Olympiads in Informatics, 2020, Vol. 14, 77–85

DOI: 10.15388/ioi.2020.06

Operator Utilization and Abstract Conceptions

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

Abstract. Algorithmic challenges occasionally embed the utilization of specified operators. Suitable operator utilization is tied to recognition and capitalization on its characteristics, which should be unfolded and comprehended. In seeking comprehension, one may invoke abstraction perspectives with which to view the operators' concrete features. We display invocations of such perspectives in problem solving of several "unplugged" challenges, and mention our experience with students. Problem solving of such challenges elaborates on comprehension and verification. It enhances conceptual practice, without programming considerations. Practice of the interplay between the abstract and the concrete elevates problem solving competence and confidence.

Keywords: abstraction, problem solving.

1. Introduction

Abstraction is a primary notion in computer science (CS). Common themes of abstraction include top down, design patterns, abstract data types, divide & conquer, recursion, and more. They are related to design, problem solving, and the mixture of both. Problem solving abstractions are relevant at various levels in solving algorithmic challenges (Wing, 2006; Ginat and Blau, 2017). Problem solvers may need to think concurrently at multiple levels of abstraction and "move" back and forth between the concrete and the conceptual. The concrete involves direct consideration of the givens of a problem to solve. The conceptual may encapsulate a perspective of *hiding, or ignoring details* and a perspective of *relating to properties of recognizable parts* (Frorer *et al.*, 1997). Algorithmic problem solvers, including IOI contestants, should invoke perspectives of both kinds.

One element in various IOI tasks is that of operator utilization. An operation or function is specified, and contestants are requested to repeatedly use it, in a suitable and efficient way. Some examples are the task Sorting a Three-Valued Sequence in IOI 1996; the task Median Strength in IOI 2000; and the task XOR in IOI 2002. In the 1996 task, the given operator was a simple exchange operation; in the 2000 task, the given operator

was a function that finds the median of three different values; and in the 2002 task, the given operator was an invert operation (called xor) on a rectangle of black/white pixels in a matrix. In all the three tasks, problem solvers had to relate to properties of the given operator, and capitalize on its characteristics. They also had to ignore details, while focusing on particular parts (e.g., focusing on "imagecorners" in the 2002 task).

All the three operators in the above tasks are simple operators. In addition, the task goals are simply defined (though not easily reached). The solutions require significant insight, but do not require complex computational structures. Correctness verification is fundamental; and efficiency consideration appear in various forms – optimality (in the 1996 task), various time-complexity levels (in the 2000 task), (Horváth and Verhoeff, 2002), and lower bound measures (in the 2002 task).

Operator utilization may be practiced. Suitable practice may enhance experience and familiarity with recognizing, and capitalizing on operator characteristics. Such practice may reduce gaps between abstract perspectives and concrete implementations, and strengthen operator utilization conceptions. The reduction of gaps between one and her object of thought may be a means for enhancing abstraction capabilities (Hazzan, 2002).

Practice of operator utilization may be offered with programming, as in the above IOI tasks. It may also be conducted "unplugged" (Fellows *et al.*, 2005). The former involves conceptual notions combined with applied implementation. The latter may exclude implementation considerations, while solely focusing on relevant abstraction conceptions. This may occur with exploration of hidden patterns and properties of operators, as well as with proof considerations. The tasks may be of various levels of difficulty. A tutor may conduct gradual practice and examination of her students' conceptual competencies, elaborate on abstraction perspectives, and develop student awareness and capabilities.

In this paper we demonstrate "unplugged" practices of operator utilization, with several examples that involve simple operators – `skip`, `reverse`, and `xor`. The operators' utilizations are requested in tasks that involve both algorithmic design and proof consideration. The tasks and their solutions are displayed in the next section. We embedded these tasks in-between programming challenges, at the beginning of the training of our top 30 students (before reaching the IOI level). In the last section we relate to abstraction facets of these tasks, elaborate on their role, and indicate our experience with students.

2. Unplugged Operator Utilizations

The tasks in this section involve rather simple algorithmic schemes, but not necessarily trivial algorithmic properties. In some of the tasks the goal may not be achieved for every input, or initial state. The problem solver should recognize the initial states for which the goal is attainable and the initial states for which it is unattainable. For the former case, an algorithm should be developed, and for the latter case verification of

unattainability should be devised. In another task the goal may always be reached, but a minimal number of operator invocations is required. An argument of minimality should be formulated. Both of the algorithmic schemes and the sound argumentations are based on recognized properties, and involve the abstraction perspectives of relating to properties of recognizable parts and ignoring subordinate details.

Some tasks are rather simple, and some are more challenging. The tasks are short, and different from the IOI tasks mentioned above. Yet, the considerations necessary in their solutions contribute to the practice of abstract conceptions. The first two tasks involve sorting.

Skipping pairs. The operator `skip(i,j)`, $0<i,j<N$, skips the two adjacent integers, in the locations `i`, `i+1` in a list of integers, into the locations `j`, `j+1` (respectively). Given a random permutation of the integers 1..N, $N>10$, sort the permutation using the given operator, or output "Sorting is impossible".

The task specification hints that there may be inputs for which sorting may not be obtained. One sorting scheme may be based on the following: skip the integer 1, together with the integer next to it to the beginning of the permutation; then skip the integer 2; then – the integer 3; and so on. If at the end of this process the rightmost two integers are ordered, then we are done. But what if they are not in order? Can we correct this situation by additional invocations of `skip`? Perhaps there is a better algorithmic scheme? We relate to relevant ordering properties.

The situation in which two integers are not in order is called *inversion* (Knuth, 1973). The sorting goal – an ordered permutation – involves 0 inversions. Two properties that are relevant to examine are: 1. The number of inversions in the initial permutation; and 2. The change in the number of inversions by the operator. In examining the latter, we examine "what happens" when we skip a pair of integers over a third integer, say from right to left. If both of the integers are greater than the third, then the change in the number of inversions is +2; if they are smaller than the third, then the change is -2; if one is smaller and one is greater, then the change is 0. This yields the following *parity property*:

> *In every invocation of `skip`, the change in the number of inversions is an even number (regardless of the skip length). Thus, the initial parity of the number of inversions never changes.*

The above observation implies that if the initial number of inversions is odd, then the goal cannot be obtained, since the number of inversions may never reach 0. Thus, if the rightmost pair of integers is unordered at the end of the skipping process, then sorting cannot be attained. All in all, the recognition of the unattainable cases involved the *operator's property – no change in the parity of the number of inversions*. The next task involves sorting with a different operator.

Reversing 3 and 4 tuples.

A. The operator `reverse3(i)`, $1\leq i\leq N-2$, reverses the order of three adjacent integers in a list, the left one of them being in location `i`. Given a random permutation

of the integers 1..N, N>100, sort the permutation using the given operator, or output "Sorting is impossible".

B. Answer part-A when the operator is `reverse4(i)`, 1≤`i`≤N-3, which reverses the order of four adjacent integers.

In part-A, we may immediately notice that the middle element serves as an "axis", and its location does not change in an application of the operator. The two end elements are swapped. This implies that the operator does not change the parity of the locations of the three elements. Therefore, elements in the odd locations will always remain in the odd locations, and so is the case with elements in the even locations. The operator will yield sorting if and only if all the odd integers are initially in the odd locations.

In part-B no element serves as an axis, and each of the four elements changes the parity of its location. In seeking relevant properties on which to capitalize, it may be beneficial to simplify the task and initially focus on a particular subset of the input. One such subset may be that of reversed permutations that should be inverted. Upon examining short cases of such permutations, one may notice that the cases of N=4,5,8,9 may be sorted (inverted), and the cases of N=6,7.10,11 are problematic. In the cases of N=8,9 one may use N=4,5 as generic templates that will be repeatedly used, in a *rolling* scheme, in which integers will be rolled to their desired destinations. This may be extended for larger values of N for which sorting is possible.

Why do some cases pose difficulties? We may seek the solution by relating to the notion of the number of inversions, as in the first task. In the reversed permutations of lengths N=6,7,10,11, the number of inversions is odd, while in N=4,5,8,9 it is even. This leads us to the change in the parity of the number of inversions of the operator. Indeed, as in the first task, here too, the parity of the number of inversions is not changed by the operator. This *property of the operator* paves the way for explaining unattainability of half of the initially-inverted cases.

At this stage we may widen the range of cases, and examine the general case. As in part-A, rolling may be a useful scheme for "bringing" each of the permutation elements to its final destination. However, as in the previous task, some rightmost elements may not be in order at the end of rolling. If the parity of the number of inversions is initially odd, then obviously sorting may not be attained.

But what if that parity is even among the last four integers, and they are still not ordered? Does this mean unattainability? Perhaps not. Perhaps a more involved ordering scheme is needed? Further abstract conceptions of ordering may shed additional light. We leave the answers to the interested reader.

At this stage we turn to two different tasks with the operator `xor`. In the first task, the focus is on optimality, with respect to the number of operator invocations. In the second task unattainability is relevant again.

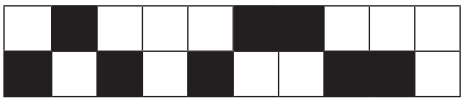
Uniform color 1.

A. The cells in an N-cell row (N>100) are randomly colored black and white. The operator `xor1D(i,j)`, 1≤`I,j`≤N, inverts the color of each of the cells between the `i`-th and the `j`-th cells (including `i` and `j`). Transform the whole row into white with a minimal number of operator invocations.

B. An additional operator is provided – `xor2D(i,j)`, 1≤`I,j`≤N, which inverts the color of each of the cells between the `i`-th and the `j`-th cells in both of the rows. Transform the whole matrix into white with a minimal number of invocations.

In part-A, the elements on which to focus are the vertical sides that separate between black and white cells, and between black end-cells and the "out". We name the total number of these sides `S`. Notice that `S` is always even. The goal is to reduce it to 0. A single employment of `xor1D` may reduce `S` by at most 2, since the relevant impact of `xor1D` is only on the end sides of a given rectangle, and not on the inner sides. An observation of *ignoring details* is that the lengths and locations of chosen rectangles are unimportant, as long as the two ends of each rectangle are sides that separate between black and white. The total number of invocations is `S/2`. It is the minimum.

In part-B, rectangles' dimensions may be 1D or 2D. Below is an example matrix.



Should we consider not only vertical sides, but also horizontal sides? Not quite. The change in the horizontal sides derives from the change in the vertical sides. The horizontal sides may be *ignored*. We may view `xor2D` as reducing `S` by at most 4 vertical sides that separate between black and white.

The algorithmic solution will be divided into two stages – a stage of employing `xor2D` and a stage of employing `xor1D`. The operator `xor2D` will be applied as long as it can reduce `S` by exactly 4 in each invocation. Then `xor1D` will be applied on the remaining sides that separate vertically between black and white.

Can `S` be reduced by 3? In addition, should we specify how we choose a rectangle for an application of the operator `xor2D`, among several choices? The answers to these questions should be part of an argument for minimality of the number of operator invocations. If the answers to these questions are "no", then we may argue for minimality, as each stage of the two stages of the algorithmic solution above would involve a minimal number of steps

Uniform color 2.

A. The cells in an N×M matrix (N,M>100) are randomly colored black and white. The operator `xor2(i,j,d)`, 1≤`i`≤N, 1≤`j`≤M, inverts the colors of the cell <`i`,`j`> and its adjacent cell in direction `d`, where `d` is one of four values – `l`, `r`, `u`, `d` (left, right, up, down). Transform the whole matrix into white, or output "Transformation is impossible".

B. Answer part-A when the operator `xorL` is provided instead of `xor2`. `XorL(i,j,d1,d2)` inverts the colors of 3 cells in an ⌜ or ⌟ shape – the cell <`i`,`j`>, and its adjacent cells in the directions `d1` and `d2`.

C. Answer part-A when the operator `xor3` is provided instead of the previous two. `Xor3(i,j,d)`, 1≤`i`≤N, 1≤`j`≤M, inverts the colors of: cell <`i`,`j`>, its adjacent cell in direction `d`, and the next cell in direction `d`.

In part-A We start by trying to whiten the matrix in a "snake-like" path, starting in the top-left cell and ending in one of the bottom corners, while repeatedly applying the operator on a cell in the path and on the next, adjacent one. All the cells in the path, apart from the last one will become white. There are initial states for which the last cell will become white, and initial states for which it will not. This is justified by the following *invariant property*.

> *The parity of the number of white cells (as well as back cells) never changes.*

If the initial parity of the number of white cells is different from the total number of cells in the matrix, then the goal may not be achieved.

Part-B is subtler. Since the operator operates on three cells at a time, parity of the total number of cells of a particular color is not preserved. We start by trying to whiten the matrix in a "snake-like" path, and progress slightly different in the bottom two lines. Progression in these two lines may be done concurrently, from left to right, until two cells remain – the right-bottom cell and an adjacent one. If all the matrix becomes white, then we are done. But, one or both of the last two cells may be black.

At this point we may seek an illuminating property, or pattern. The operator is applied on three adjacent cells in every application. It may be useful to *divide the matrix cells into three groups*, so that during the "snake like" process, the operator will be applied on one cell from each group. One way of doing so is by adding an auxiliary number to each cell in a "diagonal manner" as follows.

1	2	3	1	2	3	1
3	1	2	3	1	2	3
2	3	1	2	3	1	2
1	2	3	1	2	3	1

Each cell belongs to group1, group2, or group3. It is always possible to apply `xorL` on three cells, such that each is from a different group. In the above diagram, the size of group1 is 10 (cells), the size of group2 is 9 and the size of group3 is 9. It may be proved (possibly by induction), that the difference between the sizes of every two groups is at most 1. This will help us later.

If we implement the "snake like" scheme (with special progress in the two bottom lines) on the above example matrix, we will end up with an all-white matrix, accept for the cell numbered 2 in the right column (which will be black).

Why is that? When we count the initial number of black cells in each group, we notice that that there are three black 1's, four black 2's, and five black 3's. The parity of the number of black 2's is different from that of black 1's and black 3's. When `xorL` is applied, it inverts the color of one cell in each group. This implies the following *invariant property*.

Every application of `xorL` *maintains the difference of the parities of the number of black cells between two groups.*

The above invariant implies that if there is a parity difference between the initial number of black cells of two groups, then this difference will remain; and there will never be a point in which the number of black cells in both groups will be concurrently zero. This explains the result of trying to whiten the matrix of the above example. All the cells of group1 and all the cells of group3 may be white, but one cell of group2 may remain black.

We mentioned earlier that in the numbering of cells, the differences between the amounts of 1's, 2's and 3's are at most 1. This implies that if the parities of the amounts of black cells are initially equal in the groups, then the matrix can be whitened. If the amounts of 1's, 2's, and 3's could be larger than 1, then the condition of equal parities would be an insufficient condition for whitening the matrix, since the amount of black cells could be 0 in one group and 2 in the other.

In Part-C, it is possible to whiten the whole matrix with the operator `Xor3`, except for a 2×2 structure of cells that will remain. In this structure, two groups (out of 1, 2, 3) will have one representative, and one group will have two representatives. While the cells of the two groups with one representative may both be white, the two cells of the third group may be both white or both black. What is a property of the initial number of black cells that guarantees that these two cells will be white in the end of the whitening process? We leave this question to the interested reader.

All in all, the key element that paved the way to the solution of part-B was the *abstract view of looking at the matrix cells as cells that belong to three interleaved groups* of similar sizes, such that `xorL` may be applied each time on one cell of each group. The resulting invariant property yielded the recognition of the cases in which the matrix may be whitened and the cases in which it may not.

3. Discussion

The abstract perspectives of *ignoring details* and *relating to particular properties of recognizable parts* are relevant in problem solving. The task solutions presented here encapsulate them in computations with repeated utilizations of operators. Two operators were used for sorting and one was used for transforming binary matrix values.

The recognizable parts in the sorting tasks were pairs of integers. Unlike various sorting schemes, the focus here was not on pair adjacencies, but on pair inversions. The central element was the *invariant property* of the parity of the number of inversions. This property was the key for understanding the operators' characteristics and their limitations.

The recognizable parts in the matrix transformation tasks were borders between matrix cells. In the first matrix task, the natural tendency is to look at cell values, which are colors of areas. But the important *recognizable parts* are borders between areas; more

specifically – borders between cells of different colors. The focus here was on the *metric property* that relates the initial number of these borders to the amount that can be decreased in a single invocation of the operator. Borders between cells inside a given rectangle were *ignored*. So were possible selections between alternative choices of lengths and locations of rectangles that kept the desired property.

In the second matrix task, the recognizable parts were adjacent cells in a matrix, which *were viewed* in a way that suited a single operator application. Two *properties* led to the solution of part-B of the task: 1. The partition of the matrix cells into three groups of near sizes, so that an application of the operator on any cell of one group may also be on representatives of the other two groups; and 2. The parity differences between the number of black cells in the groups are preserved. The layout of the black and white cells in the matrix was *ignored*. So was the number of white cells. Only the number of black cells counted.

In our experience with students, the more challenging element during problem solving is the decision of what to focus on and what to ignore, as well as the kind of properties to look for. Novices tend to focus on the explicit data in a given problem, and try to associate it with their familiar cognitive schemes. However, a primary theme in solving non-routine problems is the recognition of hidden patterns and capitalization on these patterns (Schoenfeld, 1992). Such recognition should be practiced.

One relevant practice involves problems like the tasks presented here. Although these tasks may be posed as programming problems, their asset is in focusing on abstraction perspectives and verification of recognized patterns. Programming may "bypass" the latter. One may provide a suitable programming solution without sufficient insight into the problem at hand, when only input/output outcomes are examined. An "unplugged" experience is more thorough, and elaborates on the importance of comprehension and verification. The focus is primarily on the conceptual practice. No considerations of computer implementation are involved, and one focuses on exercising exploration and recognition of hidden patterns.

Learners learn definitions, theorems, and methods, but their primary mean for progress is learning from examples and practicing examples (Sinclair *et al.*, 2011). Problem solving with examples like those presented here enhances the practice of relating abstract perspectives to concrete, explicit givens. Repeated practice of relating abstraction perspectives to the concrete develops problem solving competence and enhances one's confidence in her abilities.

We embedded the tasks displayed here in the activities of our top 30 students during the beginning of their advanced training. Although the primary focus of the training was on solving IOI-like tasks, the practice of the tasks displayed here, in-between programming tasks, widened the students' viewpoint and encouraged their verification tendencies. At first, they struggled with the more challenging tasks, and were unsure about the verification of properties; but with further practice they felt more confident, realized the relevance of such tasks, and related observations in these tasks to later programming tasks. This was particularly apparent with the more competent students.

References

Fellows, M., Witten, I., Bell, T. (2005). *Computer Science Unplugged*, LuLu Pub.
Frorer, P., Hazzan, O., Manes, M. (1997). Revealing the facets of abstraction, *International Journal of Computers in Mathematical Learning*, *2*, 217–228.
Ginat, D., Blau, Y. (2017). Multiple levels of abstraction in algorithmic problem solving, *SIGCSE'48*, ACM Press, 237–242.
Hazzan, O. (2002). Reducing abstraction level when learning computability theory concepts, *ITiCSE'02*, ACM Press, 156–160.
Horváth, G., Verhoeff, T. (2002). Finding the median under IOI conditions, *Informatics in Education*, *1*, 73–92.
Knuth, D. (1973). *The Art of Computer Programming, Vol. 3*, Addison Wesley Pub.
Schoenfeld, A. H. (1992). Learning to think mathematically: problem solving, metacognition, and sense making in mathematics, in Grouws D. A. (Ed.), *Handbook of Research on Mathematics Teaching and Learning,* 334–370.
Sinclair, N., Watson, A., Zazkis, R., Mason, J. (2011). The structuring of personal spaces, *Journal of Mathematical Behavior*, *30*, 291–303.
Wing, J. (2006). Computational thinking, *Communications of the ACM*, *49*(3), 33–35.

D. Ginat – headed the Israel IOI project during the years 1997–2019. 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.