Introductory Programming and Word Problems

Jozef Hvorecký
Dept. of Mathematics
University of Papua New Guinea
hvorecky@upng.ac.pg

Abstract

Teaching introductory programming courses in developing countries has its specifics. It must reflect a different structure of their ducational system, the lack of resources (teachers, textbooks, classrooms, etc.), and a shorter tradition of education itself. Particularly, our students have many difficulties with doing good programming abstractions. Their thinking and reasoning is very concrete, the students can not formulate algorithms (designs of their programs). As a result, their programs are "random sequences of instructions" rather than "computerized solutions of problems". That's why I have tried to shape a methodology, which could show them the consecutive development of programs - starting with verbal formulation of the problems. In the paper the milestones of the methodology are presented: First, our problems are formulated as word problems. Each problem exploits the student's previous knowledge and experience and activates his/her creativity. Then, we want students to describe algorithms - we only ask for their verbal (but written) design. The correctness of the designs is verified. If the solution is correct, it is expressed in a programming language. Another important aspect of our methodology is that all new notions, concepts, and programming techniques are introduced in the "bottom-up" manner. The students can "reinvent" and use them before the notion is formally defined. In such a way, they can easier understand its meaning and observe the most frequent ways of its usage. Morever, we try to solve every problem using several different ways and do encourage students to do the same with their assignments. Then, we discuss their advantages and disadvantages. We see such critical analysis as a necessary backgroung for their future professional carrier. Thus, the approach could be described as "from word problems to (validated) computer programs". In our presentation some of those problems and their solutions will be demonstrated.

1 Introduction

The education in developing countries must be based on different methodology compared to that of the developed ones. In general, it must reflect a different structure of their educational system, the lack of resources (teachers, textbooks, classrooms, etc.), and a shorter tradition of education itself. Particularly, the lack of such a tradition seems to be the most critical factor. The pressure of parents and the society on students to study harder and to improve their skills is much smaller than an outsider could expect. The reasons are simple:

- Most parents can not guide their children in their learning and the selection of their study field simply because their education is minimal (if any).
- The structure of the society does not correlate to that of the most advanced countries. Most university newcomers have no idea about main characteristics of their future professions, because they have never had any opportunity to observe it in their neighborhood.
- Due to lack of long-term planning the university subjects and the numbers of their attendees might not fit the real situation students will face after their graduation. That results into the students' lower interest in their studies.

Altogether, the organization and orientation of the university study must concentrate on production of graduates able flexibly reacting to the future changes in their countries. Educators should concentrate on forming the students' thinking and reasoning skills, problem solving methods and positive attitude to their future cultivation. In this paper we show an attempt to reach these goals in programming.

2 Expectations and Experience

Currently I teach computer science subjects at UPNG little more than one year. At the time of my arrival I presumed:

- My students are mostly studying mathematics. So, they should have developed their *problem solving* skills.
- The students are already computer literate, because they completed the one-semester *Introduction to Computing* course. Thus, my C++ course could be a *regular programming* course similar to those given at other universities round the world.
- In a short period I found the opposite:
- The students show low problem-solving skills. The low quality of elementary and high school teaching is probably the main reason. Particularly, they are quite good in performing any kind of mathematical operations, but very bad in solving word problems. This causes their weak

orientation in the interconnections between mathematics and our real world.

- Their hands-on experience with the computers was low as a result of insufficient access to computers in the past. Moreover, the students made their first contacts with computers rather late – in the end of the second year of their university study. Luckily, this factor is now gradually changing with better hardware facilities.

3 Ways Out

Critical situations require non-standard solutions. Thus, I have decided to modify existing programming courses and to build a new approach. It is based on the following principles:

1) Understanding basic concepts of programming: Any abstract thinking is based on a deep comprehension of relationships among concrete facts. Thus, each notion and concept is to be explained using two ways:
   a) As an isolated element – the concept is introduced in its “pure form“.
   b) As a part of a mosaic – the interconnection of the concept with other (already known) notions and concepts is explained. Their various combinations are studied, tested and analyzed.

2) Developing problem-solving skills: Programming is something more than just typing commands executable by a computer. The commands must express the idea of the programmer. The ideas become our primary objective as no one can make a good program without a good idea. All this means that we start every explanation with a problem. The students are encouraged to find out as many its solutions as possible. Then, the solutions are tested and evaluated.

3) Slow pace: The students’ comprehension is preferred to the number of notions. Naturally, this principle slows down the speed of presentation. The speed follows the students’ progress.

4 Problems and Their Solutions

Every notion is only introduced when it is necessary for solving a problem. Under the problem we understand a triple[1]

```
/* Pre: Precondition */
A
/* Post: Postcondition */
```

The precondition *Pre* specifies the set of values that are expected as input data. The postcondition *Post* defines the results that should be reached. As the way of their obtaining is unknown, the postcondition only specifies a relationship between the input data and expected results. Our goal is to find an algorithm A, which transforms data satisfying the precondition to data satisfying the postcondition. However, every solvable programming problem has many solutions with different complexities. Thus, we present some of them to students and encourage them to create their own and discuss their advantages and disadvantages.

4.1 Introducing Basic Concepts of Programming

We apply the problem solving approach from the very beginning. For example, first C++ commands are introduced and studied using the following simple problem:

```
/* Pre: Two integer variables A and B are given, A < B */
A
/* Post: A ≤ B is valid for the results */
```

This problem has many solutions:

1) *Null command*: It is easy to show that the null command (“Do nothing!“) solves the above problem. If $A < B$ holds for two variables, $A \leq B$ is also valid. Thus, none of them need be changed. That command will frequently used as a branch of conditional commands.
2) *Assignment command*: It is one of the basic commands of most programming languages. In our case we use the assignment A = B. After its execution the equality $A = B$ holds[2]. Naturally, its validity also implies the validity of the postcondition $A \leq B$. To understand its properties students are encouraged to look after similar assignments (say, B = A).

[1] S. Alagic, M. A. Arbib: The design of well-structured and correct programs, Springer-Verlag, New York, 1987

[2] The usage of the equality symbol “=“ as the assignment symbol in C and C++ is not very fortunate. The author feels it as a step back compared to the “:=“assignment symbol in Pascal.

3) Recurrent assignment: A = A + 1 is the most important variation of the assignment. There exist several of its variations to be discovered by students:

a) A = A – 1;
b) B = B – 1;
c) B = B + 1;
d) A = A – 76 (or any other constant)
e) etc.

4) Compound command: {A = 36; B = 36;}. When the same constant is assigned to the both variables, A = B becomes valid. As in the previous case, that proves the validity of the solution. An important characteristic of this solution is that it has infinitely many variations. So, students are asked to show which of the above solutions can be used as similar generators. A deeper discussion shows that the 3a, 3c and 3d can generate infinitely many solutions, but 3b not.

5) While-loop: **while** (A < B) /***do***/ {A = A + 1;}. Such a "Pascal-like" *while* loop is exclusively used, because it is better readable than the standard C++ loop without /***do***/[3]. The following analysis – made with the students' "help" – investigates whether the loop body can be replaced by another assignment. It is easy to see that the solution 3b (i.e. the assignment statement B = B – 1) can be used as the loop body, but not the others (3a, 3c, nor 3d). All of them create infinite loops[4]. On the other hand, when the loop body is replaced by the solutions 2 or 4, the loop body is executed only once. That indicates that these solutions are "simpler" that the loop.

6) Do-loop: **do** {A = A+1;} /***while***/ (A < B). A similarity in appearance of two texts does not mean an identity of their performance. In this case the sections of the previous text has been exchanged. As a result, the execution of the loop body and testing the loop condition are also made in a different order. Similar modifications help students understanding of a correlation between the text of a program and its execution.

4.2 Elements of the Computational Complexity

Finally, after collecting many solutions, we can raise the question: *Which solution is the best – and **why**?* It allows us to introduce elements of computational complexity: The solution, which requires fewer commands to be executed for a given input, is better. Students easily conclude that there exist three important activities during the execution:

[3] We do the same with all other commands. For example, the conditional command is used in the form: **if** – /***then***/ – **else**. Additional commentaries significantly improve the readability of the commands.

[4] Students "discover" the existence of infinite loops before they run their first loop at computer. Naturally, they are later less surprised by this "strange" behavior of computers.

- Assignments of values into the variables;
- Comparisons of two values;
- Arithmetic operations.

One can easily see that there exists one solution, which is undoubtedly better than all others – the first one. It requires zero operations. That means that there can never exist a better solution[5]. Later, these basic concepts are trained and developed on more complicated problems.

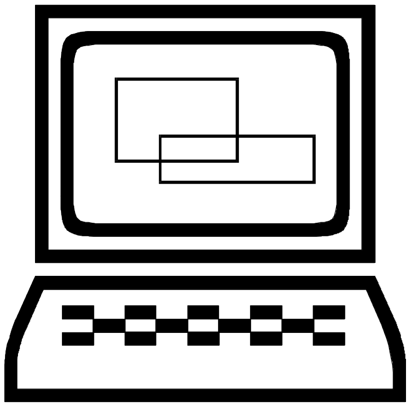

Figure 1

4.3 Algorithmic thinking

After starting solving more complex problems we concentrate our efforts around one crucial idea: ***"Simple"*** *does not mean the same for a computer and for a human*. To program some problems, which are trivial for a human, might be very complicated. Many of our decisions are based on our intuition. Roughly speaking, "we do not know what we know". In such a case the programmer has to "discover" a process simulating our decision-making[6]. It is an ideal way of allowing students to feel "*Heuréka!*" – the moment frequently lacking in our education and the best tool to raise the students self-confidence.

Figure 2

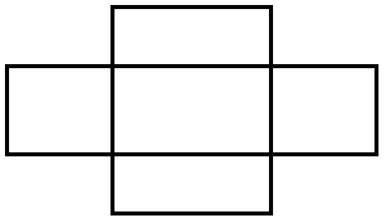

Figure 3

The following problem is a good example of such reasoning. It is connected to overlapping two windows at a computer monitor (Figure 1): *There are two rectangles at the monitor. Their sides are parallel with the edges of the screen Do they have a common area?*

[5] Any better solution should execute a negative number of operations. An occasional discussion on "the algorithm with a negative number of commands" was very interesting. It gave to the students an insight on the concept of the natural numbers.

[6] It does not mean that we solve the problem in the same way. Just the results of the processes are the same.

When students generate the rectangles at screen (using a random numbers generator), they can easily test their programs. And they see that the results of their programs differ from what they see at screen.

As the rectangles are defined by the locations of their upper left and lower right corner, solutions are usually based on the investigation whether a corner of one rectangle is located inside the other. Because the investigation requires testing many complex conditions, a natural question arises: *How many corners are to investigate? Eight? Four?* “Eight“ is a safe, but long way. “Four“ need not be enough – see Figure 2. None of the corners of the outer rectangle is located inside the inner one. However, five is enough, because the fifth corner must belong to the inner.

Most students believe that this is the complete solution of the problem. It has always been a good idea to show them a counterexample. There exist pairs of rectangles, which are overlapping, and at the same time none of their corners is located inside the other rectangle (Figure 3).

4.4 Functions

C++ is an object-oriented language. Therefore, the deep and correct understanding of functions is an important prerequisite for the students’ future progress with the objects. Again, we use a set of problems, which bring them nearer to such comprehension. There is one of them: *A **palindrome** is a textual or numerical string that reads the same backwards as it does forwards (**deed**, **radar**, and **1991**). There is a way to generate numerical palindromes: Suppose a number is given. If it is not a palindrome, add its mirror image to it. Repeat the same with the result until you get a palindrome.* Figure 4 shows the computation, which starts with 5843.

Top-down design is our preferred way to program creation. For our first version of the program looks as follows:

```
{input Number;
while !(Palindrome(Number))
   /*do*/{Number+=MirrorImage(Number);
          print Number;
};
};
```

```
    5843
    3485
--------
    9328
    8239
--------
   17567
   76571
--------
   94138
   83149
--------
  177287
  782771
--------
  960058
  850069
--------
 1810127
 7210181
--------
 9020308
 8030209
--------
17050517
71505071
--------
88555588
```

Figure 4

Here we use two functions: *Palindrome* and *MirrorImage*. First, this example presents a variety of function calls: One function is used inside a loop condition, the other in an assignment command. Secondly, it can also demonstrate a possibility to call a user-defined function inside the other. It is easy to see that a number is a palindrome only when it is equal to its mirror image. This consideration leads us to the function:

```
int Palindrome(int Number)
{if (Number == MirrorImage(Number))
     /*then*/    {return Yes;}
       else      {return No;};
}
```

In this way students easily understand the reasons for our preference of the top-down design to other program development methods.

4.5 Word Problems

There exists another big methodological problem. Students do not have a right picture on the interconnection between programming and our real life. They do not understand that any programming language is just a tool, which is used for the expression of our intentions and their formulation in a form acceptable by a computer. The correct comprehension of this relationship is crucial for their future professional performance. To build it up we force them to solve many word problems. Here we show some of them.

A bank uses the following algorithm with its credit cards: When the card is issued to a customer, he/she types a four-digit number to the bank computer. For the privacy reasons, the number is not stored in its original form. It is encoded as follows:

a) All digits smaller then five are increased by 1.

b) Decrease 2 all other digits.

Write a program to perform the above transformation.

For various reasons students do not always understand the text well. That’s why from two to five sets of testing data are given. In the above problem the sets have the form:

a) For input 3707 the value 4515 is stored.

b) For input 4648 the value 5456 is stored.

As we have already mentioned, the students have a weak mathematical background. Their picture of a “proof“ is quite uncertain. Frequently, they assume that the validation of a hypothesis for a concrete data set is the equivalent of a proof. To disprove their conviction a problem, which uses random data is needed. In this case the student can not know what will be the actual input of his/her program. Thus, his/her solution must be “bullet-proof“. There is a good example:

N soldiers stand in a line (side by side). On their commander's order "Left face" they (randomly) turn 90^o to the left or to the right. In the next second those who stand face to face with another soldier, realize that "Something is wrong" and turn about face (i.e. 180^o). The same repeats in the next second etc.

a) Write a program, which simulates the soldiers' behavior. Program should display their line in one-second intervals. Use the character "<" for a soldier facing to the left and ">" for a soldier facing to the right.

b) Determine and display the duration of the process from its beginning to the moment when no soldier moves.

c) Prove that the process always terminates.

d) What is the maximum duration of the process with N soldiers?

Realize that the last three questions are theoretical. There is no way to solve them using the computer[7]. In this way we underline another of our aims: Forcing students' thinking about their programs not facing their computers. They should build up their mental structures to become capable of algorithmic reasoning anywhere[8].

Most word problems are also oriented to specific topics. The following problem has been constructed to pinpoint a possible usage of the same value for two different goals: *A farmer sells 50 watermelons. The melon weights vary from 1 kg to 10 kg, but their price is the same – K1 per piece. (That's why the farmer only registers their weight in integers – it does not play an important role in his considerations. As we will see, it will play a very important role in those of ours.) There exists at least one melon of each weight from 1 to 10 kg. The weight of melons is stored in elements of the integer array* ***Melon****. Our car can only carry 100-kg load.*

a) Design an algorithm, which counts the number of melons of each particular weight from 1 to 10 kg and stores the numbers in the array ***Amount****.*

b) Design an algorithm, which helps us to select the maximum weight of melons with the minimum price. (The previous algorithm can be used as its part.)

c) Show that if the total weight of melons is at least 100 kg, one can always select melons in their total weight 100-kg.

d) Write a C++ program that corresponds to the algorithms.

Particularly, the weights of melons are used as values of array elements in the array *Melons*, but they must be simultaneously used as the indexes of the array *Amounts*.

[7] There is one "way": To run all possible combinations of the starting positions N soldiers. However, the actual number of the positions excludes that possibility.

[8] The author is convinced that the position in the front of a computer is probably the worst place for thinking about your program. This position provokes your examining immature ideas.

5 Concluding Remarks

The tradition of education in Papua New Guinea is very short. It explains why the students' background is quite weak. However, we can not feel comfortable with this judgement. We have to elaborate teaching methods, which would accelerate such an appreciation to education similar to that of developed countries. We believe that the teaching methods similar to the presented one will help the students build up their abstract thinking and reasoning. And we wish it would be so simple and so fast as in the following problem used for practicing the recursion:

One lecturer teaches programming in a class of 12 students. After 4 years the students are qualified enough to start their own classes of 12 students. The students are also obliged to train 12 students in 4 following years etc. How long would it last while the whole population of Papua New Guinea[9] *is computer literate?*

References

[1] J. Adams, S. Leestma, L. Nyhoff: *C++ An Introduction to Computing.* Prentice Hall, Englewood Cliffs, 1995

[2] P. Brusilovsky, E. Calabrese, J. Hvorecký, A. Kournichenko, P. Miller: Mini-languages: a way to learn programming principles. *Education and Information Technologies,* Vol. 2, No. 1 (1997), pp. 0065-0083

[3] P. Drlík, J. Hvorecký: *Informatika - nácrt didaktiky* (*Information Science – an Outline of Didactics*). Pedagogická fakulta, Nitra 1992

[4] J. Hvorecký: *On a Connection Between Programming and Mathematics. SIGCSE Bulletin*, Vol. 22, No. 4 (Dec. 1990).

[5] J. Hvorecký, J. Kelemen: *Algoritmizácia* (*Algorithm Design*). Alfa, Bratislava 1983, 1986.

[9] The estimated population is 5 million inhabitants.