Olympiads in Informatics, 2008, Vol. 2, 48–63
© 2008 *Institute of Mathematics and Informatics, Vilnius*

Teaching Algorithmics for Informatics Olympiads: The French Method

Arthur CHARGUÉRAUD, Mathias HIRON

France-IOI
5, Villa Deloder, 75013 Paris, France
e-mail: arthur.chargueraud@gmail.com, mathias.hiron@gmail.com

Abstract. This paper describes a training curriculum which combines discovery learning and instructional teaching, to develop both problem solving skills and knowledge of classic algorithms. It consists of multiple series of exercises specifically designed to teach students to solve problems on their own or with the help of automated hints and intermediate problems. Each exercise is followed by a detailed solution similar to a lecture, and synthesis documents are presented at the end of every series. The paper then presents a structured problem solving method that is taught to the students throughout this curriculum, and that students can apply to organize their thoughts and find algorithms.

Key words: teaching algorithmics, problem solving techniques, guided discovery learning.

1. Introduction

France-IOI is the organization in charge of selecting and training the French team to the International Olympiads in Informatics (IOI). As main coaches of the French team since its early days in 1997, our main focus is to develop teaching materials and improve training methods. Since there are no computer science curriculums in French high schools, the students who come to us are usually self-taught programmers with almost no experience in the field of algorithmics.

All the teaching materials that have been produced are made available to the public on the training website (`http://www.france-ioi.org`), where the students can study all year long. Intensive live training sessions are held several times a year for the top students. The long-term objective is to reach a maximal number of students, so the online training follows the same principles as the live training sessions. The aim of this paper is to describe these principles. Section 2 gives an overview of the training curriculum. Section 3 then presents the structure of the training website. Finally, Section 4 describes the problem solving method that has been developed along the years and is now taught throughout the website.

2. Philosophy of the Training Curriculum

In this section, we explain how the training curriculum is designed around series of problems for the teaching of classic algorithms and data structures as opposed to providing lectures. We then present the benefits of this approach.

2.1. *Introduction to the Structure of the Training*

The training curriculum we present differs from those that can be found in many algorithmic courses. Most curriculums follow a common pattern:

1. *Provide basic knowledge*. Teach algorithms and data-structure through lectures, providing pseudo-code and complexity analysis.
2. *Show examples*. Illustrate how to apply this knowledge to practical situations.
3. *Give exercises*. Solidify what was taught, and try to develop solving skills.

This approach counts mostly on a student's innate abilities when faced with new, more difficult problems. Our thesis is that *improving skills to solve new, difficult problems is what really matters in algorithmics*. Therefore the true aim of a teaching curriculum should be to improve those skills. Providing the student with knowledge and experience is necessary, but should not be the first priority.

The interactive training is structured around sets of exercises with each set focusing on one particular area of the field. Within a set, each exercise is a step towards the discovery of a new algorithm or data-structure. To solve an exercise the students submit their source-code and the server automatically evaluates the code, in a fashion similar to grading systems that are used in many contests. If the students fail to solve an exercise, they may ask for automated hints or new intermediate problems, which in turn ease the progression. Once the students succeed, they get access to a detailed solution that describes the expected algorithm and presents a reasonable path to its discovery. At the end of each sequence of exercises, a synthesis of what has been learned is provided.

Overall the curriculum can be seen as a guided step-by-step discovery completed with instructional material as opposed to more standard curriculums which are typically based on instructional material and very little discovery learning.

2.2. *Teaching the Problem Solving Method*

While training students along the years, it was tried as much as possible to give reusable hints, i.e., hints valuable not only to the current problem but for others as well. Reusable hints were then collected and sorted in an organized document. In recent years, the authors have improved this document by carefully analyzing the way they come up with ideas for solving difficult problems. As a result, a *general problem solving method* was developed. This method is not a magical recipe that can be applied to solve any algorithmic problem, but someone who is sufficiently trained to apply this method will be able to solve more difficult problems than he/she could have solved without it.

As the document describing the method appears very abstract at first, asking one to *“apply the method”* is not effective. Instead, the teaching of the method is spread throughout the training curriculum in three ways: [1]

1. The hints given to students when they need help are (in most cases) hints that could have been obtained by applying the method.
2. Each detailed solution is presented in a way that shows how applying steps of the method leads to the expected algorithm.
3. Advanced students are provided not only with the reference document describing the problem solving method, but also with versions specialized to particular areas.

To teach this method, we follow an inductive approach rather that a deductive one, since it is extremely hard to explain without many examples to rely upon.

2.3. *Advantages over Traditional Curriculums*

To improve skills for solving unknown hard problems, it is necessary to practice solving such hard problems. So in a way, explaining an algorithm in a lecture can be seen as wasting an opportunity to train that skill. It gives away the solution to a problem before giving the students a chance to come up with a few good ideas on their own. By finding some, if not all of the ideas behind an algorithm by themselves, students not only learn that algorithm but also improve their problem solving skills. Also, by making these ideas their own, they are more likely to adapt them to future problems.

Once students have spent time thinking about a problem and have identified its difficulties, they are in a much better position to understand and appreciate the interest of the reference solution that is then presented to them. More importantly, students can compare the process they have followed to find the solution against the process that is described in the lecture. They can then update their strategies for the next exercises.

Also, discovering by oneself the principles of breadth-first search, binary trees, and dynamic programming generates a great deal of self-satisfaction. Even though the previous problems and reference solutions made it much easier by putting the students in the best conditions, they get a sense of achievement for having come up with their own ideas, gain confidence about their own skills and a strong motivation to go further.

Discovery learning is often proposed as an alternative to the more traditional instructional teaching, but has been strongly criticised recently (Kirschner *et al.*, 2006). Our purpose is not to advocate the *pure* discovery learning that these papers focus on and criticize extensively. The curriculum we present is indeed based on *guided* discovery learning, and is completed with extensive and carefully designed instructional teaching. With a structure designed to ensure that each step is not too hard for students, and which reinforces what is learned after each exercise, the risks associated with pure discovery learning, such as confusion, loss of confidence and other counter-productive effects are avoided.

[1]Note: the training material is constantly evolving, and what is presented here corresponds to the authors’ current vision. At the time of writing, many existing elements of the website still need to be updated to fit this description completely.

To summarize, we teach problem solving techniques by relying in the first place on the good aspects of discovery learning, and then consolidating the insight acquired by students through instructional teaching. Teaching of pure knowledge, which is a secondary goal, also follows the same process. This is the complete opposite of standard curriculums which start by teaching knowledge in instructional style, and then leave students to develop solving techniques mainly on their own.

3. Structure of the Training Curriculum

In this section, we give the global structure of the training curriculum, and describe each of its components in more detail.

3.1. *Overview of the Structure*

The training website aims at being an entirely self-contained course in algorithmics. The curriculum is divided into three levels of difficulty: beginner, intermediate, and advanced. The only requirement for the "beginner" level is to have some experience of programming in one of the languages supported by the server. At the other end, the "advanced" section contains techniques and problems corresponding to the level of the IOI. Note that the training website also includes instructional programming courses and exercises for the C and OCaml languages.

Each of the three levels contains several series of exercises of the following kinds:

1. *Step-by-step discovery.* A sequence of problems designed to learn about a set of classical algorithms on one particular topic. Each problem comes with a detailed solution, as well as optional automated hints and intermediate exercises.
2. *Application problems.* A non-ordered set of problems of various topics involving variations on algorithms and techniques previously learned through step-by-step discovery.
3. *Rehearsal exercises*. Students are asked to write their best implementation of each standard algorithm studied in a given domain. Reference implementations are then provided.
4. *Practice contests*. Students are trained to deal with limited time and get them used to zero-tolerance on bugs.

When students complete a series of exercises, they are granted access to a synthesis document that recapitulates all the encountered concepts in a well-structured presentation. Once students complete an entire level, they are provided with various elements of the problem solving method and a document summarizing techniques that have been implicitly presented throughout the reference solutions.

Little by little, these documents constitute a sort of reference manual that a student can look back to on a regular basis.

3.2. *Hints and Intermediate Problems*

To improve students' problem solving skills, they are trained on problems that are hard enough to make them think seriously and apply problem solving strategies, but feasible enough to have a good chance at finding a solution within a reasonable amount of time without the risk of losing motivation. Clearly there is no hope that a single set of problems will suit all students and their disparate levels.

The issue is addressed in the following way: a set of "main problems" that are non-trivial even for the best students is given. For each problem, intermediate problems and/or hints are given to students who cannot find the solution after a certain amount of time. They are carefully selected to not only help the students reach the solution, but to also make them realize they could have thought of the solution by themselves with the right strategy, that they will then apply to solve the following problems.

From a practical point of view, a hint consists of information that is added at the end of a task and may be remarks on properties of the task, a simplified formulation of the problem, a well chosen graphical representation of an example, or a suggestion about the kind of algorithm to look into. The last hints provided give the main ideas of the solution and make sure students do not get stuck and give up on the task.

Sometimes intermediate problems are provided instead of hints. They are typically simplified versions of the original problem or versions with weaker constraints for which the students can submit a program and obtain a detailed solution. This gives them a strong basis to attack the original problem.

One can compare each series of exercises to a climbing wall that leads to the discovery of the algorithms in a particular area: when a hold is too high to be taken the students are helped out by being provided with one or more extra intermediary holds. If this is not enough, the student has the option to contact a coach. Overall the structure adapts itself to the level of the students by providing steps of difficulty corresponding to their levels, which help them to learn as much as possible.

3.3. *Example: a Series of Problems Introducing Graph Algorithms*

In this section, we illustrate the notion of step-by-step discovery with an example series of problems that introduces students to very basic graph algorithms. This series appears roughly in the middle of the beginner level of the training curriculum. Its only prerequisites are the completion of a set of problems that covers basic data structures (introducing stacks and queues), and another that introduces recursive programming.

The tasks from this first graph series all take a maze as input: a two-dimensional grid in which each cell is either an empty square or a wall. The locations of the entrance and exit are fixed (respectively at the left-top and bottom-right corners). The following list describes these tasks and the list of hints and/or intermediate problems that the website provides on demand.

1. Read a maze and print the number of empty squares that have respectively 0, 1, 2, 3, and 4 empty adjacent squares. The purpose of this problem is to introduce the

notions of neighbors and degree, and to show how to manipulate these notions in the code in an elegant way.

   a. Hint (Intermediate problem): “read a maze and print the number of empty squares”. The aim of this new problem is to check that input data is read correctly.

2. Read a 10×10 maze and print the number of different ways to go from the entrance to the exit without walking through the same cell twice within a single path.
   a. Hint: the following question is asked: “what question can you ask on each of the neighbors of the first cell in order to compute the total number?”
   b. Hint: the answer to the previous hint is given, and insists on the fact that there is a set of cells one cannot use in the rest of the path.
   c. Hint: the main idea of a recursive function is given, that maintains the current set of cells that cannot be used for the rest of the path.
3. On a maze of up to 1000×1000 cells, give the total number of cells that can be reached from the entrance.
   a. Intermediate problem: same exercise with a 10×10 maze. This problem comes with a hint of its own, telling how to reuse the idea from the previous problem by marking every visited cell. It also comes with a detailed solution of a working exponential algorithm.
   b. Hint: the following suggestion is provided: “try to apply the algorithm given in the previous hint by hand on an example and find out how to reduce the amount of work involved”.
   c. Hint: running of the algorithm is demonstrated with an animation that clearly shows sequences of steps done several times.
4. Given a 10×10 maze, find the longest path going from the entrance to the exit without traversing the same square twice. Print the path as a sequence of letters (‘W’, ‘N’, ‘E’, ‘S’). When there is more than one such path print the one that is first in alphabetical order.
   a. Intermediate problem: any longest path is accepted as an output.
      i. Intermediate problem: only output the length of the path
      ii. Hint: the following question is asked: “in the solution provided for the previous intermediate problem, when can you say that the square corresponding to the current step of the recursive exploration is part of the longest path found so far?”
      iii. Hint: the answer to the previous question is provided, and demonstrates a way to record the steps of the longest path.
   b. Hint: the following question is asked: “in the solution given for the first intermediate problem, in which order should you try to explore the neighbors?”

This structure allows students to manipulate algorithms such as exhaustive search, depth first search, and printing the path corresponding to an exhaustive search. Strong students can do this quickly by solving 4 problems while weaker students can go at a slower pace with a total of 8 problems.

3.4. *Detailed Solutions*

Access to the detailed solution of each problem is only given once the problem has been solved by the student and checked by the server. The students are encouraged to read these analysis documents carefully. Indeed, having solved the problem successfully does not necessarily mean one has understood everything about the algorithm he/she discovered.

This document contains complete instructional material on the algorithm studied that consists of the following elements:

1. A step by step description of a thought process that leads to the key ideas of the solution. The intent is to have the students realize that they could have applied this process entirely on their own.
2. A well chosen graphical representation of the problem. Such diagrams help the students to see the interesting properties of the problem and can be reused later when working on similar problems.
3. A clear presentation of the algorithm. First through a few sentences giving the big picture and then through precise pseudo-code. The rationale of why the solution works is given, but no formal proof.
4. A complexity analysis of the solution.
5. Implementations of the reference solution, in C++ and OCaml that are as elegant and simple as possible.

These elements are given not only for the expected solution of the problem, but also for valid alternative solutions. Solutions usually start with a description of the principles and complexity analysis of algorithms that are correct but not sufficiently efficient, since they are typically intermediate steps in the thought process. Counter-examples to frequently proposed incorrect algorithms are sometimes presented to explain why those algorithms cannot work. This is done in a way that teaches the students how to create their own counter-examples.

This combination of elements gives the students a solid basis of knowledge and techniques on which they can then rely to solve other problems.

3.5. *Application Problems*

When working on problems from the step-by-step discovery series, students are in an environment which is particularly prone to bringing about new ideas. For instance, in the middle of the graph algorithms series, the students expect each problem to be a new graph problem and moreover expect its solution to be partially based on the elements discovered in the previous exercise. While such series of exercises are great to learn about graph algorithms, they do not train recognition of graph problems among a set of random tasks.

So once the basic knowledge of each field is acquired it is important to train students to solve tasks outside of an environment that suggests a given type of solution. Therefore each level ends with sets of problems of various kinds that cover most of the elements encountered throughout the discovery series. These tasks train the students in three ways.

First, they train to recognize the nature of a problem without any obvious clues. Second, they train to apply the knowledge and solving techniques acquired to variations of standard algorithms. And third, more difficult problems would typically require a combination of algorithms coming from different fields.

These application problems also come with hints and detailed analysis that insist on the different steps needed to find the solution. Also, these solutions often describe some useful programming techniques which help making the code shorter and cleaner, thus less error-prone.

4. Introduction to the Problem Solving Method

This section gives an introduction to the problem solving method that we have developed along the years and now teach throughout our website. We do not attempt to describe the whole method in details since this would be way beyond the scope of this paper, but instead we try to convey its main principles. To that end, we first explain how this method has been obtained, then illustrate its working on three particular steps, and finally give an overview of the other steps that it involves.

4.1. *Origins of the Method*

Throughout the years spent training the students on a regular basis through intensive live sessions, sets of problems on the website or frequent email interactions, there have been numerous occasions to look for the best advice to help students solve a given problem without giving them the solution itself, or even part of the solution. On each of these occasions, it could be determined which advice were the most successful.

It became apparent that some types of advice were very efficient over a variety of problems. Little by little a collection of techniques was synthesized which led to a full method for solving algorithmic problems.

It was then observed that on various occasions including IOI competition rounds, students applying the new method in a systematic manner would find the right ideas for difficult tasks more often than students who only counted on their intuition. Since then, improving this method and the way it is taught throughout the training program have been the top priorities. Every time coaches or contestants solve a hard problem, time is spent analyzing what helped to get the right idea. This is then taken into account to update the method when appropriate.

4.2. *Dimensions of the Problem*

This section describes a process that not only is a key to the application of several solving techniques, but which also helps to get a clear understanding of a task.

The objective is to build an exhaustive list of the dimensions of the problem. The word “dimension” is to be taken in its mathematical sense; informally it corresponds to everything in the problem that can take multiple values (or could if it was not settled to a specific value). Dimensions should be ordered by type: they can be the values from the input data, from the output data, or intermediate values that are implicitly necessary to be manipulated in order to get the output. Beginners are given a technique to ensure no dimension has been missed (due to lack of space, this is not described here). Trained students do this step as they are reading the problem statement.

For each dimension in that list, the range of possible values should be indicated. The constraints for the input dimensions are usually given in the task. For other dimensions, some calculations (or approximations) might be needed.

To illustrate the process, consider the following problem:

> "*You are given the coordinates* (x_1, y_1) *and* (x_2, y_2) *(*$0 \leqslant x_1, x_2, y_1, y_2 \leqslant 100\,000$*) of two diagonally opposite corners of each of* N *(*$0 \leqslant N \leqslant 10\,000$*) rectangles. Write a program that computes the maximum number of rectangles that have the same area.*"

Table 1 describes the dimensions for this problem. Notice that there is more to consider than just two dimensions (x, y) of the plan since x_1 and x_2 as well as y_1 and y_2, can be changed independently and have different meanings. Note the potential overflow for the surface of rectangles which means 64 bits integers will be needed at some point in the implementation.

Filling the second column may not always be obvious, and one has to be careful not to blindly write down what is given in the problem statement. In particular, the range of values for a dimension can be significantly reduced when it is impossible to reach all the values from the range given in the task.

Having this table of clearly listed dimensions at the top of one's paper is very useful both while looking for ideas and during the implementation phase. The next two sections will show how this list can be used as a first step to some very effective techniques.

Table 1

Dimensions of a problem

Dimension	Range of values
Input dimensions	
id of a rectangle	[0..9999]
x_1	[0..100,000]
y_1	[0..100,000]
x_2	[0..100,000]
y_2	[0..100,000]
Output dimensions	
Number of rectangles of a given surface	[1..10,000]
Implicit dimensions	
Width of a rectangle	[0..100,000]
Height of a rectangle	[0..100,000]
Surface of a rectangle	[0..10^{10}] (overflow!)

4.3. *The Power of Simplification*

The most recurring advice given to the students when they are stuck on a hard problem can be synthesized in the following way:

> "*Simplify the problem, and try to solve each simplified version as if it was a new, independent problem.*"

This simple advice is undoubtedly the most powerful problem solving technique in the method. It is effective in that its sole application suffices to resolve many hard problems and in that there is a very precise recipe to generate simplified versions of a given problem.

The idea that simplifying a task may help to find the solution of a problem is not new. It is described for instance by Ginat (2002), and mentioned in the analysis of various tasks such as "Mobiles" from IOI 2001 (Nummenmaa *et al.*, 2001). What is presented here is a technique to look for every useful simplification of any task, that students are asked to apply systematically.

There can be many different ways to simplify a given problem and not all of them give as much insight into the complete problem. Moreover, some useful simplifications may not come to mind immediately. So what is needed is a way to come up with the most useful simplifications in a quick manner. The following recipe can be used to go through every simplified version of a problem.

1. For each dimension of the task (see Subsection 4.2) try to simplify the problem by either: (a) removing the dimension, (b) setting the value for this dimension to a constant, or (c) reducing the range of possible values for that dimension.
2. Among the resulting simplifications, rule out those which clearly lead to a non-interesting problem. Then, for the sake of efficiency, sort the remaining simplified problems according to their interest – this is to be guessed by experience.
3. For each simplification considered, restate the corresponding problem as clearly as possible and try to solve it as if it were a completely independent problem. This may involve applying the problem solving method recursively, including a further simplification attempt.
4. Try to combine the solutions of the different problems in order to get some ideas for the complete problem. (There are some specific techniques to help with this step).

Notice that there is no need to simplify more than one dimension at a time since the simplification recipe is called recursively when necessary.

The fundamental idea behind this technique is that although solving a simplified version of the problem is almost always easier than solving the whole problem, it is often very helpful. Indeed, the complete solution needs to work at least on instances of the simplified problem, so any algorithm or clever observation required in the simplified solution will most likely be a part of the complete solution.

Finding the solution to a simplified version has an effect akin to getting a very big hint. Given that the simplification technique is so useful to produce hints and that it is so easy to apply (at least with some experience) it is tried to have the students apply this technique as a reflex.

Going from the solution(s) of one or more simplified versions of a problem to a solution for that problem can be difficult, and a separate document provides with techniques that make it easier. The content of that document is out of the scope of this paper.

As an example, consider the task "A Strip of Land" (France-IOI, 1999) where, given a 2-dimensional array of altitudes, the goal is to find the rectangle of maximal surface, such that the difference between the lowest and highest altitude within that rectangle is stays below a given value.

Examples of simplified versions of this task that are obtained by applying the described method are:

1. Remove the y dimension; the task is then, given a sequence of integers, to find the largest interval of positions such that the difference between the minimal and the maximal value from that interval is less than a given bound.
2. Reduce the altitude dimension to only 2 values, 0 and 1; the task is then to find the largest rectangle containing only zeros.
3. Apply both simplifications; the task is then to find the maximal number of consecutive 0s in a sequence of 0s and 1s.

Each of these simplified versions appears much easier to solve than the original task, and a solution to the original problem can be obtained by combining ideas involved in the solutions of these simplified problems.

4.4. *Graphical Representations: Changing Points of View*

Drawing different examples on a piece of paper and looking at them can be a very good way to get ideas. Drawing helps to show many properties clearly and all the elements that are laid on the paper are elements that do not need to be "maintained" in the limited short term memory. That memory can then be used to manipulate ideas or imagine other elements moving on top of the drawing. For a given problem, however, some drawings are much more helpful than others.

When students are working on hard problems they typically write some notes about the problem on a piece of paper and draw a few instances of the problem as well as the corresponding answers. Then they stare at their paper for a while thinking hard about how to solve the problem. When a student do not seem to be moving forward on a problem, coaches look at how he drew his examples and often think "no wonder he can't find the solution, with such a drawing no one could".

The students' drawings represent examples that are too small for anything interesting to appear and they are asked to try with larger ones. Other times the problem lies with their choice of a graphic representation for the problem. Students tend to draw things in a given way and often stick to that representation throughout their whole thinking process. They sometimes make a similar drawing multiple times hoping that new ideas will come. Their mistake is to forget that there can be several different ways to draw the same example and the first one that comes up is seldom the one that does the best job at bringing ideas.

To illustrate the point, consider the following task:

"*Given a sequence of positive integers, find an interval of positions within that sequence to maximize the product of the size of the interval and the value of the smallest integer within that interval of positions.*"

Faced with this task most students will naturally write a few sequences of integers on their paper and compute the product for various possible segments.

However, there is a way to represent such sequences that is much more expressive: draw each number of the sequence as a vertical bar whose height corresponds to the value of that number. Segments that are potential answers for a given instance of that problem can then be represented as rectangles whose height is the minimum height among all bars along its length. The answer then corresponds to the rectangle with the largest area. This new representation displays some of the properties of the problem in a much clearer way, and makes it easier to solve the task (Fig. 1).

The most important advice that is given regarding graphical representations is the following:

"*Don't stick to your first idea of a graphical representation and try different ways to draw examples*".

Applying this advice in an efficient way is not as easy as it seems. Students may quickly think about several representations, but often miss the most interesting ones. Students are taught to apply a simple technique to enumerate the most interesting graphical representations and select the best ones.

The main idea behind that technique is to observe the following fact: there are only two dimensions on a piece of paper. So only two dimensions of the problem can be presented in a very clear ordered way, where values can be compared in an instant, by looking at their relative positions. Selecting the two dimensions of the problem that will be mapped to the x and y axis of the sheet of paper is an essential part in selecting the right representation. The following steps summarize this process:

1. Among all the dimensions of the task (see Subsection 4.2) identify the most important ones starting with the dimensions that correspond to values that need to be compared easily. Consider grouping the dimensions that are compatible as one (x_1 and x_2 may both be represented on the same axis). This first step is used to optimize the chances of finding the best representation quickly.
2. For every possible pair among these dimensions consider a graphical representation that maps each dimension of the pair to the x and y axis of the paper and find a reasonable way to represent the others in the resulting grid.

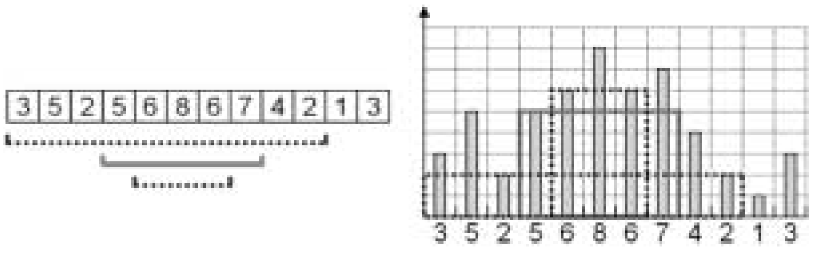

Fig. 1. Two graphic representations for the same problem.

3. Among these possible representations, apply each of them on a simple example. By comparing the results, it is usually clear which representation is the most helpful to visualize the problem and bring ideas.

In most cases, there are only a few pairs of dimensions to enumerate so this technique can help to quickly find the best representations. When the problem is more complex and the number of dimensions is higher, there can be quite a few pairs to enumerate. This may seem like a long process, but with some experience and a couple of rules of thumb to discard the least interesting representations, drawing an example for each potentially interesting pair can be done fairly quickly.

Of course, selecting these two dimensions is not enough to get a good representation and much advice can be given depending on the type of problem. In most cases though, what should be represented on the paper to get ideas is mostly the same: the components of the example, and the answer for that example. Students often do fine naturally at selecting which elements to draw, but may need some advice on what example to draw, and how to draw it.

Once the students have a nice and carefully drawn graphic representation of a good example in front of them it is observed that they are much more likely to get the right idea. We often observe students spend a lot of time working on a problem only to see the solution "appear" to them as soon as they are asked to use a graphic representation that can be obtained by the aforementioned technique. All that was needed for them to find the solution was to change their point of view on the problem. Forcing oneself to try out different graphical representations is a very effective way to try new points of view.

4.5. *General Structure*

So far, three particular elements of the problem solving method have been presented. In this section, we give an overview of its structure by briefly describing each of the main steps.

The method can be divided in two parts. The purpose of the first part is to help to bring a better understanding of the problem and to bring solution ideas. The second part contains steps to apply on each idea that comes up during the first part. Unless the task is really easy and the solution is obvious, students should at least go through the first four steps of Part 1 to make sure they have a clear understanding of the problem before working on a given idea. After that, they may jump to the second part at any time and come back later to Part 1 if necessary.

The following steps should be attempted in the order they are listed. It might however be necessary to come back to a previous step at some point to spend more time on it. For example, it is often useful to come back to Step 4 and generate some new examples.

1. Restate the problem (or sub-problem) in the simplest way, discarding superficial elements of the story, to get a clear idea of its essence. A good formulation usually involves two sentences: the first one describes the data and the second one the question that must be answered on this data. Someone who has not seen the full task should be able to start looking for a solution after reading this description only.

2. Write down a list of all the dimensions involved in the task and the corresponding constraints, as described in Subsection 4.2. This further helps to get a clear idea of the task and is necessary both to find a good graphical representation (Step 3) and to simplify the problem (Step 6).
3. Find the best graphical representations for the task by applying the technique described in Subsection 4.4. This can make working on examples (Step 4) much more effective at bringing ideas.
4. Generate different kinds of examples and solve them carefully and entirely by hand. These examples should be large enough so that the answer is not obvious and needs some effort to be determined. This step has many benefits. First, it helps to get all the details of the problem clearly in mind. Also since the brain is naturally lazy, ways to avoid doing too much work will often come automatically, which can lead to good ideas. These examples will be used again later to test new ideas and check the final implementation, so writing them rigorously is rarely a waste of time.
5. Look for an algorithm that gives a correct answer to the problem, without worrying about efficiency, and describe it clearly. The purpose is to separate the concern of correctness from the concern of efficiency and this often helps to clarify the recursion involved in the task. In some cases, writing it as pseudo-code may be a good idea. Depending on the form of this "naive" solution, different specific methods can then be applied to transform it into an efficient solution. Note that it is sometimes useful to implement such a brute-force solution, to generate data from some examples that one can then analyze to look for specific properties.
6. Simplify the problem and try to solve each simplified version as if it were a new, independent problem, as explained in Subsection 4.3. Then try to combine the different ideas into a solution for the original problem.
7. Try to see the task from different point of views by listing standard algorithms and wondering how they could be applied. One may ask questions like "can the problem be seen as a graph problem?", "could it be solved with a shortest path algorithm?", or "how could a sweep-line technique be applied?" and so on. Even in the case where the solution is not a standard algorithm, changing point of view on the task in this way may help to bring new, original ideas.

For each promising idea obtained during this first part the students are asked to go through the following steps. Note that Step 5 should be applied on any "reasonable" idea that is found to be incorrect.

1. Describe the solution in a couple of sentences. The objective is to make it clear enough that anyone who knows the task and is experienced with algorithmics can understand the idea. Depending on the type of the algorithm, the method provides standard ways to describe the solution.
2. Find a good graphical representation of the algorithm. In a similar fashion to what we described in Subsection 4.4, there is often one or more good ways to represent the dynamics of an algorithm graphically. This can help to understand why it works and brings attention to special cases or possible optimizations. Again, the method provides standard representations for certain classes of algorithms.

3. Using this graphical representation, try to execute the algorithm by hand on a couple of examples. This gives a better feeling of how it works and may bring up elements to think about during the implementation.
4. Look for counter-examples to this algorithm, as if it were a competitor's solution that you want to prove wrong (i.e., forget that you really hope it works). If a counter-example can be found, then it often offers an excellent example to use when looking for better ideas. It is also useful to explicitly state why the idea does not work.
5. After spending some time looking for counter-examples without finding any, it usually becomes clear why there is no chance of finding a counter-example. In other words, it becomes clear why the algorithm is correct. While it is too hard and too long to carry out a formal demonstration of correctness, stating the main invariants that make the algorithm work reduces the risk of implementing an incorrect algorithm.
6. Determine the time and memory complexities as well as the corresponding running time and actual memory needed. At this point you may decide if it is worth going ahead with this algorithm or better to keep looking for more efficient and/or simpler solutions.
7. Write the pseudo-code of the algorithm, try to simplify it and test it before going ahead with the actual implementation.

On many occasions during this process, students may encounter sub-problems that need to be solved. For example it could be a simplified version of the original problem or something that needs to be done before starting the main algorithm. When students encounter such sub-problems, they tend to work on them with less care and apply less efficient strategies than when they work on the original task. It is important that faced with such a sub-problem, they work on it as if it were the original problem and apply the method (recursively) on it, starting with Step 1.

The structure of the curriculum aims at teaching the students how to apply all of these steps. The hints and intermediate problems often correspond to applying steps of the first part. The detailed solutions try and follow the method closely. Documents synthesizing algorithms and data-structures providing domain-specific techniques are designed to help out during this process, particularly during Step 7 of Part 1. Finally, each step of the method is described in great detail in an independent document.

5. Conclusion

We described a complete curriculum for teaching algorithmics, organized around the aim of teaching problem solving skills. Unlike most traditional curriculums which follow an instructional approach, this curriculum combines the benefits of both guided discovery learning and instructional learning, using the first to introduce new notions and relying on the second to consolidate the knowledge acquired. Thanks to a system of hints and intermediate problems, the discovery learning component is effective for students of different levels.

Throughout the structure, we teach the application of the problem solving method that we have developed along the years. This is done by following an inductive approach: the application of the method is illustrated through the hints, intermediate problems and solutions to the tasks, and then generalized into synthesizing documents.

This training website has introduced algorithmics to hundreds of students over the years, many of whom developed a strong interest in the field. Motivated students often solve more than 200 of the problems in our series within one or two years, going from a beginner's level with only some experience in programming to a level that allows some of them to get medals at the IOI, ranging from bronze to gold.

This success shows that such a curriculum can be a very effective way to teach algorithmics and more specifically, problem solving skills. In the future, we aim at improving this curriculum by adding more series of problems to cover a wider range of domains, by optimizing the structure it is based on, and by perfecting the method that it teaches.

References

Charguéraud, A. and Hiron, M. *Méthode de résolution d'un sujet.* `http://www.france-ioi.org/train/algo/cours/cours.php?cours=methode_sujet`

Ginat, D. (2002). *Gaining Algorithmic Insight through Simplifying*. JCSE Online.

France-IOI. *France-IOI Website and Training Pages.* `http://www.france-ioi.org`

Hiron, M. *Méthode de recherche d'un algorithme.* `http://www.france-ioi.org/train/algo/cours/cours.php?cours=methode_recherche_algo`

Kirschner, P.A., Sweller, J. and Clark, R.E. (2006). Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, **41**(2), 75–86.

Nummenmaa, J., Mäkinen, E. and Aho, I. (Eds.) (2001). *IOI' 01 Competition.*

A. Charguéraud is the vice-president of France-IOI. After his participation at IOI'02, he got involved in training the French team to the IOI. He has designed many tasks and tutorials for the training website, as well as tasks for contests. He is a PhD student, working at INRIA on formal verification of software.

M. Hiron is a co-founder and president of France-IOI. He has selected and trained French teams for International Olympiads since 1997, and is the co-author of many elements of the training website. As a business owner, he works on projects ranging from web development, to image processing and artificial intelligence.