Olympiads in Informatics, 2019, Vol. 13, 31–39

DOI: 10.15388/ioi.2019.03

On Implicit Means of Algorithmic Problem Solving

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

Abstract. Students of challenging algorithmics learn and utilize a variety of problem solving tools. The primary tools are data structures, generic algorithms, and algorithm design techniques. However, for solving challenging problems, one needs more than that. Many creative solutions involve implicit notions, whose creative employments yield elegant, concise, and efficient solutions. We elaborate on such notions and advocate their relevance as valuable means in one's problem solving toolbox. We display our experience with students who lacked awareness of these notions, and illustrate the relevant role of three such notions – the notions of "candidate", "complement", and "invariance".

Keywords: algorithmic problem solving.

1. Introduction

The algorithmic problem solving toolbox includes a variety of generic design patterns and techniques. The generic design patterns, or schemes, include algorithms such as sorting, searching, and graph algorithms. The design techniques include problem solving strategies such as top-down design, divide and conquer, the greedy strategy, dynamic programming, reduction, and more (e.g., Cormen *et al*., 1990). Students learn and employ these patterns and techniques, together with a variety of data structures, from the very basic courses to the more advanced algorithms courses. Yet, more can be offered to learners, in particular those of challenging algorithmics.

For example, the very basic pattern of max computation involves the notion of "candidate". The computation involves running through a list of values, while keeping the largest value read so far as the current candidate for the answer. While the design pattern is very basic, the notion of "candidate" is a notion that may be used in advanced, creative ways in challenging algorithmics. One challenging task in which it is creatively applied is the task of seeking majority (i.e., a value that appears a majority

number of times) in a very very long list of values (Boyer and Moor, 1991). The input cannot all be kept in memory (by elements or range), but may be read more than once. Reading is expensive. The challenge is to devise a solution that involves reading the input list as fewer times as possible.

An efficient, elegant solution is based on the following insightful declarative observation: *if the value v is in majority in a list of N values, and we delete v and a different value u, then v is still in majority in the remaining list of N-2 values*. This observation yields an appealing use of the notion of candidate. The candidate will be the element that will remain after deleting pairs of different values, and will be checked for being majority. The computation will involve two passes over the input: a first pass for deleting such pairs and leaving a sole candidate for majority (if there is one), and a second pass for checking whether this candidate is indeed a majority (Boyer and Moor, 1991; Ginat, 2002).

Creative utilizations of the notion of "candidate" appear in a variety of additional task solutions. And there are more soft notions that are repeatedly employed in creative ways in algorithmic solutions. Additional notions are: "complement", "symmetry", "parity", "invariance", "the pigeon-hole principle", and more. Although these notions play a significant role in algorithmic solutions, they are not underlined or explicitly mentioned in textbooks and teaching materials. Perhaps this is due to tutors' assumptions that the implicit utilization of these notions may be sufficient for acquiring them and invoking them. In our experience, this is not always the case, even with talented students of challenging algorithmics. Students may benefit from explicit indication and practice of these notions.

In what follows we describe our 'notion-invocations' experience with the students of our advanced OI stages in the last few years. We display several tasks whose solutions require invocations of such notions, reveal student difficulties, and present elegant employments of these notions. In the last, Discussion section we discuss our findings, and advocate elaboration of these notions as important means of algorithmic problem solving.

2. Implicit Problem Solving Notions

In this section we display three notions that are useful in solving algorithmic tasks. The first notion – candidate – is particularly relevant in algorithmics. The other two notions – complement and invariance – are relevant in both algorithmics and mathematics.

All these notions are related to a declarative point of view, which involves a "what" characteristic perspective. This point of view precedes the operational point of view, which is related to "how", of the computational actions. While novices often 'get by' without the declarative perspective, this perspective becomes more and more important as the algorithmic challenge increases.

The Notion of Candidate

The example in the Introduction displayed a creative utilization of "candidate", based on a mathematical observation. The innovative feature in this utilization involved the decomposition of the computation into two sub-tasks – one of seeking a sole candidate, and another of checking whether this candidate is indeed the desired element. This utilization of the notion of candidate appears in additional tasks. One such task is the Celebrity problem (Manber, 1986), in which a person who knows nobody, but is known to all, is sought.

Utilizations of the notion of "candidate" may have a variety of forms. Another form appears in the solution to the Widest Inversion problem (Ginat, 2011), in which the longest distance between two unordered numbers is sought. The efficient solution of this task is based on pre-processing in which lists of candidates for the left-end and the right-end are found, and then elegantly processed, based on their characteristics.

The following task displays an additional, creative utilization of the notion of candidate, this time in an "as if" manner (Ginat, 2010). In our experience, a non-negligible amount of talented students did not turn to this perspective, but to a naive utilization of a candidate.

> **Longest Plateau.** We define a *plateau* as a sequence of integers in which the difference between every two (not necessarily adjacent) is at most 1. Thus, 4 3 4 3 3 is a plateau, whereas 4 3 4 3 2 is not a plateau. Given a list of N integers, output the length of the longest plateau.
>
> ***Example***: For the input 2 3 3 4 3 5 5 4 3 3 2 3 2 2 1 3 the output should be 6, due to the sub-sequence 3 3 2 3 2 2.
>
> Notice that a sub-sequence (part) of a plateau is also a plateau, though it is a plateau that may be extended to a longer one.

At first glance this task may seem boring and straightforward. Yet, this was not the case for quite a few of our students. It was particularly so when we asked for an extension, in which the definition of a plateau allowed a difference of K, greater than 1 between every two elements of a plateau.

The challenge in the task stems from the possibility of plateaus' overlap. For example, in the sequence 2 3 3 4 3 the plateaus 2 3 3, and 3 3 4 3 overlap, as they share the sub-sequence 3 3.

Many students offered a cumbersome on-the-fly solution using a single candidate for the current plateau. The underlying idea in their solution was to accumulate the current plateau, while handling a series of history considerations of sub-sequences' overlaps. They kept the information of the latest overlap history and manipulated it when an overlap ended. While this idea may be suitable, it yielded cumbersome and erroneous implementations. In addition, it is not extendible for more general plateaus.

One may do better, and simpler if one abstracts the view of the task, by looking at each input value *"as if" it belongs to two "active" plateaus concurrently* – one in which this input value is the lower value, and one in which it is the higher value. Thus, at any

given time, the current element belongs to **two concurrent** candidate plateaus. When one of the candidates ends, its length is compared to the longest plateau so-far.

Each input value v may be viewed as contributing a "block" to its *lower candidate plateau*, and a "block" to its *upper candidate plateau*. For example, upon looking at the sub-sequence 2 3 3 4, starting at the value 2, this value contributes a block to its lower candidate-plateau $p^{1,2}$, and a block to its upper candidate-plateau $p^{2,3}$. The next value, 3, contributes a block to its smaller candidate-plateau $p^{2,3}$ (which was just the upper candidate-plateau of the 2) and a block to its upper candidate-plateau $p^{3,4}$.

Based on the above abstract view, we may devise the following *candidate-plateau scheme*. A new candidate-plateau begins when the next input value contributes a block that does not extend a current candidate-plateau. A candidate-plateau is extended by a block, when the next input value contributes a block to it. A candidate-plateau ends when the next input value does not extend it. Fig. 1 below illustrates visually the scheme with the input of the problem statement.

This concise scheme annuls the need of keeping history details, and requires that we remember at any given time only the length of the longest plateau ended so far, and the two current candidate-plateaus. The scheme's underlying perspective is specified declaratively, by relating two "as if" candidate plateaus to every input element.

The Notion of Complement

The notion of complement is a powerful notion in both mathematics and computer science. Its underlying principle is that sometimes it may be more beneficial to look at "the whole" minus the given elements rather than at the given elements. We illustrate it with the following task (which we learnt from IOI colleagues).

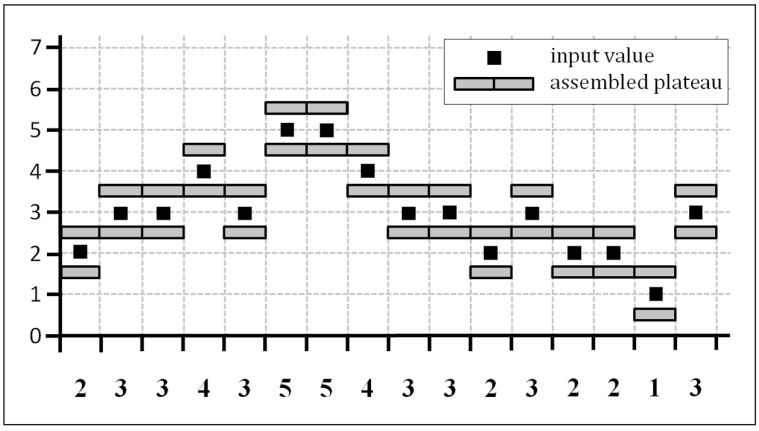

Fig 1. A view of the candidate-plateaus. The input is the sequence of values written under the X-axis. The small black boxes represent the input values, and the grey sequences represent the candidate-plateaus that were assembled from left to right.

Safari. In a very large safari, with N animals, some animals get along with one another (and can stay together), and some do not. For every animal, an indication is provided about all the animals with which it gets along. The number of animals in the safari is a multiple of 3. It is known that exactly 2/3 of the animals all get along with one another. Each of the other animals gets along with some subset of the N animals. The safari management wants to select exactly 1/3 of the N animals to be put together. Given for each animal the animals with which it gets along, output N/3 animals that may be chosen to be put together.

A non-negligible amount of students struggled with this task. All of them represented the task with an undirected graph, and attempted various ways of processing the edges of the specified clique of 2N/3 animals. Unfortunately, their attempts were either erroneous or cumbersome and inefficient. The suitable approach is not to process the given graph edges, but rather the missing edges, of the complement graph.

An edge in the complement graph connects two graph nodes (animals), such that one of them or both of them are not in the specified clique of 2N/3 animals. Thus, if we **remove** the two **nodes** of a missing edge, and all their incident edges, then we remove **at most** one of the 2N/3-clique nodes. By repeating this process again and again, we remove at most N/3 animals from that clique. Therefore, when this process ends, the remaining graph will be a clique of at least N/3 animals. If it is exactly of that size, then all its animals will be displayed as output; and if it is larger, then any N/3 animals taken from this remaining graph will be a suitable output.

All in all, the key point, which paved the way for the surprisingly simple solution, was the utilization of the complement graph of the given input. Capitalizations on complement graphs appear in a variety of graph-based solutions. And the notion of complement appears in many other representations.

The Notion of Invariance

The notion of invariance involves assertions that are repeatedly kept during an iterative or distributed process. It is an essential means in mathematics and computer science. In both domains it is used for arguing and proving characteristics of sets and processes. In computer science it is a fundamental means in the domain of logic of programs, for describing loop executions and distributed-system characteristics. Dijkstra and others (e.g., Gries, 1981; Dijstra, 1989) argue that computer-program design should go hand-in-hand with arguing its correctness, and the design of loops should be based on devising their underlying invariants. However, the notion of invariance is not advocated in most programming and algorithms textbooks, and its essential role is overlooked. We present below two short examples that demonstrate its importance.

Diplomats' Separation. N diplomats enter a large conference hall. Each of the diplomats has **at most** 3 rivals among the others. Rivalry is symmetric. The conference organizers were asked to separate the group of the N diplomats into two sub-groups – A and B – such that each diplomat will be with **at most** one rival in her group. Given

the rivalries of each diplomat, output a separation into two suitable sub-groups, or notify that it is impossible to obtain such separation.

The task is not a difficult task. Yet, some students approached it in a way that was intuitive for them, but not that successful. They offered the following scheme. Start with all the diplomats in sub-group A, and check for each diplomat whether there is more than one rival in her sub-group. If this is the case, then leave one rival in her sub-group, and **transfer the other(s)** to the other sub-group. Repeat this process, with sub-groups A and B, until the desired condition is met.

Although this intuitive process seems suitable, it is questionable, as a diplomat may be transferred back and forth between the sub-groups and it is not clear whether successful termination is guaranteed.

A small modification of the above scheme, supported by invariance argumentation, makes a big difference. When a diplomat is found to be in a sub-group with more than one rival – **transfer that diplomat** to the other group, rather than her rivals. This idea is supported by the following invariant assertion:

> *Each transfer of a diplomat reduces the total number of rival-diplomats inside the sub-group.*

The assertion involves a *measure* (specified in its latter part). It is an *invariant assertion*, since it asserts a characteristic that is kept after **every** diplomat transfer. It implies that the suggested scheme **improves** the situation with every transfer. Even if a particular diplomat is transferred more than once, this iterative process is guaranteed to end, with a suitable separation of the diplomats into two sub-groups, since the decreasing number of rival-diplomats in the sub-groups cannot go below 0. The utilization of invariance and a measure that implies successful termination enfolds the fundamental computer science elements of safety and liveness (Alpern and Schnieder, 1985).

• • •

One sub-domain in which invariance is fundamental is that of mathematical games. Games are present in both mathematics and algorithmics. Many games are two-player games, in which a winning strategy for one of the players is required. The strategy may be specified with an algorithm, but the underlying characteristic that yields and justifies the algorithm is expressed with an invariant. For example, the first task in IOI'96 was a game with a line of 2N numbers, such that each player takes, in her turn, a number from one of its ends. The winning strategy was based on an elegant invariant: *after every move of the first player, the numbers in the two ends of the line are in locations that were originally of the same parity*. The following task also involves a game invariant.

> **DVD Game**. Given a line of 2N cells, N>100, two players play the following game. Each player, in her turn, writes a "D" or a "V" is an empty (not yet used) cell. The first player who completes the sequence "DVD" wins the game. (Players may use one another's written letters in forming the desired sequence.) Devise a strategy for winning the game.

As in the previous task, although the challenge here is rather limited, quite a few of the students offered incomplete or erroneous solutions. Most of them noticed that the winner should be the second player (the one that does not start), and should create in her first two moves a "trap", in the form of D_ _D. The player who will be forced to write a letter between the two D's will lose the game.

At this point some of the students got confused because of the possibility of additional traps created during the game. They were unsure about the winner's responses for the opponent's moves. Their common strategy was to imitate the last opponent move, by writing the same letter that she just wrote, next to her letter. This strategy involved all kinds of cumbersome conditions for avoiding traps. Some were incorrect.

The one thing that many students missed was a simple invariant on which to capitalize:

> *After each move of the first player, there is at least one sequence of empty cells of an* ***odd*** *length.*

Thus, after creating the trap in the first two moves, the winner has a very simple strategy: if it is possible to win the game in the next move, then do so; otherwise, write a "V" in the the middle of an odd-length sequence of empty cells.

It is simple to justify termination of the game with the second player's win. The key point was to recognize and capitalize on the simple invariant. The students who offered the cumbersome, sometimes erroneous solutions indicated that they focused on an immediate reply to the opponent, next to her move, rather than on a global characteristic of the game line. This direction was heuristic, and lacked rigor.

3. Discussion

Chapters of algorithms textbooks are usually designed according to programming constructs, data structures, design patterns, and design techniques. These elements are the primary tools, or means for algorithmic problem solving. But they should not be presented as the only ones. Algorithmic problem solving involves a collection of implicit notions, which may be considered as tools, since they are repeatedly utilized in various ways, particularly in challenging algorithmics.

In many cases the employment of these notions is essential. They pave the way to a desired solution prior to utilization of the primary tools. They belong to the task analysis stage that should progress hand-in-hand with a solution design. While the characteristic of the primary tools is primarily operative, the nature of these notions is declarative. They direct the design and justify its outcome.

Competent problem solvers are acquainted with these notions, and possibly invoke and employ them without explicitly denoting them. They assimilate these notions upon learning and practicing implicit utilizations in a variety of task solutions. But not all students grasp these implicit notions. Explicit elaboration may considerably help, particularly for developing awareness of these notions.

In the previous section we presented our experience with talented students who did not turn to these notions and provided unsuitable solutions. One may say that at least some of these notions (such as the notion of complement) are common knowledge and should not be explicitly underlined. We believe that explicit indication and practice may lead students to seek utilizations of such notions. This is our experience with OI students. Repeated elaborations of these notions enhanced flexibility and abstraction upon approaching task solutions – flexibility in the sense of creative utilizations of familiar means; and abstraction in the sense of conceptual exploration (which yielded, for example, the "as if" perspective in the Longest Plateau example).

Such notions also upgrade one's discipline in the process of problem solving. Awareness may lead one's attempts to employ a declarative point of view prior to an operative implementation. The declarative perspective may yield further insight, which will assist in considering different alternatives of progress. In addition, it may strengthen one's conviction and rigorous argumentation about her solution.

We illustrated creative employments of the notions of candidate, complement, and invariance. The notion of candidate is unique to algorithmics, as it is explicitly tied to progression during a computation process. The notions of complement and invariance are relevant also in mathematics. They may be regarded as resources in the problem solving model of Schoenfeld (1992), in the sense of means with which one should be acquainted for recognizing, specifying, and justifying characteristics. In algorithmics, they are also relevant for devising concise and efficient computations.

More illustrations of these notions will enrich problem solvers' toolboxes. Tutors aware of these notions may embed and underline their explicit utilizations during the teaching of design patterns and design techniques. Repeated embedment and elaboration will raise students' computational thinking competence.

Acknowledgement

I thank my colleagues Hanit Galili and Sharon Zuhovitzky from our Israel OI team for the Safari task story, and its student solutions' summary.

References

Alpern, B., Schneider, F.B. (1985). Defining Liveness. *Information Processing Letters*, 21, 181–185.

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

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

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

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

Ginat, D. (2010). The baffling CS notions of "as-if" and "don't care". In: *Proc of the 41st ACM Computer Science Education Symposium – SIGCSE*. ACM Press, 385–389.

Ginat, D. (2011). Algorithmic problem solving and novel associations. *Olympiads in Informatics*, 5, 3–11.
Gries, D. (1981). *The Science of Programming*. Springer.
Manber, U. (1986). *Introduction to Algorithms: a Creative Approach*. Addison Wesley.
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.

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, with particular focus on various aspects of problem solving.