Olympiads in Informatics, 2012, Vol. 6, 44–52
© 2012 *Vilnius University*

Insight Tasks for Examining Student Illuminations

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

Abstract. Students who attend the national Olympiad process join the activity with diverse programming backgrounds and experience. Thus, in the process of selecting the better students, we examine their competence, given their heterogeneous backgrounds. In various parts of this process, we may want to "mask" programming advantage. This may be done by posing insight tasks that require limited programming knowledge. In this paper, we display such tasks, of various levels of difficulty, and elaborate on their relevance. Particular emphasis is put on the importance of assertional observations, creative operational ideas, and suitable problem representations.

Key words: insight tasks, problem solving, problem representation.

1. Introduction

Consider the following *Find Duplicate* task (posed to me by Eric Roberts from Stanford University). Given an array $\boldsymbol{A}$, of N integers, such that each integer is in the range $[1..N-1]$, devise an $O(N)$ time, $O(1)$ space algorithm for finding an integer that appears more than once in $\boldsymbol{A}$. (Note 1. The array $\boldsymbol{A}$ serves as a ROM, and the $O(1)$-space requirement refers to the RAM needed for the computation. Note 2. There may be one or more integers that appear more than once; output just one of them.)

The above task is a non-trivial algorithmic task. Yet, it requires little knowledge in programming. The desired outcome is an algorithm composed of basic programming features. No special knowledge of advanced computational schemes, data structure or design patterns (e.g., Astrachan *et al.*, 1998) is needed. It may also be posed without the terms $O(N)$ and $O(1)$. (One may require, instead, "no more than $5N$ iterations in the computation, and no more than a few variables".) Yet, two important elements are required – insightful illuminations and creativity.

Olympiad competitions require insight and creativity, embedded in (often rather subtle) programming. When students join the Olympiad activity, their programming backgrounds divert. Some are well acquainted with programming schemes and data-structures, whereas others are not. During the process of recognizing and selecting the top students, we should take into account their heterogeneous backgrounds.

In particular, at various points of the national selection process, we may want to "mask" programming advantages. Gaps in programming competence are much easier to "close" than gaps in insight and creativity. Thus, it may be beneficial to pose to students tasks that require challenging insight, but basic programming, in order to learn about their: insight, illuminating perspectives, and creativity.

In this paper, we offer three algorithmic tasks, which require different levels of insight and creativity. We use these tasks, and additional ones, in our national practices and competitions (in conjunction with more involved programming tasks.) All three tasks involve sequences, which are common entities in programming. Their solutions require rather simple algorithmic schemes, and no knowledge of data structures. However, they require illuminating observations, which are reached by applying suitable problem solving heuristics (Polya, 1954; Schoenfeld, 1985) and discrete mathematics elements. The heuristics particularly involve various perspectives of problem representation. The discrete mathematics elements involve the notion of invariance, element decomposition, induction, basic mathematical principles (e.g., pigeon-hole), basic arithmetic, and features of mathematical entities.

Our objective is to elaborate on the tasks' required illuminations, and demonstrate the relevance of such tasks to Olympiad activities. We indicate our experience with posing these tasks to students. In the next section, we display the tasks, in increasing level of challenge. We then reflect, in the Discussion section, on the relevance of posing (such) insight tasks, with little programming, to Olympiad students.

2. Insight Tasks with Basic Programming

In what follows we display three tasks that encapsulate different insight characteristics. The tasks are suitable for different levels of problem solving experience. The first task is suitable after rather limited problem solving practice, whereas the latter two tasks are suitable after more advanced practice. The reader may better appreciate the illuminations needed in the task's solution, by trying to solve the tasks herself before reading their displayed solutions. Each task is displayed in a separate sub-section, which is titled according to its relevant problem solving perspective and discrete mathematics elements.

2.1. *Graphical Representation and Shape Invariance*

The following task involves on-the-fly sequence computation. We devised this task, in order to examine whether students conceive a sequence of values not just as a collection of separate elements; but rather as a chain of related elements, with some invariant characteristic.

Longest Balanced Sub-sequence. We call a sequence of numbers *balanced*, if the amount of positive values in the sequence equals the amount of negative values. Given a list of N positive and negative integers, output the length of the longest balanced sub-sequence (of consecutive elements).
For example, for the sequence $-1\ \ -5\ \ 1\ \ -7\ \ 8\ \ -6\ \ -9\ \ -2$ the output will be 4 (due to the sub-sequence $-5\ \ 1\ \ -7\ \ 8$ (or the sub-sequence $1\ \ -7\ \ 8\ \ -6$)).

A naïve solution to the task involves the examination of all the $(O(N^2))$ sub-sequences. One way to do that is by examining the longest balanced sub-sequence that

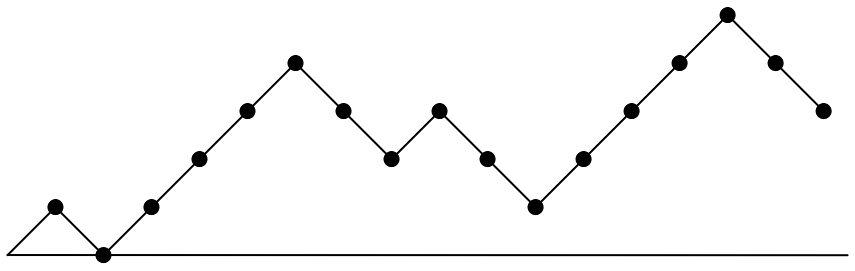

Fig. 1. Graphical illustration of the accumulated value of the difference between the amount-of-positives and the amount-of-negatives of the sequence: $5 \quad -2 \quad 1 \quad 3 \quad 7 \quad 9 \quad -9 \quad -1 \quad 6 \quad -7 \quad -1 \quad 2 \quad 8 \quad 3 \quad 1 \quad -2 \quad -1$ (the horizontal line marks the X axis, i.e., value 0).

starts from each element in the sequence. In our experience, students with limited abstraction competence demonstrate difficulties in devising a better solution.

Others seek illuminations from the basic task feature of the accumulated difference between "positives" and "negatives" throughout the sequence. The difference between the amount-of-positives and the amount-of-negatives accumulated so-far, changes exactly by 1 with each additional element. The intuitive tendency of problem solvers (in our experience) is to look at the absolute value of this difference. This absolute value depends on all the elements read so far. Yet, since the local change, of $+1$ or -1, in this difference depends at each point only on the new, additional element, we may also choose to look at the relative "behaviour" of this difference. One way to look at such behaviour is by drawing its graphical representation, as illustrated in Fig. 1.

In examining the graph of the accumulated difference of the 17-point sequence above, we may notice that the same values appear more than once. For example, the value 0 appears initially, and after processing the 2nd element; the value 1 appears three times (after processing the 1st, the 3rd, and the 11th elements); and the value 2 appears four times.

In the naïve computation, the longest sub-sequence for each starting point is found upon reaching the furthest location in which the initial value 0, of the accumulated difference, returns. However, upon looking at the above graphical illustration, we may notice that it is unnecessary to separately repeat the computation for each starting point. If we remove any prefix of the original sequence, the *shape* of the graphical behaviour of the remaining accumulated difference does not change. The only feature that changes is the "height" of this shape; i.e., its "distance" from the X axis.

This implies that if we start the computation, for example, only after the 4th point, then upon looking at the above shape we may regard the "height" 2 (which would be reached after processing the first four points of the original sequence) as 0; and we should seek the furthest location in which this "height" returns. More generally, we may formulate the following two observations:

*The **shape** of the accumulated-difference behaviour, of any postfix of the original input, **remains invariant** after removing its corresponding prefix.*

A sub-sequence between two points, for which the value of the accumulated-difference is the same, is a balanced sub-sequence.

The above assertions imply that the longest balanced sub-sequence is the longest distance between two locations for which the value of the accumulated difference is the same. Thus, in the example of figure 1, the output will be 12, which is the largest distance between the furthest two points for which the value of the accumulated difference is 3 (points 5 and 17).

The latter observations yield a simple algorithm, of on-the-fly, $O(N)$ time computation, using $O(N)$ space. The algorithm will use an array $\boldsymbol{A}$, in which $\boldsymbol{A}[i]$ will keep the location of the first time that the value of the accumulated difference is i. (We leave open the question whether the computation may be conducted in $O(1)$ space.)

All in all, the algorithm is very simple. It is obtained after reaching the illuminations specified in the above observations. These illuminations derived from a graphical problem solving perspective, which illustrated the shape invariance of the behaviour of the accumulated-difference values.

2.2. *Problem Decomposition and Majority Characteristics*

The task presented in this section was originally posed and solved by Boyer and Moore (1991) (see note there), and later by Misra and Gries (1982). It involves the computation of majority in a sequence of elements. The primary asset of this task, with respect to our focus on illuminating observations, is that it may be solved in several ways, each based on different illuminations. The algorithmic schemes are simple and require no data structures (apart from a few variables). However, the required illuminations are not easily reached.

Majority Number of Appearances. Given a very long list of N integers, output a message saying whether one of the sequence elements appears a majority number of times (i.e., more than $N/2$ times); and if so – output this element.
For example, in the sequence $1 \;\; 2 \;\; 2 \;\; 1 \;\; 2$, the output will be “2 is majority”.
Since the list is very long, it is impossible to keep it all in memory. However, the computation may *read the input twice*.

We posed this task to students in our advanced practice stage. They offered a few different solutions, based on different perspectives of decomposition, as thoroughly described in Ginat (2002). The very elegant solution that appears in the literature was offered by students only after some hint. Yet, a few students devised without a hint a creative solution of slightly higher time complexity (than the literature’s elegant solution). We first elaborate on the student illuminations, and then display the elegant, literature solution.

One perspective that one may attempt upon approaching the task is to turn to the binary representation of the input elements, as each bit may only be of one of two values – 0 or 1. If we look at a single bit across the input, either the appearances of the value 0 will be in majority, or the appearances of 1 will be in majority; or, 0 and 1 will have exactly the same number of appearances. Thus, it may be beneficial not to look at each integer in the input list as a separate unit, but rather follow an orthogonal point of view, and look separately at each position of the integers' binary representation, across the input. This yields an illuminating observation:

If there is a majority, then the value of each of its bits should appear a majority number of times across the input.

We may illustrate the above observation with the example 1 2 2 1 2. The binary representation of these integers is: 01 10 10 01 10. Since 2 is a majority, the value 1 appears a majority number of times in the left (most significant) bit, and the value 0 appears a majority number of times in the right bit.

However, we may obtain a majority value for each bit, without having a majority in the given input. This may occur for example with 1 2 2 3 0, whose binary representation is 01 10 10 11 00. (Although there is no majority, the value 1 appears a majority number of times in the left bit, and the value 0 appears a majority number of times in the right bit.) The case of no majority may also occur if one of the bits' values exactly $N/2$ times appears across the input.

In order to cope with the latter observations, we may seek a way to capitalize on the task characteristic that allows us to read the input twice. The following creative idea is very helpful:

We may divide the computation into two stages. In the first stage, we may compute the majority value for each bit across the input, and construct a ***candidate*** *for majority. If there is a majority, it can only be that candidate. In the second stage, we may go over the input again, and check whether the constructed candidate is indeed majority.*

This nice solution requires $O(N \log M)$ time and $O(\log M)$ space, when the binary representation of each integer requires M bits. (A separate counter should be kept for each bit of the binary representation.) The computation is simple, and requires little memory. Its novel illuminations stem from an orthogonal view of the input's bit representation.

The literature's solution is also based on the notion of candidate, but involves an inductive perspective rather than an orthogonal one. The majority in a list encloses a very simple, but powerful arithmetic characteristic:

If a value v is majority in a list, and we remove from the list two elements – v and an element u not equal to v, then v is still majority in the remaining list.

In fact, simple arithmetic shows that the removal of two non-equal elements even increases the "weight" of the majority in the list (examine, for example, a list of five elements that is reduced to a list of three elements).

The above observation yields a very elegant algorithm, which is based on "removing" pairs of different elements, in an on-the-fly processing manner, using only two variables – a variable that keeps an on-the-fly candidate and a variable that keeps the number of accumulated appearances of the candidate that were not yet matched with different values. The candidate may change again and again throughout the first stage of reading the input. The candidate that remains at the end of this stage is verified as majority in a second stage, as in the previous solution. The detailed, elegant algorithm (see in the indicated literature) requires $O(N)$ time and $O(1)$ space.

All in all, both algorithms are based on simple characteristics of majority, and employ the notion of *candidate*. They both employ the notion of decomposition, but in very different forms – orthogonal decomposition across the input in the first solution, and inductive decomposition in the second one. Although both forms yield simple computations, they are not easy to reach, and require both insightful observations and creativity.

2.3. *Values as Addresses and Cycle Detection*

The task discussed in this section is the **Find Duplicate** task displayed in the Introduction. Although the final solution involves a simple computation scheme, the problem solving process here requires a series of illuminations, which in our experience are not trivial for problem solvers.

In our experience, students that were less acquainted with programming schemes, sometimes performed better than the more experienced ones. In particular, the more experienced students turned at first to familiar searching and sorting schemes that did not help them much. The less experienced students were more "open", and less fixated on familiar schemes, which they only knew for a limited extent.

The initial consideration that should be examined is how to capitalize on the primary task characteristic that all the values in $\boldsymbol{A}$ are in the range $1..N-1$. One simple observation that derives from the pigeon-hole principle is:

1. *Regardless of how we peek N times into cells of* $\boldsymbol{A}$*, we will surely see a value twice.*

We can devise more creative observations from the primary task characteristic (of range $1..N-1$), if we follow the perspective that cell values may be used during computation not only for arithmetic calculations, but also as pointers to addresses.

2. *Since each* ***value*** *in* $\boldsymbol{A}$*'s cells is an integer larger than 0 and smaller than* N*, we may look at each value* ***as a pointer*** *to a cell in* $\boldsymbol{A}$*.*
3. *No cell in* $\boldsymbol{A}$ *points to the cell* $\boldsymbol{A}[N]$*.*

The above observations yield a creative operational idea – to try to "walk" through $\boldsymbol{A}$'s cells:

4. *If we start a "walk" from* $\boldsymbol{A}[N]$*, our first step will surely lead to another cell in* $\boldsymbol{A}$*. (Note that this may not necessarily be the case if we start from* $A[i]$*, for* $i <> N$*.)*
5. *In walking N steps, we will surely visit some cell twice.*
6. *Once we visit a cell again, we also visit its successor again, and its successor's successor again, and so on.*
7. *Therefore, when we visit a cell for the second time, we actually just completed a cycle; and all our future steps will be in this cycle.*
8. *This cycle has an "entry cell", which is the first cell visited in the cycle. Two cells point at this "entry cell" – the cell visited just before entering the cycle, and the last cell of the cycle, just before "starting" it again.* ***These two cells, which point to the cycle's entry cell, keep the same value.***

Thus, our goal is to devise an algorithm that reaches one (or both) of these two cells, and extracts its value. How can we tell that we have reached one of these cells upon "walking" through $\boldsymbol{A}$'s cells? The following creative scheme answers this question:

9. ***After advancing** N **steps** through $\boldsymbol{A}$, starting at $\boldsymbol{A}[N]$, and referring to $\boldsymbol{A}$'s values as pointers, **we are guaranteed to be in the cycle**.*
10. *Once we are in cell c, in the cycle, we may start counting the number of steps that it takes to return to c. **This will tell us the length,** l**, of the cycle**.*
11. *Now we are almost done. We will start a "walk" from $\boldsymbol{A}[N]$ again, and advance l steps. At this point, we will start an additional "walk" from $\boldsymbol{A}[N]$, with an additional pointer. We will continue with both "walks" concurrently. Since the distance between the two pointers that lead the "walks" is exactly l (the cycle length), **it will take us less than** N **concurrent steps until both pointers will reach, at the same time, the two different cells that point to the cycle's entry cell.***

We may observe the algorithm's execution in the following example. Let the array $\boldsymbol{A}$, composed of 12 cells, keep the following values: $3\ \ 10\ \ 2\ \ 1\ \ 4\ \ 3\ \ 10\ \ 2\ \ 7\ \ 5\ \ 1\ \ 9$. We regard the index of the first cell as 1 (and not 0). We will "walk" 12 steps, starting at $\boldsymbol{A}[12]$: $12 \rightarrow 9 \rightarrow 7 \rightarrow 10 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 10 \rightarrow 5 \rightarrow 4 \rightarrow 1$. Now we are in a cycle. We will count the number of steps until we reach the value 1 again. This will take 6 steps. We will now start a new "walk", $w1$, from $\boldsymbol{A}[12]$, and an additional "walk", $w2$, after 6 steps of $w1$. We will proceed concurrently with the two "walks", and after two concurrent steps with the "walks", both "walks" will reach the value 10 – $w1$ will reach it in $\boldsymbol{A}[2]$ and $w2$ will reach it in $\boldsymbol{A}[7]$. The algorithm will output the value 10.

All in all, the time complexity of the computation is $O(N)$, and the space complexity – $O(1)$. We believe that the primary reason for the task's challenging nature is the need to progress gradually through a series of illuminating observations, as displayed above. These observations combine both illuminating insight and algorithmic creativity.

3. Discussion

The three presented tasks involve both assertional and operational observations (Dijkstra *et al.*, 1989). In order to solve each task, one had to first recognize some assertional characteristics, such as the two assertions in the first task, and the initial observations in the other two tasks. But, this by itself was insufficient. After recognizing these characteristics, one had to capitalize on them, and devise a suitable algorithmic solution. In the latter two tasks, this capitalization required creative operational ideas, of how to elegantly conduct the computation.

The assertional characteristics correspond to Ryle's notion of "knowing what", and the operational ideas correspond Ryle's notion of "knowing how", in devising operational schemes (Ryle, 1949). The assertional characteristics encapsulated insightful illuminations that one had to extract from the problems' features. They did not require knowledge in programming and computational schemes, but rather a focused reasoning perspective. A competent problem solver could reach them without advanced knowledge in algorithmics.

The operational ideas in this study did require some knowledge and experience in algorithmics, but only to a limited extent. In the first task (Longest Balanced Subsequence), they involved simple array utilization. In the second task (Majority), they involved the notion of a candidate. Novices see a trivial example of the notion of candidate already in the computation of Max in a list. Here, they had to "take it one step further" and combine it in a two-stage computation. This required creativity. In the third task (Find Duplicate), the required operational ideas were subtler, as they involved cycle detection. Yet, experience with common algorithmic schemes, such as searching and sorting, as well as experience with Data Structures, could not help much. One needed a creative operational idea for conducting the computation.

One primary aspect that was required in all three tasks was that of suitable task representation. In the first task, the illumination derived from graphical representation. In the second task – it derived from an orthogonal view of the data, across the input (the students' solution), and an inductive perspective of majority characteristics in a list. In the third task, the illumination started by viewing the array values as array addresses.

Problem representation, and problem restructuring, play a key role in problem solving, particularly in the case of insight problems (Knoblich *et al.*, 1999; Ash and Wiley, 2006). Both notions relate to Representational Change Theory (Knoblich *et al.*, 1999), and the invocation of a suitable heuristic, of "restating the problem" (Polya, 1954).

We expect Olympiad students to turn to such representation perspectives, as a part of their demonstration of competence in problem solving. Those that meet difficulties with turning to the suitable representation may experience fixation or impasse that inhibits them from making progress. We would like our competent students to overcome impasses of this kind. Thus, part of our student evaluation is based on examining their behaviour in solving tasks like those presented here. Based on our experience, we recommend embedding such tasks during the process of recognizing the better students, at various stages of the national Olympiad activity.

References

Ash, I.K., Wiley, J. (2006). The nature of restructuring in insight: an individual-differences approach. *Psychonomic Bulletin & Review*, 13(1), 66–73.

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.

Boyer, R.S., Moore, J.S. (1991). A fast majority vote algorithm. *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Automated Reasoning Series, Kluwer, 105–117. (The algorithm was invented in 1980.)

Dijkstra, E.W. *et al.* (1989). A debate on teaching computing science. *Communications of the ACM*, 32(12), 1397–1414.

Knoblich, G., Ohlsson, S., Haider, H., Rhenius, D. (1999). Constraint relaxation and chunk decomposition in insight problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(6), 1534–1555.

Misra, J., Gries, D. (1982). Finding repeated elements. *Science of Computer Programming*, 2, 143–152.

Ginat, D. (2002). On varying perspectives of problem decomposition. In: *Proc. of the 33rd SIGCSE Technical Symposium on CS Education*, ACM, 331–335.

Polya, G. (1954). *How to Solve It*. Princeton University Press.

Ryle, G. (1949). *The Concept of Mind*. Barnes and Noble.

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

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.