Olympiads in Informatics, 2014, Vol. 8, 81–91

On Inductive Progress in 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. Induction is known, first and foremost, to mathematics and computer science students as an essential means for proving theorems. But induction is much more than that. Induction is also a core heuristic in the process of problem solving. In algorithmics, a problem solver should seek gradual observations of patterns of the problem at hand, and then capitalize on them in devising an algorithmic solution. In this paper we elaborate on the heuristic of inductive progress during algorithmic problem solving. We demonstrate its essential role with three different examples. Such an elaboration may enhance the awareness of tutors and students to components of the gradual process of problem solving.

Keywords: induction, problem solving, problem representation.

1. Introduction

In his book *How to Solve it* (1954), George Polya says "… Induction tries to find regularity and coherence behind observations … In mathematics and the physical sciences we may use observation and induction to discover general laws ..." (Polya, 1954, p. 117).

The general laws to which Polya refers are assertional, or declarative patterns of phenomena and regularities. The specification of assertional patterns is fundamental in mathematics and science, including computer science. Yet, in computer science there is an additional facet to general laws – the facet of formulating a general, operative, computational scheme.

In computer science the utilization of induction is two-fold:

1. For recognizing and proving assertional patterns.
2. For formulating general, algorithmic schemes, and justifying their correctness.

These two components are essential in the design of algorithms. During the design process, one has to first recognize patterns of the relationships between the input and the output of a given algorithmic task, and then capitalize on these patterns in combining, or composing suitable algorithmic schemes. Pattern recognition is an essential component of problem solving (Schoenfeld, 1992), and the composition of suitable algorithmic schemes is the basic means of algorithmic design (Linn and Clancy, 1992; Astrachan *et al*., 1998; Wing, 2006).

In this paper, we underline and illustrate the relevance of induction during the process of algorithm design. We display examples of different levels of difficulty, and illuminate different aspects of the utilization of induction – inductive design of a rather simple algorithmic scheme, inductive extension of perspective, and inductive development of a suitable problem representation. We display each of these aspects with a separate example in the following section. In each example, we present a gradual solution process, which progresses in inductive steps.

2. Inductive Progress

We display solution processes of three very different tasks. The first task involves an inductive process of algorithmic design, which starts with gradual recognition of assertional patterns and continues with capitalization on these patterns in the design of a linear algorithm. We developed this task in order to underline inductive progress, both in the design process and in the resulting algorithmic computation. The next two tasks appeared more than four decades ago in mathematics Olympiads. We display inductive solutions to these tasks, in which we elaborate on gradual extension of the perspective underlying the tasks' solutions. The second of these tasks is tied to binary representation, which is essential in algorithmics. Each task is displayed in a separate sub-section, which is titled according to its primary inductive aspect.

2.1. *Inductive Algorithmic Design*

We start with a relatively simple task. The solution process that we present below involves inductive recognition of task characteristics, combined with the gradual development of an algorithmic scheme. The justification of unfolded patterns and the resulting algorithm involves induction as well.

> **Fence Levelling.** A fence of tiles, made of N columns, should be levelled. The total number of bricks in the fence is $N \times h$, where h is the average height of a column. In one operation of brick-moving, one may transfer any number of bricks from one column to an adjacent column. Devise an algorithm whose input is h (the average height of a column), followed by a list of N positive integers, denoting the heights of the columns of the fence; and whose output is the minimal number of brick-moving operations needed for levelling the fence.
>
> For example, for the fence of five columns of heights: 1 4 11 3 6 (where h is 5), four operations of brick-moving are required (5 bricks from the 3rd column to the 2nd column, then 4 bricks from the 2nd to the 1st column, then 1 brick from the 3rd to the 4th column, and 1 brick from the 5th to the 4th column).

One way to approach the task is inductive. We may gradually consider various inputs, of increasing N, starting with $N = 2$, and continuing with various cases of $N = 3$, 4, 5, and 6. Notice that there is no need to simulate the brick-moving operations, but only output their amount.

The case of N = 2 is trivial, as there may be at most one brick-moving operation. The case of N = 3 may require up to two operations. One operation is required if exactly one of the end columns is of height *h*. Two operations are required if both ends are different from h. The characteristics of the various cases of N = 4 are similar in nature to those of N = 3, just slightly longer. One particular case to notice, of N = 4, is the case in which the first two columns may be levelled separately from the last two columns, for example – the case of 6 4 3 7.

We may learn a lot from the various cases of N = 3 and N = 4. First, it seems that the number of brick-moving operations may not exceed N – 1. And, it may be lower, if separate parts of the fence may be levelled independently (as in the case of 6 4 3 7). We may conjecture the following assertional patterns from the latter observations:

> *An N-column fence may not require more than N – 1 brick-moving operations.*
> *If the number of bricks in a sub-sequence of K columns is K × h, then these K columns may be levelled independently.*

We may vary further cases, of fences of 5 columns and 6 columns. One case may be: 7 5 3 4 5 6. Another may be: 7 3 5 4 5 6. We may notice that not only may a sub-sequence of K columns, with K × h bricks, be levelled independently, but also: if this sub-sequence may not be broken into smaller independently-levelled sub-sequences (where the average number of bricks in each is *h*), then the minimum number of operations required to level this sub-sequence is K – 1. This observation may be proved by induction on K. The proof also implies our first pattern above (of at most N – 1 operations), and yields the following illuminating assertional pattern:

> *Let S be the maximal number of independently-levelled sub-sequences into which the N-column fence may be divided. Then, the minimal number of brick-moving operations required for levelling the fence is N – S.*

Up to this point, we gradually unfolded assertional patterns. Now, we may devise a computational scheme. A natural idea that comes to mind is: to first recognize the smallest leftmost sub-sequence that may be levelled independently; then recognize the next such sub-sequence to its right; and so on. This idea calls for a linear scheme, which will progress inductively over the input:

> *Read the input column-by-column and count the maximal number of independently-levelled sub-sequences, by "collecting" these sub-sequences from left to right. The "collection" of each such sub-sequence ends once the average of the "collected" columns of the sub-sequence is exactly h.*

The correctness proof of the latter scheme may be formulated by induction on the number of columns read, using a suitable invariant of the single, rather simple loop of the computation. The intuitive justification is based on the notion that the computation finds the smallest left part that can be levelled, and then continues inductively.

All in all, the process of devising the algorithmic scheme involved: an initial stage of gradually unfolding assertional patterns, while inductively examining diverse inputs; and a second stage of devising an inductive "collection" of independently-levelled subsequences. The justification of the recognized patterns and the devised scheme is also based on induction. We leave it to the reader, as our focus here is less on formal proofs and more on inductive unfolding and specification of patterns. A slight modification of the task, into a circular fence, makes the task more challenging, as there is no specific (left or right) end from which the computation may start.

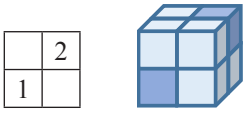
2.2. *Inductive Extension of Perspective*

The focus of the task in this section is inductive development of a suitable perspective, which encapsulates optimization. The task requires optimal placement of elements in a given structures. Computer science involves computations with diverse structures, such as matrices, trees, and graphs. The following task involves placing elements in a 3D matrix.

Rooks in a cube. Given a cube of size N (i.e., $N \times N \times N$ structure of $1 \times 1 \times 1$ unit cubes), place as few rooks as possible in the cube, so that all the N^3 unit cubes will be threatened (each by at least one rook). A rook threatens all the unit cubes that are in the X-axis, Y-axis, and Z-axis of its unit cube (including its own unit cube).

For example, for $N = 2$, two rooks will suffice – one in the bottom-left unit cube and one in the top-right unit cube. Each rook "covers" exactly 4 separate unit cubes.

In our presentation below, we display on a 2D paper, the rook placements in a 3D structure. In order to do so, we use a 2D square in which we indicate the Z-"level" (height) of each rook with an integer in the range 1..N. In addition, we use the terminology "bottom-left" and "top-right", for both 2D and 3D cases (where actually for 3D, we mean "bottom-left-front" and "top-right-back"). Thus, the 3D solution below, of the task statement example, for $N = 2$, is displayed with a 2D square as follows:



We advance, inductively to the case of $N = 3$. A natural attempt to extend the placement in the case of $N = 2$, is by placing rooks on the main diagonal, in different levels. However, this placement does not yield a "cover" of all the 27 unit cubes. The placement is illustrated in the left figure below. The right figure below shows unit cubes in the 3rd level that are not threatened. There are additional unit cubes, in the other levels, that are not threatened.

		3
	2	
1		

x		
	x	

Thus, we need to add rooks, and perhaps also change rook placements. An important characteristic that we may learn from the above placement is the following:

> *If there are K unit cubes in a level, which are not threatened by rooks of that level, then we need at least K rooks on the other levels, one in each "pillar" (Z-axis) of these unit cubes.*

Following this observation, we may first seek a placement of rooks in two levels, which will cover as many unit cubes as possible in these levels, and then add rooks in the third level. We may notice that two rooks that are placed diagonally in a level may cover 8 unite cubes in that level, as illustrated in the figure below (using the "+" sign):

+	+	
+	R	+
R	+	+

If we place two rooks in one diagonal of the bottom-left 2×2 square of the 1st level, and two additional rooks in the other diagonal of that 2×2 square, on the 2nd level, then we manage to cover 8 unit cubes in each level, plus the 2×2 bottom-left square of the 3rd level, as shown below:

·	·	
2	1	·
1	2	·

The only unit cubes not covered in the first two levels are the ones in the top-right. The unit cubes that are not yet covered in the 3rd level are those in the upper row and the right column. The covering of all these cubes may be achieved with one rook in the top-right unit cube of the 3rd level:

		3
2	1	
1	2	

So, we managed to cover the $3 \times 3 \times 3$ cube with 5 rooks. Combing the ideas used in the cases of $N = 2$ and $N = 3$, we may advance inductively to a $4 \times 4 \times 4$ cube, and notice that if we extend the top-right cube of $1 \times 1 \times 1$ into a top-right cube of $2 \times 2 \times 2$ (as we

just did with the bottom-left cube, in the transition from the case of $N = 2$ to the case of $N = 3$), then we may cover all the 64 unit cubes with 8 rooks, as follows:

		4	3
		3	4
2	1		
1	2		

It seems, from the latter cases that it may be beneficial to divide the view of the $N \times N \times N$ cube into two sub-cubes of sizes as close as possible – a bottom-left cube and an upper-right one. This view encapsulates a divide-and-conquer perspective. We may do so also in the case of $N = 5$, using 13 rooks, if we manage to cover the $3 \times 3 \times 3$ sub-cube in the bottom-left part of the following figure:

			5	4
			4	5
R	R	R		
R	R	R		
R	R	R		

Now we cannot use anymore only main diagonals (as in the simple case of $2 \times 2 \times 2$ structures), yet we may still place rooks diagonally in a systematic way, on "corresponding pairs" of diagonals, so that they will cover all the unit cubes in the first three levels, except for those threatened by the rooks at the levels 4 and 5:

			5	4
			4	5
3	2	1		
2	1	3		
1	3	2		

In extending the above inductively to the case of $N = 6$, we may cover a $6 \times 6 \times 6$ cube with 18 rooks, placed in two $3 \times 3 \times 3$ sub-structures, as follows:

			6	5	4
			5	4	6
			4	6	5
3	2	1			
2	1	3			
1	3	2			

At this stage we may generalize the rook placement, for the case of even N:

> *Place the rooks in two sub-cubes of the original $N \times N \times N$ cube, such that $N^2/4$ rooks will be placed in the bottom-left $(N/2)^2 \times (N/2)^2 \times (N/2)^2$ sub-cub, in diagonals of the sub-cub's levels (as in the figure above), and $N^2/4$ rooks will be placed in the top-right $(N/2)^2 \times (N/2)^2 \times (N/2)^2$ sub-cub, in the same manner.*

The case of an even N requires at least $N^2/2$ rooks. The proof of "minimality" is as follows: Let layer L be the layer with a minimal number of rooks, among the 3N layers of (the N) X-Y planes, (the N) X-Z planes and (the N) Y-Z planes. Let *L* be an X-Y plane, and let its rooks dominate r rows and c columns, where $r \geq c$. There are at least $(N-r) \times (N-c)$ rooks that dominate the one-unit cubes in *L* that are not dominated by the rooks in *L*. If we now change perspective, and look at the N layers of the X-Z planes, we notice that N-*r* of these layers contain $(N-r) \times (N-c)$ rooks; and in each of the remaining *r* layers there are at least *r* rooks (by the choice of *L*). The minimum of the expression $(N-r) \times (N-c) + r \times r$ is obtained with the value $N/2$ for both *r* and *c*. The case of an odd N is similar, and requires at least $(N^2+1)/2$ rooks.

All in all, the inductive solution process involved gradual illuminations, including: the relationship between the number of rooks in a particular layer and the number of additionally required rooks; the different ways of placing rooks diagonally in a square so that they will threaten the whole square; and the construction of two rather sparse cubes of rooks, of sizes $(N/2)^2 \times (N/2)^2 \times (N/2)^2$, each threatening three more, "non-rook" sub-cubes of the same dimension. The inductive process yielded a divide-and-conquer perspective of the whole structure of size $N \times N \times N$, as a "coarse" $2 \times 2 \times 2$ structure.

2.3. *Inductive Extension of a Representation*

The last task that we present is solved by recognizing a suitable representation. The selection of a suitable representation is a key element in problem solving in general, and algorithmic problem solving in particular. It illuminates task characteristics on which an elegant solution may be based.

> **Buckets.** Given three buckets of water, the goal is to empty one of the buckets, by repeated pouring of water between the buckets. At any given time, one may pour water from one bucket to another, in an amount that doubles the water in the bucket into which the water is poured. Thus, the bucket from which water is taken must contain at least as much water as the bucket into which it is poured. Each bucket is very large, and never overflows. Devise an algorithm whose input is three integers – A, B, C, denoting the water amounts in the three buckets; and whose output is the sequence of operations for emptying one of the buckets.
>
> For example, for the initial water amounts: 10 5 3, we may first pour 3 from the 2nd bucket to the 3rd, and obtain the amounts: 10 2 6 in the buckets, then pour 6 from the 1st to the 3rd and obtain 4 2 12, then pour 2 from the 3rd to the 2nd and obtain 4 4 10, and finally pour 4 from the 2nd to the 1st bucket and obtain: 8 0 10.

It may seem at first glance that perhaps there are initial cases for which there is no solution. Apparently this is not case.

An initial examination shows that the last operation is always conducted between two buckets with equal amounts of water. For gaining further insight, we turn to induction again. We may first solve the task for the case of the amount C = 1, then for C = 2, and then for larger values of C. In our description below, we use the terms A, B, and C, to denote bucket names as well as bucket amounts.

Thus, we first solve the task for the initial amounts: A B 1. We may notice that if we keep pouring water into bucket C (i.e., the 3rd bucket), without pouring water from this bucket, then the amounts of water in this bucket will always be powers of 2. The water amount in the bucket will grow from 1 to 2, to 4, to 8, etc. If we can transform the amount in one of the other two buckets to a power of 2 as well, then we may be able to obtain two buckets with equal amounts of a power of 2.

The notion of powers of 2 is an essential notion of problem representation. We know that each integer may be represented as a sum of powers of 2. Thus, we may pour water from one of the buckets A or B, in quantities which are powers of 2, and leave in that bucket an amount that is a power of 2. For example, let B = 57. Then we may represent B as the sum: $57 = 32 + 16 + 8 + 1$. If we pour from B to C first 1, then 8, and then 16, we will be left with 32 in B. So, we may start by pouring 1 from B to C, bringing the amount in C to 2. In order to pour 8 from B to C, we need to have 8 in C. That is, we need to pour 2, and then 4 into C, but not from B. At this point, bucket A will help us. We will pour 2, then 4, from A to C, and then continue to pour 8 and 16 from B. Both B and C will reach 32, and we will be able to empty B (or C).

Notice that we used A as a "complementing" source of water, whenever we needed to increase C with amounts that will not be taken from B. This will always be possible if A is not smaller than B initially. In addition, the representation, or perspective, of powers of 2 corresponds to binary representation. We may summarize the above scheme as follows:

> *For the case of A B 1, where $B \leq A$, we may empty B, by: pouring into C (which starts with 1) powers of 2 amounts from B, which correspond to the 1-bits in the binary representation of B, interleaved with pouring into C powers of 2 amounts from A, which correspond to the 0-bits of the binary representation of B.*

Progressing inductively, we may now examine the case in which C = 2 initially. If B is even initially, then the above scheme will lead to an empty B. But, if B is initially odd, we will be left with 1 in B in the end, as the beginning of the process of pouring into C starts by pouring of 2. However, if we are left in the end of this process with 1 in B, we may apply the scheme again, this time with B in the role of the smallest bucket.

We may now proceed to the case of A B 3. Following the idea underlying the case of A B 2, we may notice that if we keep on pouring water into C then it will keep growing in amounts that are powers of 2 multiplied by 3. That is, C's value will progress from 3×2^0 to 3×2^1, to 3×2^2, to 3×2^3, and so on. Thus, we may now look at the representation of B

as: $3 \times$ (*a sum of powers of 2*) + *remainder*, and apply the A B 1 scheme on A B 3. As with the case of C = 2, here too, we may be left with some remainder in B. But, this remainder may only be 0, 1, or 2. If it is 0, then we are done; if it is 1 or 2 then we will apply the scheme again, this time with B in the role of the (new) smallest bucket.

For example, let the initial state be: 20 17 3. The amount in B may be represented as: $17 = 3 \times (2^0 + 2^2) + 2$. Thus, we may pour 3×2^0 from B into C, then 3×2^1 from A into C, and finally 3×2^2 from B into C. This process will result with the remainder 2 left in B, which now becomes the smallest bucket. We may apply the scheme again, this time with a smaller C than in the previous iteration.

Following the analysis of A B 3 we may extend the utilization of the binary representation described above, and formulate the following scheme:

> *For the case of A B C, where A > B > C, represent B as: C × (a sum of powers of 2) + remainder. Pour water from A and B in accordance with B's above representation, and the powers-of-2 strategy (above) of the case of A B 1. If in the end of this process, the remainder left in B is not 0, then solve the task again, this time with B as the new C. Repeat this computational scheme until the remainder left in B is 0.*

All in all, the inductive process led us to an initial idea of capitalizing on integer representation as a sum of powers of 2, which was later extended to a representation of: an integer multiplied by a sum of powers of 2, plus a remainder. The suitable problem representations were the underlying key for the solution. Each water-pouring iteration of leaving a remainder in B is bounded by log (N) pouring operations, where N is the largest among A, B, and C; and there may less than N iterations, as the remainder left in B at the end of each iteration is always smaller than the remainder left in B of the previous iteration. Thus, the total number of pouring operations is bounded by N log (N).

3. Discussion

The notion of induction goes much beyond proofs and correctness argumentation. Its essential nature is related to the process of seeking a solution, and discovering general laws (Polya, 1945; Holland *et al*., 1986). Careful application of inductive search, by examining simple cases upon looking for hidden patterns, may be a key element in successful problem solving. Careful application of inductive design, upon devising an algorithmic scheme, may serve as a constructive means in algorithm design. Our objective in this paper was to underline and elaborate on these latter two elements.

The solution process of the first task in the previous section illustrated and underlined careful, gradual progress, which initially focused on the recognition of underlying patterns. Illuminated patterns then served as underlying characteristics in the design of a linear, greedy computation scheme. In a sense, this design process demonstrated Dijkstra's perspective of combining assertional and operational elements "hand-by-hand" in the design of algorithms (Dijkstra *et al.*, 1989).

The solution process of the second task unfolded gradual observations of the problem at hand. An initial picture was extended gradually, and involved accumulated notions, derived from extending the size of the problem. The final outcome yielded an elegant divide-and-conquer scheme.

The solution process of the third task involved inductive, flexible extension of an initial idea. The initial idea involved binary representation, or the representation of an integer as the sum of powers of 2. Yet, binary representation alone was insufficient for solving the general task. Inductive progression was applied, for more general cases than the basic one. A subtler solution scheme was developed, which combined binary representation with a multiple by an integer, plus a remainder.

The computation schemes reached during the three inductive processes – of greedy computation, divide-and-conquer, and extended binary representation – are essential in algorithm design (Cormen *et al.*, 1990). In examining and teaching our students, one of our objectives was to enhance their awareness of such outcomes and develop their transfer competence in "inductive" problem solving (Mayer and Wittrock, 1996).

We posed the three tasks at different stages of our national Olympiad activity. The first task was posed in one of our early national competitions, as the easier among four tasks. The second task was posed in a later stage, in order to examine students' observations and illuminations. The third was posed in an even later stage, to the better students. It is a challenging task if posed as is. It is much easier if presented in stages, according to the inductive process described above.

In our experience, students demonstrate different levels of competence with these tasks. Some offer erroneous solutions. Others offer partial solutions to some of the cases. And some solve the tasks, but with limited insight and without being able to elaborate on their observations. They phrase a solution, which they reached with relevant associations, but their "picture" of the task characteristic is vague.

A primary objective in examining our students and teaching them, during the Olympiad competitions and training, is to improve their problem solving competence, and strengthen their computational thinking perspective (Wing, 2006). An ordered inductive progress like the ones presented here may assist is attaining this objective. It illustrates, in an apprenticeship manner, an essential approach that may increase students' awareness of the process of problem solving, and enhance the link between assertional patterns, problem representation, and algorithmic schemes.

References

Astrachan, O., Berry, G., Cox, L., Mitchener, G. (1998). Design patterns: an essential component of CS curricula. In: *Proc of the 29-th SIGCSE Technical Symposium on CS Education*. ACM, 153–160.

Clancy M.J., Linn M.C. (1992). The case for case studies of programming problems. *Communications of the ACM*, 35(3), 121–132.

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.

Holland, J.H., Holyoak, K.J., Nisbett, R.E., Thagard, R.P. (1986). *Induction*. MIT Press.

Mayer, R.E., Wittrock, M.C. (1996). Problem solving transfer. In: Berliner, D.C., Calfee, R.C. (Eds.), *Handbook of Educational Psychology*. 47–62.

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

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.

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.

D. Ginat – heads the Israel IOI project since 1997. He is the head of the Computer Science Group in the Science Education Department at Tel-Aviv University. His PhD is in the Computer Science domains of distributed algorithms and amortized analysis. His current research is in Computer Science and Mathematics Education, focusing on cognitive aspects of algorithmic thinking and learning from mistakes.