Recursions and how to teach them

Predrag Brođanac
V. gimnazija, Zagreb
predrag.brodanac@skole.hr

Summary – earning programming is difficult. This refers to primary and secondary school students, but to university student beginners at programming too. These particular reasons stand as a challenge to teachers, as well. Most of the authors agree that recursion presents one of the most difficult concepts at programming, whereas some authors think it is the most difficult concept in programming. On the other hand, it is no longer only an important mathematical concept, but a programming technique, a way of algorithm thinking, and a tool to problem solving. There have been many attempts how to teach recursions. Hereinafter, one can find certain approaches for which authors claim to bring a better understanding of this important concept.

Key words: recursion, learning models, learning styles.

I. INTRODUCTION

Recursion is a technique within which a program function, a procedure or a method calls itself; and a recursive algorithm is the one self-referencing [1] If a function continuously calls itself, the process would never stop, therefore a recursive function should meet three of the following conditions [2]:

- recursive calling termination condition,
- recursive relation,
- consecutive recursive calling should be converged according to the recursive calling termination condition.

The basic example of a recursive function is the function for product of first n positive integers, known as the factorials. In the Python programming language function is as follows:

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n - 1)
```

Command (#1) has a defined recursion termination condition. The issue here is a basic case with a trivial problem solution. In the upper example, a basic case is the one in which one should multiply the first 1 positive integer and that product is 1. Should the basic case not be the situation here, the function performance is done recursively (#2). Namely, the product of the first n positive integers can be attained in the manner that the product of the first n–1 positive integers is multiplied by n. This statement is evident, for example, at multiplication of the first 5 positive integers:

\[
\text{fact}(5) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = \frac{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5}{\text{fact}(4)} = \text{fact}(4) \cdot 5
\]

At the same time, we can observe recursion as a loop generalization [4]. A loop consists of a set of commands that repeat. The most common loops used in programming languages are the for and while loops. Commonly, the loop has one of the following two shapes:

```python
command-1
command-2
... command-n (decision on repeated performance)
```

or

```python
command (decision on one-time execution)
command-1
command-2
... command-n (decision on repeated execution)
```

One can easily notice that decision on executing of a block shall be made at the end or at the beginning of the block of commands. Recursion can be observed as a case where a decision on repeated execution of a block of commands can be made at any point within the block:

```python
block-1
command
... Jump to block-1 (condition satisfied)
... command
end of the block
```

During executing of the loc , the execution comes to a stop and the commands of the same block are called again. This repeated call may cause another call, etc. After the commands of one block have been executed fully, we return to the following command of the block that has called the block execution, wherein the executed block's results may be used in the future for executing of the block.
that the block executed. It is important to mention that while calling the new block, the status of the block that called it should be filed, and then later on loaded, and continue to execute. It is realized by runtime stack [5]. Each time a recursive subprogram has been called, variables, parameters, return address and other required data are stored on the stack. At the moment when a subprogram is returning the control over to the subprogram which called it, the related data are popped from the stack.

A large number of programs, that are usually solved recursively, are possible to be solved by using loops, as well. For example, such is the program for multiplication of the first \( n \) positive integers. Its non-recursive solution in the Python has the following form:

```python
def fact(n):
    t=1
    for i in range(2, n + 1):
        t *= i
    return t
```

There are recursive problems which cannot be solved by loops or the loop-based solution is much more complex, and this may serve as motivation for recursions [6]. Sometimes, instead of the recursive, one can implement a problem solution which, using the stack, simulates recursion, e.g. for labyrinth. It is realized by elegant explanation of complex algorithms and data structures [7]. There are situations in which recursive solution will not be efficient. Such example are the Fibonacci number computations, therefore avoiding the multiple computing of such a recursion is generating all permutations of the set from \( n \) various elements.

The appropriate recursive solution in the python would be:

```python
def b_z(n):
    if n < 3:
        return 1
    else:
        return b_z(n-1) + b_z(n-2)
```

![Picture 1: recursive computing of the \( i \) onacci number](image)

As one can see, certain recursive calls, e.g. , occur on several occasions, which shall result in slower function performance than the following non-recursive solution:

```python
def b_z(n):
    a = b = 1
    for i in range(3, n + 1):
        a, b = b, a + b
    return b
```

One should mention that recursive solution of the \( n \)th Fibonacci number computation may be upgraded in the way that the already calculated problem solutions be stored in the data structure, therefore avoiding the multiple computing of such a recursion [8]. Numerous researches show that the concept of recursions is highly complex for programming beginners. Problems arise at defining the basic case, already [9]. A far greater problem is pinpointing the recursive relation in the problem set [10] and [11]. Furthermore, it is no simpler to follow the recursive relation performance and establishing the recursive function value for the defined values of parameters [12], [13] and [14]. One of the possible reasons of having a hard time understanding the recursions is non-existence of adequate analogies in the real world, [1] and [14]. In order to make this concept understandable to beginners, a set of tools has been created for recursion visualization, motivational games have been created in which a beginner subconsciously develops their recursive thinking, furthermore real-life examples have been chosen to help understanding recursions, and methodologies have been invented how to teach recursions, and other things as well.

II. CLASSIFICATION OF RECURSIONS

Recursive algorithms can be classified on multiple grounds. One of such categorizations has been stated by [15]:

- **linear** – has only one recursive call. A typical example of such a recursion is computing factorials. A special type of linear recursion is **tail recursion**. This recursion has recursive call as the last command within the function.
- **multiple or exponential** – doesn't have only one recursive call, but it can have more of them. An example of such a recursion is generating all permutations of the set from \( n \) various elements.
- **nested** – recursive call is one of its arguments. **Ackermann function** is an example of such a recursion,
- **mutual** – this one doesn't call only itself, but another function, which then calls the first function again. We can have more than two functions. An example of such a recursion is a slightly different point of view on computing the \( n \)th element of the Fibonacci sequence [15]

\[
F_i = A_i + B_i, B_i = \begin{cases} 1 & i = 1 \\ A_{i-1} & i \neq 1 \end{cases},
\]

\[
A_i = \begin{cases} 0 & i = 1 \\ A_{i-1} + B_{i-1} & i \neq 1 \end{cases}
\]

There is another division of the recursive programs defined by [16]. He addresses three types of recursions:

- **structural** – at recursive call, the arguments, by which the recursion is called, are either unchanged or closer to the basic case. If the parameter of the recursive function be, for example, data structure, the recursive call will happen over the real subgroup of that structure. That way, when binary tree traverse, one recursive call traverses only one tree branch,
- **generative** – the arguments used to call the recursive function are computed each time over again and there is no guarantee that a change of parameters leads to the basic case. Examples to such recursions is the **Collatz problem** and Euclidean algorithm for

\[
A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0 \land n = 0 \\ A(m - 1, A(m, n - 1)) & m > 0 \land n > 0 
\end{cases}
\]

---

1 The first and second Fibonacci number are 1, and each number is the sum of the two preceding.
calculating the largest common divisor of two positive integers,

- **accumulative** – in these recursions, among the parameters of recursions, there is one or more or several additional parameters containing partial solutions of problems.

### III. ARE RECURSIONS TRULY THAT DIFFICULT?

Almost every author writing about recursions, already in their introductions claims that it is one of the most difficult concepts for programming beginners. Even [17] and [18] agree with the fact that it is the most difficult concept. According to the research described in [18], including 559 students and 34 teachers at 6 European universities, the most difficult concepts for the students were: **recursions, pointers, a strict data types** etc.

Having finished learning the recursions, students will be using them seldom for any other further problem solution, except if it has been explicitly demanded from them to do so [19], and even in situations when they have an offered solution to the same problem by the loop, i.e. by recursion, they will get a better understanding of the solution implemented by the loop [20].

Recursion problem is linked to constructivism [1], is argued by the fact that making references to the already existing knowledge is really important for gaining new knowledge, whereas recursions have nothing to refer to on account of them being so specific.

### IV. CONCEPTUAL MODELS FOR TEACHING RECURSIONS

Many an author has made an attempt to make the recursions more appealing to beginner programmers. Some authors think that suitable preparatory actions, such as playing specially designed games, can help in later understanding of recursions. Whereas, others are of the opinion that analogies to problems that students are familiar with are of much importance; and there are some who think that visualization of the recursion flow is essential, while there are those whose opinion is that following the flow is not necessarily connected to development of recursive thinking, but they regulate a line of steps how to develop a recursive way of thinking.

**Mental model** defines a certain way of understanding abstract concepts or certain real systems [21]. It determines our way of thinking and acting, as well. **Conceptual model** stands for a manner of presenting new contents by an individual. Understanding of the content can be described as possessing a mental model on the content. Conceptual model represents an important tool for understanding of and teaching new contents. One of the teacher's tasks is to build a conceptual model enabling students develop a suitable mental model. According to the **structural mapping Theory** an important factor in adopting new content are **analogies**. Analogies represent a connection between a known (use) and unknown (target) domain. The conceptual model, within that context, would represent the base we take the analogies from for the unknown domain (target). Considering the area which we take the analogies from, we can talk about

- **a strict conceptual model** – its base is abstract, for example the mathematical model,
- **concrete conceptual model** – its base is concrete, for example a concrete object.

There is a set of models for teaching recursions, [21] bring 5 models:

- **Russian Dolls** – the Russian doll can contain a smaller doll within itself, which contains a smaller one, etc. (recursion),
- **process Tracing** – tracing recursive calls. It describes the functioning of recursion,
- **table Simulation** – recursion mechanism is simulated from the computer perspective, where each recursive call creates a new record on the runtime stack,
- **Mathematical Induction** – recursions from the formal, mathematical aspect,
- **Structure Template** – explains recursions based on the example of recursive programs, describing therefore the basic case and recursive relation. The template for solving recursive problem comes down to: searching for recursive relation and defining the asic case.

In their work, the authors refer to learning styles, making reference to the Kolb's definition of learning styles, according to which a manner of information processing is one of the basic factors having influence on mental model design. Combination of manners, in which we perceive information and in which we process them, defines our learning style. Speaking of Kolb’s learning styles we can observe them through two dimensions:

1. **way of adopting knowledge concrete experience and a strict conceptualisation**
2. **way of experience transformation reflective and active perceiving**

School students and university students with different learning styles will have a different reaction to various teaching methods, therefore teaching methods should be suitable to students' learning style. The authors have performed a research showing that students with abstract learning style are more successful at learning recursions than students with concrete learning style. No research has shown that the abstract type has a better understanding of recursions if they have been explained by abstract model, nor that the concrete type has a better understanding of recursions if they have been explained by a concrete model.

Two mental models for teaching recursion describe [7]:

- **The little people model** – there is a large number of little people in the computer, each of them being an expert for a specific task (function) performance. When there is a certain problem, it is to be divided into smaller pieces and the little people hired to solve these smaller pieces. An example of such a model from perspective of a manager and a worker has been described even in [3].
- **Top Down Frames** model – enables tracing recursive calls on paper. A new window is created for each recursive call that is within the window which has called it. Within the window there is recursive function program code with concrete parameters values, whereas at the bottom of the window there is a value that the appropriate recursive call returns.

\[ \text{For the positive integer } n \text{ we define transformations in the following manner: if the number is even, it becomes } n / 2 \text{, otherwise it becomes } n^2 . \text{ The question is how many of transformations, such as this one, over the } n \text{ is to be performed in order to reach number 1.} \]
The author [5] addresses the following recursion teaching models in his work:

- **Induction** – recursion is explained as a function that calls itself. The Fibonacci numbers are set as an example. It is an important approach, however it lacks a deeper understanding of how recursions function.

- **Runtime stack** – a simulation runtime stack is created in computer, which stack is an occurrence in recursive calls. The shortfall of this manner of tracing occurrences in the runtime stack is it being done on paper.

- **Unravel performance tracing** – each time there is a recursive call, a line is written with the name of the subprogram being called, and with suitable parameters. The same thing happens at returning from the recursive call. At output, one has to take care of the indents in order to visualize recursive calling. Tracing can be realized on paper, as well, and it makes a runtime stack simplification. Presenting recursive functions that call themselves iteratively may not be the perfect solution, e.g. functions for computing the Fibonacci numbers, i.e. Ackermann's function.

- **Recursion tree** – tree whose nodes represent the momentary surrounding, encompassing the parameters and local variables. At that point, the nodes are identified with recursive calls, and a node's children are recursive calls occurring in a suitable surrounding. Some of the shortfalls of the recursive tree are: (1) it does not contain the value returned by the recursive call; (2) it is not easy to draw a tree in case when we have more than two recursive calls within a function.

- **Activation tree** – a combination of the runtime stack and recursive tree. Each node contains, besides the name of the function and parameters, the function's return value. The author is of the opinion that this manner has no shortcomings.

### Picture 2 Activation Tree node

<table>
<thead>
<tr>
<th>Name of the function</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>parametre 1</td>
<td></td>
</tr>
<tr>
<td>parametre 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**A. Preliminary work that improves recursion learning**

One of the ways that contributes to a better understanding of recursions later on, is playing a well thought computer game that will develop the recursive way of thinking. Authors [22] and [2] describe one of these games, called Cargo Bot, showing that it helps in recursion learning.

**B. Concrete Teaching examples**

The largest number of authors has been studying some of the learning models. [23] provides 3 examples that can serve as introductory examples, of which each may have its drama version:

- **Penning of presents** (closed brackets problem) – a friend has given you a watch, however, in order to keep the suspense going, the watch had been placed inside a box, inside another box, etc. The objective is to describe the gift opening algorithm in one's own words.

- **Chain design** (factorials) – there are links with springs defined, which are to make a chain together. The idea of recursive solutions is to give one's friend to make a chain of length, and then add one more link at the end of the chain.

- **Ating chocolate** (searching for elements on the list) – at our disposal we have a bar of hazelnut chocolate. We are to eat only those squares containing the hazelnuts.

The authors of the [24] have an interesting way to describe how to adopt the recursion concept. It is their opinion that in times of object-oriented programming one should change their approach to teaching recursion, as well. They, also, think that recursion learning should be implemented over data structures. This is illustrated by an abstract data structure list _ur ist in the python_, whose every instance had two properties: _head_ – preserves the element value, and _rest_ – all the remaining list elements. The authors have been able to explain this kind of a structure to the students, and, through role playing, the students were to come up with certain basic operations over the lists: counting how many times does a certain element appear on the list, verifying whether or not the defined element _e ists on the list_ adding an element at the end of the list and so on. Each student represents an instance of the list and the teacher has established which student represents the whole list. Only the students sitting next to each other are allowed to speak to each other.

The authors [25] provide an example to recursion teaching of children at the age of 11 to 14. Recursions are described on examples not directly connected to programming. The following problems are stated as examples:

- **Genome sorting** – the students were handed envelopes containing integers (from 1 to 14) and were to line up the numbers without opening the envelopes. They were using only one student, _comparator_. He was the only one able to open the envelopes and say which envelope contains the larger number. The comparator's service costs one monetary unit for each comparison. The objective is to sort the envelopes the way to spend as little money as possible.

- **Sierpinski Carpet** – one of the students is drawing, and the other one is describing. The one describing was given the picture of the _sierpins i Carpet_ and is to describe, to the one who is drawing, how to draw the picture.

- **Line Up** – students are lined up, as in a queue, and their eyes have been blind-folded. They are to determine how many students are there in front of them in the queue, without ever leaving the line.

Parallel car parking principle, along the road can also be a good introductory example to illustrate recursions [26]. For example, there is a street 10 units long and we would like to know how many cars can fit into these 10 units. After a car has been parked in a random space, it occupies a single unit and it „divides” the street into two parts. The total number of cars parked is going to be: the number of cars that can be parked in front of the parked car, the number of cars that can be parked behind the car parked, increased by one parked car.
The author [27] describes an interesting example of introducing the recursion through examples from the computer world. Recursions have been described on the example of:

- Computer system of files and folders - going through the tree like structure of files and folders by using the `python os` module.
- Web searching – using `urlli .parse` module, by which we can reach each of the URLs on a single page and then recursively visit all those URLs.

Recursions are not natural in imperative languages [28], [29] whereas functional languages are stated to be the alternative.

Numerous works ([13], [30], [31]) deal with models of stack simulation, i.e. with visualisation of recursion tracing. To be able to understand recursions it is key to understand the principal of a subprogram execution termination, and transfer of the execution flow onto another subprogram [13]. In this context it is the moments that are important when a subprogram calls another subprogram, i.e. itself in a case of recursion. The author calls this moment active transfer of the execution flow. The moment the called subprogram has been executed till the end, the execution control returns to the subprogram that called it – execution flow passive transfer. These, but many other procedures have been visualised in the system (e.g. representer of program invocation). These authors’ work demonstrates that usage of this program improves understanding, and writing of recursive programs, as well.

Besides computer, recursion mechanism visualisation is possible on paper, too. [31] speaks of the recursion graph notion - `raph`. The idea for this kind of an approach has come to life as a reaction to the recursive tree, which, according to its author, doesn’t have an accentuated passive transfer and value return at the end of the recursive call; which is solved in `raph`. `Ware nodes` in the graph represent recursion call, while `oval nodes` represent preliminary actions for recursion call, i.e. actions after recursion call. Moreover, there are two types of edges: `dashed line` and `full line`. The dashed ones are those that go from the higher level onto the lower level, whereas the full lines represent the edges among the nods on the same level.

![Image](image.png)

**Picture 3: visualisation of execution of the nth element of the Fibonacci sequence using the RGraph**

Hybrid visualisation model is the one trying to unify more conceptual teaching models. One of such has been described by [12]. Their idea was to unify the conceptual model of stack simulation and structure templates. The result of it is creation of a system for helping students learning recursions, called ChiQat Tutor. Stack simulation model is based on the graphic variant of the `raph`. The system offers a few standard problems that enable studying of recursions (factorials, palindrome etc.), and provides recursions tracing, complementing of recursive algorithms, creating the RGraph for a set recursive code, animating of recursion execution, etc. After using this tool, a significant leverage in solving recursive problems has been observed.

C. **A strict order amplies**

Concrete model of recursion teaching is not enough for a deeper understanding of recursions [10], [32], [33]. Recursion execution tracing is the basic mechanism that is going to help the recursions-inexperienced students to be successful at computing the recursive function value, however that does not mean that they will be able to understand the recursion fully, and especially the passive transfer of execution control that is happening at the moment of our exiting the function’s recursive call.

Programming represents a specific way of thinking and addresses problem solving [10], where the focus is placed on figuring out algorithms, and not on the programming language syntax.

Algorithm to approach the recursive problem solving has been described by [10] and [11]. According to [10], the basic steps are:

1. **Induction hypothesis** – the smallest instance of the problem with a basic solution,
2. **Rules simplification** – manner to reduce the size of a problem,
3. **Natural recursion** – link between a simplified and an actual solution to the problem,
4. **Finish** – merging of the previous steps into a single recursive function.

V. **LOOPS BEFORE RECURSIONS OR VICE VERSA?**

There is also a debate on when to teach recursions. Several authors think that they should be taught after loops, whereas another group thinks that it is better to show the recursions before the loops.

The research according to which it is better to learn loops first, and then recursions, has been presented in [34]. On the other hand, [35] and [36] propose learning recursions before iterations. The authors [35] state several arguments to their hypothesis: recursion makes a special case of Divide Conquer rule method; to understand recursions it is enough to know functions and branching whereas to understand loops one needs to understand a random concept, the reason for not understanding recursions arises due to the created mental model of the loop.

VI. **CONCLUSION**

Most researchers will agree with the statement that, on one hand, recursions make an extremely important concept in computing, and an exceptionally complex one for beginners, on the other hand. There are various methodologies for teaching recursions, which can generally be divided into concrete and abstract. One group of researchers point out that concrete methods are better, whereas another group states that one should use abstract methodology in order to understand recursions fully. The truth is, surely, somewhere in the middle. One should use various concrete methods, their visualisations, etc., to be able to understand the recursion mechanisms. However, this is not the way the students will develop a recursive manner of thinking. They will develop it only if they have a deep knowledge of recursions, if they will be using them in various situations during their education, if the teacher has the habit of leading them into thinking could the problem on the table be solved recursively, and so on.
BIBLIOGRAPHY


[32] T. Scholtz and I. Sanders, »Mental models of recursion: how much should we emphasize the computing environment — how much should we emphasize the computing model?,« in *TIC*, vol. 1, no. 1, pg. 42-50, 2007.


