Olympiads in Informatics, 2021 Vol. 15, 13–21
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.02

Self-Generated Figures in Sequence Processing

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

Abstract. A figure may convey an idea, an argument and even a proof, sometimes better than words. It may also elicit an idea, an argument and a proof. In problem solving, a figure may give a "feel" of a problem. A self-generated figure may help getting insight, or serve as a means for representing one's inner associations, or mental model of the problem. This paper presents self-generated figures in algorithmic problem solving. Students of our IOI advanced stage demonstrated constructive utilization of self-generated figures in solving challenging sequence processing tasks. The figures elicited associations of hidden patterns, whose recognition yielded elegant and efficient algorithmic solutions. We advocate the application and examination of self-generated figures in algorithmic problem solving.

Keywords: algorithmic problem solving, figure drawing.

1. Introduction

One of the essential problem solving heuristics is that of drawing a figure (Polya, 1945; Schoenfeld, 1985; Van Meter & Garner, 2005). Figure drawing is apparent in geometrical problem solving (e.g., Nunokawa, 2004), in the solution of word problems (e.g., Van Essen & Hamaker, 1990), in physics problem solving (e.g., Maries & Chandralekha, 2017), and more. In geometry, drawing involves dynamic examination and manipulation of the givens; in word problems it includes modelling, or representation of relations between objects; and in physics it involves both. These facets are evident in problem solving in computer science and algorithmics as well. For example, in OOP figures are used for modelling, or representing the "world", while in graph algorithms they are used for examining different execution progression scenarios. Algorithmics also involves the tool of visualization, which offers the display of entity values during execution of computer programs (Sorva *et al.*, 2013).

Drawing of figures, and visualization offer means for analysis and comprehension of given problems and their solutions. Yet, current algorithms textbooks do not advocate the drawing of self-generated figures during the process of problem solving. Textbooks use a lot of illustrations – of data structures, relations, functions, generic algorithmic ideas (e.g., binary search), and more. They exemplify and advocate heuristics such as problem decomposition, divide and conquer, backward reasoning, inductive reasoning, and more (e.g., Cormen *et al.* 1990), but do not underline the invocation of the heuristic of visual analysis of a given task. Visual analysis may be very helpful. In our experience with problem solvers, upon IOI training at all level, not many invoked this heuristic, but those who did invoke it often demonstrated its illuminating role in attaining sound solutions. Some of these students' self-generated drawings yielded elegant solutions together with their justifications. The drawings elicited constructive connections between elements, helped consolidating ideas and observations, and enabled external representation of inner mental models (Johnson-Laird, 1980) of posed problems.

In the next section we display illustrations of these phenomena, in the sub-domain of sequence processing. Sequences (and lists) involve a variety of fundamental tasks, from max computation, through searching and sorting, to string matching and more (e.g., Manber, 1986). The illustrations illuminate visual analyses that were offered by students. In the last section we discuss these findings of self-generated drawings, and argue the heuristic's relevance for teaching, displaying it to students, and encouraging them to employ it.

2. Visual Analysis Illustrations

This section shows three different scenarios of visual associations demonstrated by students. Each scenario involves a task that we posed during the third, advanced stage of our national activity towards the IOI. About 30 students reach this stage every year, and its early tasks require basic algorithmic knowledge, limited to 1D arrays, functions, recursion, searching and sorting, and complexity measures. Yet, the problem solving is not elementary. On the way to this stage, as well as in its early practices, students attempt algorithmic challenges whose solutions required capitalization on hidden, unfolded patterns. The tutors' challenge is to pose tasks that require little knowledge, but involve challenging problem solving for that stage. Diverse sequence processing tasks are suitable. The illustrations below include three such tasks.

The following task was one (the easier) of the six tasks of IOI 2005 in Poland. The phrasing here is shorter and less formal.

Mean Sequence. Given an increasing sequence of N positive integers, output the <u>number</u> of mean-sequences of length N+1 of the given sequence; where a *mean-sequence* is a sequence of non-decreasing integers, such that the mean of the i-th and the i+1-th integers is the value of the i-th element in the input sequence.

<u>Example</u>: For the input **4 9 16 21 23** the output should be **3**, as there are 3 mean-sequences (of 6-integers) that fulfill the required condition: **2 6 12 20 22 24** and **3 5 13 19 23 23** and **1 7 11 21 21 25**. Notice that every integer in the input se-

quence is the mean of two corresponding integers in each of the above sequences. Also notice, that a sequence starting with **0** may not be a valid mean-sequence, as the sequence will be: **0 8 10 22 20 26**, which is decreasing between the 4-th and the 5-th integers. For a similar reason, sequences starting with **-1**, or **4**, may also not be valid mean-sequences.

A first glace over the task yields several observations.

- Once an integer is chosen as the starting point, the rest of the sequence is determined.
- The output may not exceed by more than 1 the difference between the "closest" two integers in the input. (In the example of the problem statement, the two closest integers are **21 23**, thus the output may not exceed **3**.)
- A hasty conclusion from the previous observation may lead to the assertion that the value mentioned in the observation will be the output. Yet this may not always be the case; e.g, for the input sequence **4 6 10 20 22** the output should be **0** (and not **3**).

Following the first observation, problem solvers who follow a brute-force approach often seek a solution based on examining separately each sequence yielded from a starting point that enables the first two mean sequence points (the point before the first input value and its successive mean sequence point). This solution requires O(N×D) time and O(N) space, where D is the difference between the first two input points.

Other problem solvers regard the latter solution inefficient and seek a hidden pattern on which to capitalize. One such problem solver, whom we interviewed, conjectured that the starting points of legal sequences may be adjacent to one another, and form a range of consecutive integers. In order to examine his conjecture, he had a visual association. His idea was to start with a range of size D+1 and view each of the input integers as a *hinge* for *flipping* the initial range "forward", through the input points while possibly chopping it. He viewed the process of "flipping" the range, as a *"rod shrinking through the input values"*, and examined his idea with: **4 9 16 21 27 32 37**.

The visualization in Fig. 1 displays an elegant algorithmic idea, and informally also shows its justification. Although the figure only shows an example, and does not display formal notations, one may extract an argument of correctness from the figure. In a sense, it offers a hand-in-hand design + justification, and yields the following task solution.

> *The output is the size of the range remaining from "passing" the initial range of successive candidates through the input sequence, in a manner of flipping a "shrinking rod" through a series of hinges.*

The solution scheme enables an on-the-fly computation. All in all, the "flipping" process requires O(N) time and O(1) space. The computation visualized in Fig. 1 displays a scenario that represents the students' inner mental model of the task solution. The figure elegantly expresses an intuitive, elegant idea, without any notations.

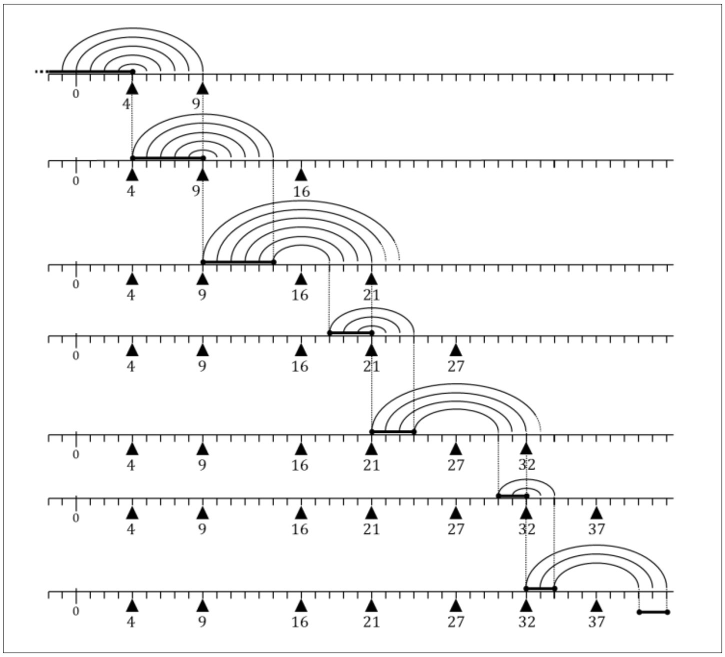

Fig. 1. Visual association of *flipping while chopping* the initial range of starting-points through the input sequence.

* * *

The next illustration involves visualization related to ordering. Ordering is a fundamental notion in algorithmics. While the natural tendency may be to simply examine integers, a figure that displays heights of values may help reaching beneficial associations.

Widest Inversion. Given a sequence of N different integers, output the widest inversion in the sequence, which is the maximal distance between two unordered integers, i.e., the larger of them is to the left of the smaller.

Example: For the input **4 3 8 9 1 6 10 7** the output should be **5**, due to the distance between the **8** and the **7**. Notice that there are many inversions in this sequence, e.g., the inversions of **4** and **1**, of **4** and **3**, of **3** and **1**, of **9** and **7**, and more.

(We first introduced this task in Ginat (2008), with diverse correct and incorrect solutions. Here, we examine it with respect to the relevance of drawing a figure.) One of the problem solvers of this task chose to examine it with several examples, for which she drew figures of bars. Below is one of the figures, for an input which is a permutation of the integers 1..11.

The student indicated that in the beginning she examined the brute-force solution, which computes the widest inversion for each input value as a left-end of an inversion. Obviously, the computation is inefficient, as it requires repeated "passes" over the input. In addition, the student mentioned that it was difficult for her to gain insight into the task

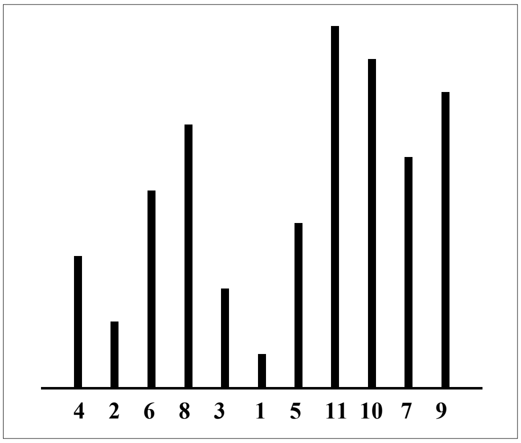

Fig. 2. Bars that represent the input values by height.

by repeatedly looking at series of integers. She sought a representation of the task that will be more illuminating. At first, she tried to draw lines between inversion ends in the list of integers, but decided that these lines do not clarify the "blurred" picture. Her next attempt was to draw a figure of bars, whose heights correspond to the input values, as displayed in Fig. 2. This representation helped her realize several observations, which "popped from the figure", regarding left-ends and right-ends of inversions.

- A value v may be in several (even many) inversions as a left-end. The only relevant value to consider as a right-end for v is the rightmost value u smaller than v. Thus, u must be smaller than <u>all</u> the values on its right.
- A similar observation is relevant for a value v as a right-end.
- The above observations imply that a value may be a **candidate for a right-end** of the widest inversion only if it is <u>smaller than all</u> the values on its right.
- Similarly, a value may be a **candidate for a left-end** of the widest inversion only if it is <u>larger than all</u> the values on its left.
- In the above figure, the values **1**, **5**, **7**, **9** may be candidates for a right-end of the widest inversion; and the values **4**, **6**, **8**, **11** may be candidates for a left-end.
- The two lists of candidates must be **in increasing order**, due to the properties of candidates. (Question: May a particular value be in both lists of candidates?)

Following the above observations, an O(N) time and space solution will first generate the two lists of candidates, and then capitalize on their increasing characteristic and "run" concurrently over these lists from left to right, while finding for each left-end candidate its corresponding right-end match.

All in all, the visual analysis enabled illuminating observations, that stemmed from looking at the bar heights and noticing the characteristics of left-ends, right-ends, and increasing candidate lists. In our experience, although these characteristics are simply phrased, problem solvers often struggle and do not recognize them while "going back and forth" over the sequence of integers. The drawn figure assisted in revealing them, and invoked constructive associations.

* * *

The last illustration involves visualization related to the recognition of a hidden sequence property on which to capitalize.

Circular Fence. Given a sequence of N positive integers, each describing the height of a column of a fence, in terms of its number of bricks, output the minimal number of blocks of bricks that should be transferred between adjacent columns in order to level the fence. The total number of bricks in the fence is **3N**, and the fence is **circular** (the N-th column is adjacent to the the 1-st column). A transfer of a block of bricks may involve any positive number of bricks, which will be moved either to the left or to the right of the column from which they are taken.

Example: For the input **5 3 6 2 1 1** the output should be **3**, due to a transfer of a block of 2 bricks from the column of **5** to the column of **1** cyclically on its left, and two more transfers of bricks – 3 bricks from **6** to **2**, and then 2 bricks from **2** to **1**.

One may first attempt the task for a **linear**, non-circular **fence**, with a left end and a right end. A single "pass" over the input from left to right will yield the desired output. The input will be accumulated, and for every i, $1 \leq i \leq N$, if the total number of bricks accumulated up to (including) the i-th column does not equal i×N, then the total number of block-transfers will be increased by 1. The latter operative description implies the following declarative observation, which relates to the places of no block transfers.

Let S be the number of autonomous sub-sequences. A sub-sequence is regarded ***autonomous*** *if its total num of bricks is 3 times its num of columns, and no prefix of it is has this property (of 3 times the num of columns). The output is N-S.*

Notice that if the fence in the example of the task statement is regarded linear, then the whole fence is one autonomous sequence. It cannot be partitioned into smaller autonomous parts. Since there are 6 columns in the fence, a total of 5 block transfers are required. Had the fence been circular, it could be partitioned into 3 autonomous sub-sequences, and only 3 transfers were required.

The reader may assess the correctness of the declarative observation above, for a linear fence. Notice that in the case of a linear fence there is only one partition possible, since there are given left end and right end. In the case of a circular fence there may be a variety of possibilities for partition. We need to find the best partition, i.e., the partition that yields as many autonomous sub-sequences as possible.

One way of seeking the best partition is to examine N different cases of a linear fence – the case of column-1 as the left end, then column-2 as the left end, then column-3, and so on. However, this solution is "expensive" time-wise. Its time complexity is $O(N^2)$.

Two students who sought a more efficient solution drew figures, which elicited their associations to a more efficient solution. One of them considered the following input of 11 columns **5 2 7 1 1 2 4 6 1 2 2** and examined the accumulated value of remaining bricks while levelling the fence on-the-fly, **2 1 5 3 1 0 1 4 2 1 0** (starting

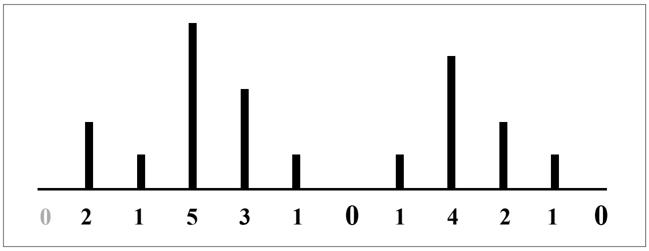

Fig. 3. The accumulated values while levelling the fence on-the-fly, starting at column-1.

from column-1). The leftmost **2** in the latter sequence indicates the number of bricks left after leveling column-1, the following **1** indicates the number of bricks left after leveling the first two columns, and so on. The last value in that sequence must be **0**, as the total number of bricks is 3N. Notice that for other inputs, the latter sequence could include negative values, which indicate deficits that require transfers of bricks to the left, from columns ahead. The student generated Fig. 3.

Fig. 3 shows that when the left-end is column-1, two **0**'s are generated, indicating 2 autonomous sub-sequences. When the student looked at the bars of the figure, he noticed that **1** appears 4 times. If he could "turn" the **1**'s to be **0**'s, he would have had 4 autonomous sub-sequences! He realized that this can be obtained by starting the levelling from the 3-rd position with **0** accumulated bricks, as shown in Fig. 4.

The student noticed that he could also start from the 6-th, 8-th, or 11-th column, and obtain the same result of 4 autonomous sub-sequences. He noticed that no matter where one starts the cycle, the sequence of accumulation-while-levelling numbers will be a shift of the original sequence in Fig. 3. The difference will only be in the location of the X-axis of the figure.

All in all, he realized that he may generate the accumulation sequence just once, starting from column-1; and then find the number of appearances of the value S that appears the maximal number of times. The output will be N-S. The computation of S requires O(NlogN) time, a considerable improvement of the initial $O(N^2)$ solution. Careful look at the initially drawn figure yielded an observation of the bar height that appeared a maximal number of times, and elicited the association of adjusting the X-axis, so that this height will be the new 0.

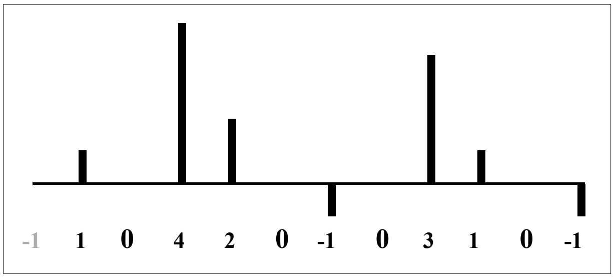

Fig. 4. The accumulated values while levelling the fence on-the-fly, starting at column-3.

3. Discussion

Visualization is a powerful method that can help one become more familiar with a given situation. It may give a "feel" of a posed problem and serve as a means for making explicit a detail or a picture in our mind. It may elicit constructive associations and connections between relevant elements. It may yield ideas. It may assist in reaching a plausible solution, as well as justifying it. A picture is sometimes worth a thousand words.

We have shown three different occasions of self-generated figures, that considerably assisted algorithmic problem solvers. Drawn figures elicited constructive associations and encapsulated mental models of posed problems. In the solution of the first task, the figure drawn by the problem solver helped him explicitly represent his mental model of the problem, with elements that involved a metaphor of rod flipping steps. His drawn visual scenario yielded not only the algorithmic solution, but also its intuitive justification, without formal mathematical notations.

The solution of the second task involved visual display of integers as bars of different heights. The visual display helped seeing ordering characteristics, especially with respect to left-end and right-end properties of inversions. The solution of the third task involved bars of integers as well, but the elicited associations were related to the recognition of multiple appearances of equal elements. The drawn figure yielded a constructive up/down shift of the X-axis of the represented values, and led to an elegant and efficient solution of sequence partitioning. The shift manipulations encapsulated not only the solution but also its justification.

Self-generated figures as a heuristic means in problem solving are studied in mathematics (e.g., Nunokawa, 2004), physics (e.g., Maries & Chandralekha, 2017), and additional domains, but not in computer science (to the best of our knowledge). In computer science, figures and visualizations mostly appear as teaching aids and computational-tracing means. Yet, self-generated figures may be most beneficial for problem solvers, as we have seen here. We believe that this problem solving heuristic should be elaborated, exemplified, and studied in the teaching of algorithmics, at all levels, including the Olympiad level. The enriching potential of figures may elicit and communicate associations and ideas informally, without words, but in a clear, elegant, and convincing manner.

References

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

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

Johnson-Laird, P.N. (1980). Mental models in cognitive science, *Cognitive Scienc*e, 4, 71–115.

Larkin, J.H., Simon, H.A. (1987). Why a diagram is (sometimes) worth ten thousand words, *Cognitive Science*, 11, 65–99.

Manber, U. (1986). *Introduction to Algorithms: A Creative Approach*. Addison Wesley.

Maries, A., Chandralekha, S. (2017). Do students benefit from drawing productive diagrams themselves while solving introductory physics problems? The case of two electrostatic problems. *European Journal of Physics*, 39, 1–18.

Nunokawa, K. (2004). Solvers' making of drawings in mathematical problem solving and their understanding of the problem situations. *International Journal of Mathematical Education in Science and Technology*, 35, 173–183.

Polya, G. (1945). *How to Solve it*. Princeton University Press.

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

Sorva, J., Karavirta, V., Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 15, 1–64.

Van Essen, G., Hamaker, C. (1990). Using self-generated drawings to solve arithmetic word problems. *The Journal of Educational Research*, 83, 301–312.

Van Meter, P., Garner, J. (2005). The promise and practice of self-generated drawing: literature review and synthesis. *Educational Psychology Reviews*, 17, 285–325.

D. Ginat – headed the Israel IOI project in 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.