# A Pre-Programming Introduction to Algorithmics

Judith Gal-Ezer[1]

The Open University of Israel,

**galezer@cs.openu.ac.il**

**submitted July 1994, revised March 1995**

**Abstract**

This paper describes an introductory course to algorithmics, which we recommend as a first course in a study program leading to teachers' certification in computer science. This pre-programming introduction to the field of computing emphasizes the algorithmic approach to problem solving, and does not involve any coding. Instead, a visual tool – a multi-layered chart – is introduced, to help in the designing and developing of algorithms. We present here an outline of the course, and the visual tool.

## Introduction

An introductory course to computing or to computer science is taught in many faculties and departments of an academic institution. When it is taught in the social sciences, it is usually meant to teach computer literacy and the use of computers in daily and professional life. When taught in the natural sciences or exact sciences, it is mainly meant to teach scientific programming. How should an introductory course be structured, when taught in computer science departments? In what sense should it be introductory? Should it teach programming? What should be the first language taught? Should it involve coding or not ?

These questions are very well discussed in the professional literature see [DIJ], for example; also many introductory text books have been written, too numerous to refer to here. And yet, in most academic institutions, the first introductory course has not dramatically changed over the years, and it almost always involves the study of a programming language. Even if the teacher has introduction to algorithmics in mind, the emphasis in practice is on the technicalities of a programming language, coding and running programs on a computer.

There is an inevitable downward migration from university, influencing high school and teachers' college curricula and courses. So, obviously the first course in high-school programs or in teachers' certification study programs is also most often devoted to programming, not always serving as a real introduction to the field of computing.

---

[1] Part of this work was written while on sabbatical leave at the Science Teaching Department, The Weizmann Institute of Science, Rehovot, Israel.

We designed a somewhat different course, and integrated it into a study program leading to teachers' certification in computer science. The course constitutes a pre-programming introduction to algorithmics, not involving coding at all, nor the running of programs on a computer. The design of algorithms in our program is done by means of a multi-layered chart, which provides a visual description of an algorithm. We feel that this "no coding" approach, makes it possible to focus on more fundamental concepts, and to provide knowledge and skills of value that is independent of specific computers or programming languages.

# Description of the course

The approach we have taken recommends to shift from "programming" (or rather coding) to "algorithmics", focusing on the algorithmic thinking and the algorithmic approach to problem solving. Instead of starting to code first and running the program on a computer, waiting for immediate feedback from the machine, students will learn how to gradually design algorithms using a visual representation tool.

The goals of the course are :

- To provide an introduction to the key concepts of computing, emphasizing the concept of the algorithmic problem and its solution – the algorithm;
- To teach some algorithms and the data they handle;
- To teach how to analyze algorithms;
- To introduce (on a small scale) the notions of complexity and correctness of algorithms;

Last but not least,

- To provide pre-programming experience in gradual designing of algorithms and implementing them.

Developing the skills needed to design algorithms and analyze them, is by no means a simple task. The construction of algorithms is a complex activity that demands many skills and specific details of knowledge, some of which are usually present in the student's repertoire while others must be learned.

In an introductory course special efforts have to be devoted to designing algorithms, emphasizing modularity, top-down and stepwise refinement. Many things have been said in favor of these notions, and we will not refer to all the papers, books and conference panels that have been devoted to these issues. Let us only mention the very famous paper written by N. Wirth [WIR], and very briefly summarize the benefits of modularity and the stepwise refinement technique.

## Modularity and stepwise refinement

The idea underlying modularity is, in short, the decomposition of a complicated task into a group of small independent subtasks. Not only is handling the smaller problems usually easier, but this method is also helpful in attaining a better insight into the original problem. Also, the comprehension of a long and complicated algorithm may become easier by considering its smaller components.

The modular structure of an algorithm eases the reading and analyzing of it, and also proving its correctness. Debugging will involve only local corrections, that is, correcting one particular module will not necessarily lead to changes in other modules.

As for stepwise refinement and top-down design, let us first comment that despite the fact that we will here advocate a top-down design method, sometimes in real world problems a bottom-up approach is needed.

The technique of stepwise refinement helps to gradually develop an algorithm or a program, in a sequence of steps. The first step is a top design of the solution, which introduces the group of subtasks the algorithm has to handle. In the next step one or more of the given subtasks is decomposed into more detailed instructions. In each of the next steps one or more instructions are decomposed into more detailed ones. This refinement process terminates when one reaches what has been defined as elementary instructions.

It is known that novice programmers who don't use top-down design and stepwise refinement may find themselves referring directly to elementary instructions, resulting in poorly-styled programs, unreadable and difficult to maintain. As mentioned in [SFA], developing algorithms this way is like going on a journey without planning its course, stopping after each step in order to decide where to go next. The way is lost time and again, backtracking and starting all over is inevitable, and the destination reached may very well not be the one initially intended. Using the top-down and stepwise refinement method when designing algorithms is like planning the journey by setting interim goals, and working out a short and safe way.

It should be noted that the top-down and stepwise refinement techniques are, in fact, general strategies for problem solving that are applicable in many fields, including solving complicated mathematical problems or even teaching complicated mathematical proofs [LER].

To conclude, decomposition and stepwise refinement can effectively draw the students' attention to the general structure of the algorithm, and turn the programming itself (translating the algorithm into the language of a certain executer) into questions of secondary importance. The student discovers that there is no need to change the structure upon transition from one executer to another. A well structured algorithm will fit into almost every new environment.

## Outline of the course

The following is a description of the contents of this course.

- Introduction; the computer as an all-purpose machine on the one hand, and the limitations of computers on the other hand.
- The algorithmic problem and the algorithm; the algorithm and the process; characteristics of the algorithm; elementary instructions.
- Constants and variables.
- Expressions and assignment statements.
- Sequentially structured algorithms.
- Ways of representing algorithms: Pseudocode, flowcharts, programming languages.
- The multi-layered chart.
- Correctness of algorithms.
- Conditional structured algorithms.
- Looping: bounded iterations, conditional iteration.
- Efficiency of algorithms.
- Functions and procedures.
- Arrays.
- Sort algorithms.
- Search algorithms.
- Recursion.

We should remark that most of the topics are introduced, taught and exercised, and then accompany the material of the whole course all along.

# The multi-layered chart

Since no programming language is used throughout the course, a good way of representing algorithms has to be chosen. There are some well known and widely used tools for representing algorithms, let us very briefly describe two of them, which have become sufficiently well known to make it unnecessary to dwell upon here.

- *Pseudocode*: This is a tool for representing algorithms in a language closely related to the natural speaking language. When compared with regular verbal descriptions, pseudocode is more precise, it involves special key words, punctuation marks, etc. Using pseudocode helps when translating the algorithm into a programming language.

  But, pseudocode does not always give a good visual representation of the algorithm. However it can be translated into a more visual code - the flowchart.

- *Flowcharts*: These are widely known, were widely used for many years, and are widely criticized (they are sometime called "flaw charts" [GRI]). It is a visual description but it does not comply with the notion of modularity. It describes the algorithm in a flat way, that is likely to lead to programs written in "spaghetti style". The hierarchical modular structure of a program is not seen when using this kind of chart.

We chose to introduce a visual presentation, a multi-layered chart. The idea underlying this type of multi-layered chart, is to maintain the hierarchical structure of an algorithm. The professional literature is familiar with a wide spectrum of hierarchical charts, from the Nassi-Shneidermann diagrams [NAS] to higraphs introduced by Harel [HAR].

"Our" chart is supposed to give a quite full and good description of all the components of an algorithm. It is composed of layers, where the first one gives the top design of the algorithm, and the other layers describe the more refined details, mirroring the stepwise refinement process.

Special boxes are used to describe the beginning and the end of an algorithm, other boxes are used for assignment statements, for input and output statements, for conditions, loops, and for procedures. We will not go into the details of describing each box and explaining what it stands for, this of course should be done in class. However, Figure 1 and Figure 2 which are self-explanatory, show all the elements needed to draw a multi-layered chart of an algorithm.

Figure 1 is a multi-layered chart of an algorithm that calculates and prints the well known Fibonacci series (i.e. $f_n$, such that $f_{n+1} = f_n + f_{n-1}$, $n>0$). The algorithm gives all the Fibonacci numbers that are smaller than a given $n$. The chart has four layers. The first layer gives the main procedures: *input*, *initialization*, and *compute* – the computation of the new Fibonacci number. The second layer describes the procedures in more detail. One of the procedures – *compute* – invokes another procedure – *next element* – which is given in detail in a third layer, and which in turn invokes the procedure *output*, that is given in the fourth layer.

The chart provides an overall look of the algorithm. Each layer and each procedure can be viewed in detail. One can zoom into the procedure *compute* for example, which is drawn in the second layer. The heart of procedure *compute* is a *while* loop. While $f_2$ is smaller than $n$, next Fibonacci number has to be computed, so procedure *next element* is invoked. When $f_2$ becomes equal to or greater than $n$, control is returned to the calling procedure (which is the main program in this case).

Figure 2 is a multi-layered chart of an algorithm that solves a quadratic equation, i.e. $\boldsymbol{ax^2 + bx + c = 0}$, or linear equation, i.e. $\boldsymbol{bx + c = 0}$. This chart has three layers, the first gives the outline of the algorithm, and the other two the refinement of it. The *branching* structure which is very well distinguished from the sequential structure, plays here an important role.

We think that this chart has many pedagogical advantages:

- It visually presents the decomposition of an algorithm into sub-tasks, motivating the modular design of an algorithm.
- It supports the top-down and stepwise design of an algorithm.
- It eases the identification of similar algorithmic structures.
- It enables a systematic and controlled passage between the declarative description of the solution and the detailed description of the execution.
- It facilitates translation from one programming language to another.

When presenting multi-layered charts to a class, different colors can be used for each different layer. This will make the description even more vivid.

However, this visual tool sets some practical limitations. The gain in overview may be lost if there are too many layers. The layout of the chart may become very cumbersome; it will probably be a multi-page layout chart, and will not achieve its goal of producing a clear visual representation of the algorithm. Some suggestions to overcome this, are given in the discussion section. However, we feel that this practical problems limit the use of the multi-layered chart to simple problems of the kind we meet in introductory courses. This is where the advantages of the chart are the greatest, and thus achieves our aim.

# Discussion

The main aim of the course outlined above, is to give a fairly good knowledge of the basic methods of designing algorithms, without using the more difficult traditional programming languages. When learning this introductory course, the students will be able to concentrate on the algorithmic thinking behind computer programming. The visual tool used to present algorithms – the multi-layered chart – supports methods of structured and modular programming, and stresses top-down and stepwise refinements techniques.

The experience, when teaching the course this way to prospective teachers, showed that most of the students internalized the modular way of thinking when setting out to solve a problem. Some students stated that their approach to problem solving has become more general and methodological in other subjects as well.

Teachers of the second course, which involves programming, stated that the students were well prepared to learn programming in Pascal, say.

We pointed out above, the advantages and disadvantages of the multi-layered chart; we explained that it may lose its overview advantage when it is composed of too many layers or modules. To overcome this disadvantage, we suggest to further develop a visual software tool, based on the multi-layered chart. By doing so we may achieve two goals: First we may overcome the multi-page layout of complex algorithms, using hypertext methods, for example; second, this is an opportunity to integrate the computer into the learning of computer science, not only as a machine that runs programs, but also as an educational tool.

**Figure 1**: A multi-layer chart of an algorithm that computes and prints Fibonacci numbers ($f_n$, such that $f_{n+1} = f_n + f_{n-1}$, $n>0$) smaller than a given $n$.

**Figure 2**: A multi-layer chart for solving a (linear or) quadratic equation ($ax^2 + bx + c = 0$).

**Acknowledgment**
I would like to thank my colleague G. Zwas for many very fruitful discussions, and my student Ela Zur, whose Master's thesis motivated this work.

# Bibliography

[DIJ] Dijkstra, E. W., "On the Cruelty of Really Teaching Computing Science", *Comm. Assoc. Comput. Mach.*, **32**, 12, 1989, pp. 1398-1414.

[GAL] Gal-Ezer, J., "Computer Science Teachers' Certification", 1994 submitted.

[GRI] Gries D., *The Science of Programming*, Springer-Verlag, New York, 1981.

[HAR] Harel D., "On Visual Formalisms", *Comm. Assoc. Comput. Mach.*, **31**, 5, 1988, pp. 514-530.

[KUS] Kushan, S. B., "Preparing Programming Teachers", *SIGCSE Bulletin*, **26**, 1, 1994, pp. 248-252.

[LER] Leron , U., "Structuring Mathematical Proofs", *American Mathematical Monthly*, **90**, 3, 1983, pp.174-185.

[MER] Merrit, S., et al., *ACM Model High School Computer Science Curriculum*, Assoc. Comput. Mach. Inc., 1994.

[NAS] Nassi, L. and Schneiderman, B., "Flowcharts Techniques for Structured Programming", *SIGPLAN Notices of the ACM*, **8**, 8, pp. 12-26.

[ROG] Rogers, J., et al., "Computer Science in Secondary School: Course Content", *Comm. Assoc. Comput. Mach.*, **36**, 12, 1985, pp. 270-274.

[SFA] Sfard, A., *Teaching Theory of Algorithms in High School*, Ph.D Thesis, in Hebrew.

[WIR] Wirth, N., "Program Development by Stepwise Refinement", *Comm. Assoc. Comput. Mach.*, **14**, 4, 1971, pp. 221-227.