Vol. 5, 3–11



# Algorithmic Problem Solving and Novel Associations

David GINAT
*Tel-Aviv University, Science Education Department*
*Ramat Aviv, 699978 Tel-Aviv, Israel*
*e-mail: ginat@post.tau.ac.il*

**Abstract.** We elaborate on the essential role of novel associations between recognized task patterns and invoked algorithmic schemes, during algorithmic problem solving. We display three algorithmic tasks of different levels of difficulty, and characterize them by their required pattern-scheme associations. We display diverse student solutions to the tasks, which reflect different levels of competence; and suggest a series of considerations of which tutors should be aware upon selecting and posing algorithmic challenges to students.

**Key words:** problem solving, patterns, algorithmic schemes.

## 1. Introduction

The domain of algorithmic problem solving involves tasks of different levels of difficulty. The less challenging tasks involve straightforward analysis and familiar utilizations of algorithmic schemes (templates, or design patterns; Astrachan *et al*., 1998; Linn and Clancy, 1992). The more challenging tasks require insightful observations and more involved utilizations of such schemes. A primary objective of a tutor who poses more challenging tasks is to examine whether problem solvers reach suitable observations, and associate them with corresponding compositions of algorithmic schemes. The corresponding compositions may derive from novel associations between observed task patterns and the composed schemes.

For example, one basic, familiar scheme is that of Binary Search, which is commonly employed on an ordered list of input values. Yet, it may be used in less familiar ways, such as searching in a range of successive integer values (e.g., a sequence of guesses, in which the answer for each guess is "got it", "too low", or "too high"). The less familiar ways may be obvious to an experienced problem solver, but may require some novel associations from an inexperienced one, who is only acquainted with Binary Search on a list.

In the design of challenging tasks, which are aimed for competitions of algorithmic problem solving, a task designer would usually like to examine contestants' original associations, in less familiar or even novel ways. Such associations are essential components of competition tasks. The objective of this paper is to elaborate on this notion through some examples, and characterize differences between problem solvers who reach such associations and problem solvers who do not.

In the next section we display three tasks that require associations, which in our experience are less familiar to students. We usually pose these tasks during the beginning of our second stage (top 40 students) of our Olympiad activities. The tasks are used for both evaluating the students' problem solving abilities and teaching them implicit elements of problem solving. The students offer solutions that involve different levels of competence, which reflect various associations (or lack of associations) between observed task patterns and employed algorithmic schemes. We display for each task its diverse student solutions.

We conclude with a discussion section, in which we suggest a series of considerations that arise from the displayed tasks and the student solutions. Tutors' awareness of these considerations may assist teaching and learning in general, and the design, selection, and characterization of challenging tasks in particular.

## 2. Novel Associations in Algorithmic Tasks

In what follows, we present three algorithmic tasks, of different levels of difficulty, which require novel associations. In the solution of each task, one has to recognize patterns on which to capitalize, invoke a suitable algorithmic scheme, and employ it in a rather less familiar way. The difficulty of each task stems from the task's hidden patterns and the novel associations required for invoking and employing the suitable algorithmic schemes. The more difficult it is to recognize the patterns and yield the associations, the harder the task.

The first task involves a simple pattern and simple scheme utilization. The second involves a hidden pattern and a sub-component of a familiar (though slightly subtler) scheme. The third task also involves a hidden pattern and a feature of a familiar scheme, but the feature is an *implicit* characteristic of the scheme, rather than an explicit sub-component. Each task is presented in a separate subsection. After the presentation of each task, we first describe our experience with unsuitable allies that were followed by some of the students, and then present the suitable solutions, obtained by other students, which derive from novel associations.

### 2.1. *Simple Patterns and Simple Schemes*

The following task is known as the Longest Stuttering Subsequence Problem. One solution of the task is rather simple, and another is much more involved (Mirzaian, 1987b). We aimed at the simple solution. Although simple, it still requires some association, which in our experience is not straightforward to all problem solvers.

> **Longest Stuttering Subsequence.** Given a string A, of length $n$, and a string B, of length $m$, where $m \leqslant n$; output an integer $k$, which indicates the largest stuttering of B in A. We say that B is *stuttered* in A $j$ times, if A contains $j$ (not necessarily adjacent) appearances of the 1st symbol of B, followed by $j$ appearances of the 2nd symbol of B, ... followed by $j$ appearances of the last symbol of B.
> For example, if A is 002332256233263 and B is 23, then the longest stuttering of B in A is 3. If B is 236, then the longest stuttering is 1.

It is clear that the output cannot exceed the value of $n/m$. In our experience, students approached this task in three different ways. They all noticed the stuttering characteristics that, "if there is no stuttering of size $k$, then there is no stuttering of size larger than $k$". But, they associated this observation with very different schemes.

One approach was based on backtracking, and involved very little capitalization on the task characteristics. The programs of these students were composed of two stages: their first stage calculated the number of appearances of each symbol of B in A, and determined an upper bound for the output (which may be smaller than $n/m$). Then, they scanned A, up to that bound, with the 1st symbol of B, followed by the 2nd symbol of B, and so on. If the scan reached a point were this bound could not be met by one of these symbols, then the program backtracked to an earlier symbol in A, and retried the scan with a smaller stuttering bound. Unfortunately, this "operational" approach is inefficient, and its implementation is error-prone.

The second approach was better related to the simple stuttering characteristic mention above, and involved linear search for the largest stuttering value. These students' algorithm started either from 1, or (descended) from $n/m$. For every considered stuttering size, $j$, they checked its validity. Correct, but still inefficient.

The students of the third approach capitalized elegantly on the stuttering characteristic and associated the above recognized pattern with the scheme of Binary Search. Their solution obtained the largest stuttering value much more efficiently. They noticed that Binary Search on $k$, the stuttering metric, is relevant here, even though their familiarity with Binary Search mostly involved its application on an ordered list of arbitrary values. They demonstrate some novel associations.

The time complexity of the latter computation is $\mathrm{O}(n \log(n/m))$. The more involved solution of this task (Mirzaian, 1987b) is based on the halving method, employed in the domain of VLSI (Mirzaian, 1987a), which was beyond the scope of our training. Its time complexity is $\mathrm{O}(n)$.

In retrospect, we may notice that the students who followed the first approach, of backtracking, expressed an "operational" ("how to do") perspective, with no underlying pattern. The students who followed the second approach did relate to the simple task pattern, but did not associate it with the most suitable algorithmic scheme. Such association was demonstrated only by the students of the third approach.

### 2.2. *Hidden Patterns and Scheme Sub-Components*

The second task that we display is more involved. It is related to the notions of list ordering and list inversions (Ginat and Garcia, 2005). Like the previous task, here too, we experienced several solution approaches, based of different levels of insight and associations.

> **Widest Inversion.** Given a list of N distinct integers, output the width of the widest inversion; that is, the distance between the two unordered elements that are furthest from one another (e.g., for the input 2 5 1 9 4 7 the output will be 3 – the distance between 5 and 4.)

Students approached this task in four different ways. One way was based on the exhaustive computation of directly finding for each element $e$ the furthest element to its right that is smaller than $e$. The other three solution approaches were more efficient, though not always correct.

Students felt that there is an efficient solution to this task, and some attempted a greedy solution. This solution was not based on any recognized pattern, but rather on a variant of a sub-component of Quick Sort, in which two pointers are "run" concurrently from the two ends of the list. One pointer is set to the left-end of the list and the other – to the right-end. They are advanced concurrently "inwards", one step at a time, until an inversion is found, or until they meet; then, these pointers are advanced separately "outwards", and the output is the widest inversion found in this process.

This two-stage scan – first "inwards" and then "outwards" – yields the correct result for the input 2 5 1 9 4 3. But, does it yield the correct output for all inputs? Not quite. The input: 7 2 4 17 6 5 13 10 18 19 falsifies this scheme, as the "inwards" scan stops when the pointers are at <17,13>, and the folowing "outwards" scan yield the inversion <17,10> as the widest inversion. Yet, this is not the widest inversion (the widest inversion is <7,5>). Some students who realized examples that falsify this approach tried to patch their solutions with some "local patches", which did not really improve the situation.

The third approach we observed was based on relevant insight. Some students noticed that it may be beneficial to look at location differences instead of value differences, as the widest inversion is tied to the largest location difference between unordered elements. Thus, if we know the location of the lowest element in the original list, and the location of the second-lowest in the original list, and so on, then the task may actually be reduced to a task of finding the "largest drop" among these locations.

We exemplify the latter idea, of transforming the point of view into a "location differences" task, with the list 7 2 4 17 6 5 13 10 18 19 above. We create a list of element locations. The locations-list for the above list is: 2 3 6 5 1 8 7 4 9 10. (the location of 2 is 2, the location of 4 is 3, the location of 5 is 6, ... the location of 19 is 10). Notice that the latter list is a permutation of the integers 1...10. We now have to calculate the "largest drop" in this permutation; that is, the maximal difference between two values such that the first among them is larger than the second. The largest drop is obtained from 6 and 1; and this indeed yields the widest inversion <7,5>.

The above insightful point of view may be tied to two algorithmic schemes – sorting (for creating the locations-list) and calculating the "largest drop". The former is of time complexity $\mathrm{O}(N \log N)$, and the latter – $\mathrm{O}(N)$. Thus, the observation of the pattern of an equivalent task, of location differences, yielded a rather efficient solution of $\mathrm{O}(N \log N)$.

Yet, one can still do better. Some students noticed the pattern that only some of the integers in the original list may be candidates for the left-end or the right-end of the widest inversion. For example, in the above list $(7 \ldots 19)$, the 2 cannot be the left-end of the widest inversion, since 7 is larger and is on its left. By the same reasoning, none of the integers 4, 6, 5, 13, and 10 are candidates for the left-end of the widest inversion. Similarly, the 13 cannot be the right-end of the widest inversion, as 10, which is smaller, is to its right. None of the integers 6, 17, and 7 are candidates for the right-end of the widest inversion as well.

Upon examining the candidates for the left-end and the right-end of the widest inversion, students noticed another pattern: the list of the left-end candidates and the list of the right-end candidates are increasing from left to right. The list of the left-end candidates is: 7 17 18 19. The list of the right-end candidates is: 2 4 5 10 18 19. (Notice that the same integer may appear in both lists.)

At this point, students who recognized the above two patterns associated their observations with the sub-component of Merge Sort in which two lists are merged into one, by scanning both lists concurrently. Thus, given the two increasing lists (of the left-end candidates and the right-end candidates), we may scan both lists concurrently from left to right, with two pointers, while finding for each left-end candidate its farthest right-end match. (After the pointer on the left-ends list will be advanced to the next candidate, the pointer on the right-ends list will be advanced as far as possible from its current location.)

The above solution may be implemented in $\mathrm{O}(N)$ time, as both the construction of the lists of the left-end/right-end candidates, and the concurrent scanning, in a Merge Sort manner are of $\mathrm{O}(N)$.

All in all, we may notice that the students who offered the first two solutions did not recognize any pattern on which to capitalize. Those in the second group turned to the "operational" Quick Sort idea, of "running" two pointers "inwards" from the two list ends, but this idea was not based on any relevant observation (and actually yielded an erroneous solution). The students of the latter two solutions did observe relevant patterns, before devising their solutions, and capitalized on these patterns. Some of them associated the original task with another task, which can be solved rather efficiently; and others noticed two insightful patterns and associated their observations with a sub-component of the familiar Merge Sort algorithm.

### 2.3. *Hidden Patterns and Implicit Scheme Characteristics*

The two previous tasks involved sequences. Their solutions involved explicit algorithmic schemes or sub-schemes. The task displayed in this section involves graphs. It is also related to a familiar algorithmic scheme, but the link between the task's relevant pattern and this familiar scheme is not as explicit as it was in the previous two tasks. Here, it is implicit.

> **Non-Modulo-3 Cycle.** Given an undirected graph of N nodes, where the degree of each node is at least 3, output the following: if the graph includes a cycle of length that is not a multiple of 3, then output such cycle; otherwise output "no such cycle".

Students approached this task in two different ways. Both ways were based on the very familiar scheme of DFS (Depth First Search; Manber, 1989). But, there was a big difference between the two ways – one way was based on a most relevant pattern of this scheme, and the other was not.

The vast majority of the students did not recognize any pattern that derives from the task specifications. Being familiar with the notion of recognizing cycles in a graph by back-edges of the DFS, they invoked DFS, in which every cycle discovered by a back

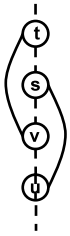


Fig. 1. Three cycles with two back edges.

edge was checked. If a "non-modulo-3" cycle was found, then it was displayed; otherwise, their algorithm notified that there is no such cycle in the graph.

Unfortunately, this hasty, "operational" design is incorrect, as there are graph cycles that may not be examined. We illustrate it with the Fig. 1. If an edge "goes back" from node $v$ to node $t$, and another edge "goes back" from node $u$, which is a descendant of $v$ in the DFS tree, to a node $s$, which is on the tree path from $t$ to $v$; then the algorithm will not examine the cycle: $u - s - \ -t - v - \ -u$. (We use "$-$" to denote a single edge, and "$- \ -$" to denote a path of one or more edges.)

Thus, the algorithm may notify "no such cycle" when there is actually such cycle. Some students noticed this difficulty, and tried to patch a correction, by starting a separate DFS from every one of the graph nodes. However, their patches, which still involved no capitalization on the task characteristics, were also erroneous.

In order to solve the task correctly (and efficiently) one has to recognize a pattern that arises from the task description. In the previous two tasks that we presented, the recognized patterns did not derive from a particular algorithmic scheme that one would have considered for solving the task. Here, it is different. The relevant pattern derives from an *implicit* characteristic of the algorithmic scheme, DFS, which one would consider for the solution.

During DFS, the computation reaches a node which is the end of a branch; i.e., a node v from which the computation "returns up" in the DFS tree. A tree-edge leads to this node. Since the degree of this node, $v$, is at least 3, there are (at least) two back edges "going back" from this node to two nodes, $u$ and $w$, in the DFS tree. These two back edges yield three cycles: 1. $v - w - \ -v$; 2. $v - u - \ -v$, and 3. $v - u - \ -w - v$, as may be seen in the Fig. 2.

It is easy to show that at least one of these cycles must be of a length that is not a multiple of 3. The output will be that cycle (and once it is found, the DFS will halt).


Fig. 2. Three cycles with two back edges from an end of a DFS branch.

Very few students noticed this pattern. Those who did recognize it managed to associate the task specification with the above implicit characteristic of DFS. These students demonstrated a novel association that combined an "assertional" perspective, of pattern recognition, with the natural "operational" view of DFS. Unfortunately, the majority of the students failed to do so, and only applied a limited, "operational" perspective of DFS.

## 3. Discussion

We presented different algorithmic tasks that require various kinds of novel associations in algorithmic problem solving. We displayed student solutions to these tasks, and revealed strong correlation between problem solving competence and the recognition and capitalization on novel associations. The required novel associations and the diverse student solutions illuminate a series of considerations of which tutors should be aware in designing and posing algorithmic challenges to students. We list and describe them below.

- **Operational and assertional perspectives.** These two perspectives encapsulate two different viewpoints of an algorithmic solution. The former focuses on "how" should a computation be performed, and what are the algorithmic schemes that describe the computation operations. The latter focuses on "what" are the characteristics underlying the computation. Both perspectives are essential. While the operational perspective is natural, the assertional perspective involves hidden patterns, which may not be easy to unfold. Problem solvers should be aware of both perspectives, and particularly seek characteristics on which to capitalize, and assertions that capture solution behaviours (Disjkstra *et al.*, 1989). Yet, our findings reveal that students do not always demonstrate the latter. The more competent students are well aware of seeking characteristics on which to capitalize, while the less competent ones do not seek, or only partly seek them.
- **Hastiness.** The less competent students do not recognize sufficient task characteristics. This phenomenon may derive from lower competence in unfolding hidden patterns; but may also due to the undesired problem solving discipline of hastily "jumping" to the composition of an algorithm, without conducting a suitable, thorough task analysis. This could be seen with the less competent students in all our examples – "jumping" into backtracking in the first task; "jumping" into the erroneous (Quick Sort based), greedy solution in the second task; and not seeking any DFS characteristic in the third task. The particular occurrence of hasty greedy solutions is described further in Ginat (2003).
- **Lack of rigor.** In algorithmic problem solving it is often the case that the problem solver is not asked for any assertional argument regarding the correctness of her/his solution. This may increase the possibility of erroneous solutions. Such an occurrence is particularly evident when students seek efficient solutions, which they are unable to argue as correct. A typical example is displayed in our findings, of the Quick Sort based solution to the second task.

- **Limited flexibility with familiar algorithmic schemes.** One of the key elements in algorithmic problem solving is flexible utilization of familiar algorithmic scheme. The natural way to employ familiar schemes is by using them in a way that is similar to the way(s) seen so far. But, sometimes the suitable way is not analogous to previous experiences. The tasks in the three examples of this paper illustrate this phenomenon. The first task required a less familiar utilization of Binary Search; the second involved a variant of a sub-component of Merge Sort; and the third task required an insightful observation and a corresponding flexible utilization of DFS. The distinction between more competent and less competent students is related to their demonstration (or no demonstration) of flexible utilization of familiar schemes.
- **Limited resources and heuristics.** Challenging algorithmic tasks may require problem solving resources and heuristics that are not always explicitly underlined to learners. The second task in this paper involved such elements. The elegant solution to this task involved the notion of "candidates". This notion appears in algorithmic tasks, such as the Celebrity and the Majority Problems (Manber, 1989), but it is not made explicit as a useful tool for algorithmic problem solvers. So is the relevance of location characteristics rather than value characteristics in list-processing tasks. This was the case in the relatively efficient solution to the second task, which was transformed to a "location differences" task. One example of such an occurrence appears in the elegant solution to the game task of IOI '96, of collecting numbers from the left-end and the right end of a given list (http://olympiads.win.tue.nl/ioi/ioi96/contest/ioi96g.html). The heuristic of reducing a given task into another task is explicitly demonstrated in advanced CS topics (e.g., NP-complete), mostly as a proof means. However, it may also be useful in algorithmic problem solving, as occurred here in the relatively efficient solution to the second task.
- **Absent novel associations.** Novel associations are related to all the above considerations. Upon algorithmic problem solving, a problem solver should apply both operational and assertional perspectives; employ careful task analysis; seek relevant, rigorous patterns on which capitalize; flexibly tie them to the task at hand, using explicit and implicit resources; and create novel associations that yield the suitable elegant solution. Less competent and more competent students differ by their employment of little, some, or much of the above, as could be seen in our findings.

The three tasks presented in the previous section yielded diverse student solutions, which were related to all the considerations described above. We referred in the paper to all these considerations, but primarily focused on the latter one, of novel associations between task patterns and familiar algorithmic schemes. We examined it through different levels of difficulty, and characterized these levels with the three titles of the sub-sections of the previous section. We believe that such characterization may be of benefit for task designers, in describing characteristics of challenges posed to students. Yet, there may be additional facets of characterization. The facet presented here may serve as an initial

facilitator for further ones, which may yield additional core characteristics of challenging algorithmic tasks.

## References

Astrachan, O., Berry, G., Cox, L., Mitchener, G. (1998). Design patterns: an essential component of CS curricula. In: *Proc. of the 29th SIGCSE Technical Symposium on CS Education*, ACM, 153–160.

Dijkstra, E.W. *et al.* (2003). A debate on teaching computing science. *Communications of the ACM*, 32, 1397–1414.

Ginat, D. (2003). The greedy trap and learning from mistakes. In: *Proc. of the 34th SIGCSE Technical Symposium on CS Education*, ACM, 11–15.

Ginat, D., Garcia, D. (2005). Ordering patterns and list inversions. *(Online) Journal of Computer Science Education*, ISTE SIG Publications.

Linn, M.C., Clancy, M.J. (1992). The case for case studies of programming problems. *Communications of the ACM*, 35(3), 121–132.

Manber, U. (1989). *Introduction to Algorithms: A Creative Approach*. Addison-Wesley.

Mirzaian, A. (1987). River routing in VLSI. *Journal of Computer and System Sciences*, 34(1), 43–54.

Mirzaian, A. (1987). A halving technique for the longest stuttering sequence problem. *Information Processing Letters*, 26, 71–75.

**D. Ginat** – heads the Israel IOI project since 1997. He 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, focusing on cognitive aspects of algorithmic thinking.