Olympiads in Informatics, 2022, Vol. 16, 3–11
© 2022 *IOI, Vilnius University*
DOI: 10.15388/ioi.2022.01

Posing Creative Reduction Tasks

David GINAT[1], Shlomit ARIAN[1], Oren BECKER[2]
[1]*Tel-Aviv University, Science Education Department, Ramat Aviv, Tel-Aviv, Israel 69978*
[2]*Centre for Mathematical Sciences, Wilberforce Road, Cambridge CB3 0WB, UK*
e-mail: ginat@post.tau.ac.il, shlomit.arian@gmail.com, oren.becker@gmail.com

Abstract. Reduction is a fundamental computer science (CS) notion (Schwill, 1994). In solving reduction tasks one must think at the problem level, in order to recognize suitable correlations between problems. Thinking at the problem level involves recognition of declarative features of problems. This requires a high level of abstraction. Reduction is well apparent in the more advanced stages of CS studies, but it is also relevant at earlier stages. It may serve as an important means for practice and awareness of abstraction. We designed reduction tasks for the earlier tutoring and training of algorithms students. Our designs are illustrated with three creative tasks of different characteristics. Student solutions for each task are presented and discussed. The students demonstrated different levels of abstraction, insight and flexibility in solving the tasks.

Keywords: reduction, abstraction, task design.

1. Introduction

Algorithmic problem solving involves various techniques, among them divide-&-conquer, backtracking, greedy computation, dynamic programming, and means end analysis. One additional approach is that of transformation, sometimes in the form of reduction. Reduction involves mapping from one problem to another based on strong correspondence between the problems. While it is primarily known as a technique for proving intractability of problems (e.g., NP-complete), it is also relevant in algorithmic problem solving. For example, the problem of Maximum bipartite matching is solved by reducing it to the Maximum flow problem, with constructing a network in which flows correspond to bipartite matchings (Cormen *et al.*, 1990).

Reduction is based on concise and efficient mapping from the entities of a source problem S to corresponding entities in a target problem T, so that the solution of Problem-T yields the solution to Problem-S. The example above demonstrates the relevance of reduction with an advanced algorithmic problem, yet it may be as relevant in solving simpler problems. Since the principal theme of reduction is exploitation of problem correspondence, it requires the problem solver to think at the highest, problem level of Perrenet *et al.*'s abstraction levels of algorithm perception (2005). Thinking at the problem level may deepen one's algorithmic conceptions, which may sometimes tend to focus on the operational "how" of a computation rather than on the declarative "that" of problem features (see (Ryle, 1946) for "how" and "that"). Awareness and practice of the latter are important for competent algorithmic problem solving.

Problem solvers naturally seek problems that are analogous, or similar to their posed problem. Often, they borrow notions, or schemes from other similar problems. When strong analogy in the form of equivalence is recognized, one may borrow another problem's complete solution. The solution may be modified, in adaptation to a given problem, or may be used as a black box, whose inner components are hidden. Reduction encapsulates the latter.

The challenge in reduction involves two phases – Phase-I, of finding a suitable Problem-T to which the posed Problem-S will be reduced; and Phase-II, of providing a concise and efficient transformation of the input of Problem-S to the input of Problem-T. There may also be a need to adjust the output of Problem-T to the output of Problem-S. In Phase-I, the search for a suitable Problem-T requires familiarity with potentially candidate problems, and flexibility upon the examination of correspondence (e.g., noticing that the minimum in a list of negative numbers is the maximum of their absolute-values list). In Phase-II, one should capitalize of Problem-T's features, and develop an elegant and efficient transformation of Problem-S's input to Problem-T's input (and later, possibly adjust Problem-T's output). The processing of the input transformation should not intervene with T's black box computation.

The mapping between problems S and T requires abstraction, in perceiving problems as objects. So is the conception of T's computation as a black box (Perrenet *et al.*, 2005). The focus is on matching structural features of the two problems. Structural matching requires competent pattern recognition (Mayer & Wittrock, 1996; Muller & Haberman, 2008). Armoni *et al.* (2006) illuminated CS student difficulties with these abstraction elements in reductive thinking, among them reduction to the solution, rather than to the problem, and the need to look inside the black box. Ginat and Armoni (2006) showed an example of student difficulties in turning to the notion of complement, when solving the problem of finding a minimum-weight set of edges in a weighted graph, such that each graph cycle has a representative in that set. Students examined graph cycles, rather than turning to a simple reduction.

IOI competitions and training involve problems whose solutions require analogy associations, various transformations, and possibly reductions to other problems. Problem solvers should develop and demonstrate abstraction competencies of thinking at the problem level (in addition to the algorithm level) and matching between structural features. In addition, they should develop creativity in applying mapping between problems, and demonstrate awareness of the importance of sound as well as efficient utilization of black boxes.

We designed throughout the years learning and practice materials for developing and enhancing the above competencies among students, already at early stages of our IOI training. In what follows, we display tasks developed and posed to trainees following our national competition. We also posed some of the tasks to CS students in the second and third years of their undergraduate studies.

The paper illustrates our design and experience with three creative reduction tasks. In the illustrations we describe the design steps and considerations. One design started from a chosen Problem-T, another started from a selected transformation, and a third – from an invented Problem-S. The solution of each task required different flexibility elements. When Problem-S was posed to the students, Problem-T was sometimes provided and sometimes not. We display our experience with students. The student solutions reflect different levels of abstraction and insight into the tasks. The reader may be interested to try solving a task before reading its design description.

2. Task Designs and Solutions

The three tasks presented in this section do not require knowledge beyond searching & sorting and the time complexities of their common algorithms. Each task presentation starts with its design description, continues with the task specification, and ends with our experience with students. In the cases where Problem-T was not provided when Problem-S was posed, students had to demonstrate both phases I and II of the solution process mentioned in the Introduction. When Problem-T was provided, only phase II was relevant. This was still challenging for quite a few.

Arithmetic Shuffle

First, ***Problem-T*** was chosen. The problem input is a list of N integers, and the output is the number of pairs of identical integers; e.g., for the input **5 3 3 2 3 3** the output will be **6**. This problem can be solved in O(NlogN) time by first sorting the list, and then counting the number of identical pairs in the ordered outcome.

Next, a ***transformation*** was chosen. Its output had to be in a format adequate to Problem-T. We chose an N-integer sequence. The transformation was chosen to be: $<x_1 \ldots x_N> \rightarrow <x_1-1 \ldots x_N-N>$. That is, for each element in the original sequence, the transformation subtracts its location from its value.

Then, ***candidates for Problem-S*** were explored. We sought a natural meaning of identical elements in the transformed sequence. To do so, we wrote the expression for identity of elements explicitly: $x_i - i = x_j - j$. This equation *is equivalent to* $x_i - x_j = i - j$. The new formulation suggested a natural meaning.

> *An identical pair of elements in the transformed sequence (Problem-T's input) corresponds to a pair of elements in the original sequence (Problem-S's input), for which the difference between the values equals the difference between the locations.*

Notice that the above new formulation, of $x_i - x_j = i - j$ may be regarded as an *arithmetic shuffle* of the original relation $x_i - i = x_j - j$.

A ***first attempt of Problem-S*** was formulated.

Given a list of N integers $x_1 \ldots x_N$ how many pairs are there such that $x_i - x_j = i - j$?

Then, an ***analysis of the first attempt*** was conducted. The naïve, brute-force solution of the formulated problem is to examine each pair of numbers in the sequence. The time complexity of that is O(N^2), which is far worse than the O(NlogN) – the time complexity of applying the transformation and then solving Problem-T efficiently with the transformed list.

Lastly, a ***refined Problem-S*** was designed, to make it more appealing. We replaced the condition $x_i - x_j = i - j$ with the more natural condition $|x_i - x_j| = |i - j|$. The solution of this formulation of Problem-S is slightly more challenging, as one has to properly handle absolute values. Yet, it is based on the same observations, and its time complexity remains O(NlogN). We let the reader complete the analysis of this formulation.

Problem-S. Values and Locations Distances. Given a list of N integers, output the number of pairs of elements in the list, for which the distance between their values equals the distance between their locations.

Example: For the input **6 5 4 1 2** the output will be **7**, – due to the pairs **6** and **5**, **5** and **4**, **6** and **4**, **1** and **2**, **6** and **2**, **5** and **2**, and **4** and **2**.

Problem-T. The number of pairs of identical elements in a list of N integers.

Problem-T was not provided to the students.

A non-negligible amount of students struggled with this task. Quite a few offered the brute-force solution, sometimes with erroneous attempts to avoid some comparisons. Other students simplified the condition of "distance" between the values to "difference", which may be negative. This did help them realize the original arithmetic shuffle specified earlier, where no absolute values are involved. They recognized the relevance of Problem-T and invoked it. Their output was correct in the cases where the results of the subtractions in both sides of the equation $x_i - x_j = i - j$ have the same sign. The better students showed further insight and provided the full answer.

The main challenge here was to represent Problem-S's specification mathematically, and possibly attempt various manipulations in stages – first manipulations when the absolute values requirement is removed, and then when it is returned. One had to demonstrate creative flexibility of the train of thought. Competence in employing the heuristic of simplification, together with flexible manipulations, expressed abstraction in the sense that one did not immediately seek the "how" of the computation, but rather carefully examined the "that" of Problem-S, sought insight into its hidden patterns, and only then looked for a relevant Problem-T and a suitable transformation.

Padding Transformation

First, a ***transformation*** was characterized. In the previous task the sizes of the input and output of the transformation were equal. However, the sizes of the inputs of problems S and T may not necessarily be equal. One should also be acquainted with cases in which the sizes are different. The idea here was to focus on this notion, without embedding additional challenges.

Next, a ***common transformation feature*** was sought and chosen. Decidability proofs employ the feature of padding when the sizes of the inputs of problems S and T differ. *Padding* is occasionally applied when the input of Problem-S should be augmented. The augmentation may be conducted in different ways. One of them is that of repeatedly adding to the input the same value in a quantity needed, in order to “bring it” to the size of the input of Problem-T.

Then, the ***relation between the inputs*** was defined. The problems S and T may be similar, but differ from one another in a relative value, or position that should be processed. One such case, in which padding may be useful is the following.

If Problem-S will compute the i-th largest element in a sequence and Problem-T will compute the j-th largest, and $j < i$; then Problem-S may be solved by transforming its input to Problem-T, and padding its input in a corresponding augmentation.

At this stage, ***Problem-S and Problem-T*** were formulated. The described augmentation depends on the values of i and j above. We chose these values to be simple, as the focus is on invoking the notion of padding. The values of i and j were chosen to be N/2 and N/3 respectively.

Problem-S will compute the median in an unordered array Arr of N different values, and Problem-T will compute the “thirdian” – the element that is larger than <u>one third</u> of the elements of Arr and smaller than <u>two thirds</u> of the elements.

Finally, the details of ***Problem-S’s input augmentation*** were written and evaluated. An O(N) reduction computation was formulated, as presented below.

Find the Max element of Arr, and add to Arr the N/2 values: Max+1, ... , Max+N/2; i.e. pad Arr with large values to be 3/2 of its original size.

The resulting task was the following.

Problem-S. Median. Given an array of N distinct values, output its median.

Problem-T. “Thirdian”. Given an array of N distinct values, output its “thirdian”, which is the element that is larger than one third of the elements and smaller than two thirds of the elements.

Problem-T was provided to the students.

We posed the task to a limited group of students. Unfortunately, padding solutions were not offered. The students turned to reduce the size of Arr. The main theme that was demonstrated was the removal of elements smaller than the median of Arr, one by one. One removal version involved a utilization of Problem-T as an operator, with repeated calls for the removal of single “thirdians”, one at a time. This reflects a degenerated transformation and exploitation of Problem-T.

Another version involved sorting of Arr, and the removal of an amount of the smallest elements of Arr that will "shift" the median to the "thirdian" position. Students erred with the correct number of removed elements. And, obviously, the computation complexity exceeded O(N).

It seems that students demonstrated different kinds of impasse. Perhaps their lack of experience with the heuristic of auxiliary construction hindered them from choosing the direction of padding the original input. They followed a direction of "in place" computation, with reduction of Arr's size. In addition, Problem-S involved the notion of median, and this may have led some in the direction of ordering calculations, even at the cost of a very inefficient solution. Thinking at the problem level of Perrenet *et al.* (2005) was very limited, and there was no capitalization on the similarity between the two problems for providing a transformation that yields a single, elegant reduction to Problem-T.

Location-Value Relation

First, ***Problem-S*** was designed. A special case of a previously invented problem – the Widest inversion (Ginat, 2008) – may be solved in a simpler way than the original, general problem. The input of the Widest inversion problem is a list of N positive integers, and the output is the largest distance between two unordered integers in the list; e.g., for the input **2 5 4 6 3** the output will be **3**. The solution is not that simple.

In the special case where the list is a permutation of the integers 1 to N, an elegant solution may capitalize on the particular property of a permutation, which is:

When the input is a permutation of 1..N, the range of values is exactly the range of locations.

Next, we sought a ***transformation***. We examined a simple example, and looked at the possibility of transforming a given permutation, such as **2 5 4 1 3** to a list of the locations of the permutation values: **4 1 5 3 2** (e.g., the 1-st value in the new list is **4** since **1** appears in the 4-th place in the original permutation). Upon looking at these two lists one may notice the following:

The "Largest drop" – the largest difference between two unordered integers – in the new list is the widest inversion in the original list.

Thus, if the Largest drop is a simple problem it may become ***Problem-T***. In an **analysis** of this problem, one may notice that the computation is simple – an O(N) time, of one "pass" over the input, where the difference between every newly read value and the current Max is examined. We obtained the following task.

Problem-S. Permutation Inversion. Given a permutation of the integers 1..N, in an arbitrary order, output the largest distance between two unordered integers.

Example: For the input **1 6 2 4 7 5 3** the output will be **5**, which is the distance between **6** and **3**.

Problem-T. Largest Drop. Given a permutation of the integers 1..N, in an arbitrary order, output the largest difference between two unordered integers. The output in the above example will be **4**. This difference occurs twice – between **6** and **2**, and between **7** and **3**.

Problem-T was sometimes provided to the students.

When Problem-T was not provided to the students, they offered two kinds of solutions to Problem-S – a brute-force O(N^2)-time solution, in which the distance between every pair of integers is checked; and an insightful O(N) solution which capitalizes on the observations italicized in the above design.

When we provided Problem-T with Problem-S, and requested a reductive solution, some students indeed offered the above elegant reduction. However, others still did not see the correspondence between the problems, and turned to a brute-force solution. Since they were obliged to solve by reduction, some of them used Problem-T as an operator which receives only a pair of values. Their solution called Problem-T's (black box) algorithm O(N^2) times, a separate call for each pair examined by their brute-force solution.

Some created a list of pairs <i,j>, such that i is greater than j, and is the furthest location of an integer smaller than the integer whose location is j, in the original input. For example, for the input in Problem-S's specification, the list of pairs will be the following: <7,2>, <7,4>, <7,5>, <7,6>. (The 7 in all the pairs is due to the location of 3; the 2 in the first pair is due to the location of 6; the 4 in the second pair is due to the location of 4; the 5 in the third pair is due to the location of 7; etc.) Each pair was computed separately, thus the time complexity is O(N^2).

Both of these inefficient solutions express a "reduction by obligation". The first, "operator based" solution utilizes a degenerated variant of Problem-T's solution, and demonstrates limited abstraction at the problem level. The second solution expresses a slightly higher abstraction by invoking Problem-T's solution only once, but lacks sufficient insight of the correlation between the problems.

3. Discussion

Reduction is a fundamental CS notion. Although it is mostly apparent in advanced courses, it may be a relevant tool also in the Introduction to Algorithms level. It requires recognition of patterns, creativity, and thinking in the problem level. As such, it may be introduced and practiced by IOI students rather early in their training.

The practice of seeking problem correspondence in reduction, as well as applying it properly and efficiently, develops one's analogical thinking and enhances awareness of essential algorithmic problem solving elements, including moving between different levels of abstraction (the problem level and the algorithm level), revealing underlying patterns, and employing flexibility is developing suitable algorithmic schemes.

The first task required the recognition of a mathematical underlying pattern. Although the pattern was simple, one needed flexible manipulations to reveal it. This was also relevant in the third task, where the underlying pattern was a simple location-value pattern. In the second task one had to demonstrate flexibility in turning to a concise, elegant construction. All the tasks involved the application of problem solving heuristics – problem simplification in the first task, auxiliary construction in the second, and a change of representation in the third task. In addition, all the tasks required thinking at the more abstract problem level, both upon seeking problem characteristics and upon mapping from Problem-S to Problem-T.

Students demonstrated various levels of the above. Many did not recognize underlying patterns, expressed limited flexibility, and did not properly relate to efficiency considerations. In addition, some students did not fully capitalize on Problem-T's characteristics. Some invoked its solution repeatedly as an operator, thus demonstrating a degenerated transformation that "misses the point" of reduction. We believe that practice and awareness play a key role in developing suitable, desired competencies. Such a development will help assimilating abstraction, which is one of the most essential elements in computer science and computational thinking.

References

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

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

Ginat, D., Armoni, M. (2006). Reversing: an essential heuristic in program and proof design. In: *Proc of the 38th ACM Computer Science Education Symposium - SIGCSE*. ACM Press, 469–473.

Ginat, D. (2008). Learning from wrong and creative algorithm design. In: *Proc of the 40th ACM Computer Science Education Symposium – SIGCSE*. ACM Press, 26–30.

Mayer, R.E., Wittrock, M.C. (1990). Problem-solving transfer. *Handbook of Educational Psychology*, 47–62.

Muller, O., Haberman, B. (2008). Supporting abstraction processes in problem solving through pattern-oriented instruction. *Computer Science Education*, 18(3), 187–212.

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.

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

Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin of European Association for Theoretical Computer Science*, 53, 274–295.

D. Ginat – served as the head coach of Israel's IOI project in the years 1997–2019 (team leader in 1997–2007). 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, with particular focus on various aspects of problem solving and learning from mistakes.

S. Arian – received her M.Sc. in Computer Science from The Academic College of Tel Aviv-Yaffo. For the last 15 years she is teaching various computer science courses, including Algorithms, Data Structures and Computability Theory. Her current research is in computer science education, particularly about abstraction facets.

O. Becker – served as Israel's Team Leader for the IOI in the years 2009-2014. He is a postdoctoral researcher at the Department of Pure Mathematics and Mathematical Statistics at the University of Cambridge. His PhD connected geometric and measurable group theory to the computer science domain of property testing. His current research is, in addition, on expander graphs, word maps and random groups.