Common Errors Related to Recursive Functions and Visual Debugger Assistance

RAMI RASHKOVITS, LAVY ILANA
Information Systems, Max Stern Yezreel Valley College
ISRAEL
ramir@yvc.ac.il; ilanal@yvc.ac.il

Abstract: - Recursive Functions are one of the difficult concepts learned by computer science (cs) students, and applying a correct recursive solution to a problem is even more difficult. The difficulty lies in the recursion concept, in which the solution to a given problem is based on the solution of the same problem with smaller size. Humans do not usually think in such a way, and they prefer to solve problem iteratively whenever possible. However, many problems are better solved recursively in terms of simplicity and complexity, hence developing recursive thinking is crucial for programmers. During their studies, students learn this issue using metaphors such as the 'little man' or the 'top-down frames', but these metaphors are not very useful when applying a recursive solution. In this paper we provide the students with an interactive tool in which recursive solutions are visualized using frames, trees and graphs, and test the quality of their solutions using these tool, compared with a control group in which the visualizer is not available. We planned an experiment with few problems requiring recursive solutions, and observed the experiment group and the control group. The results show that the tool indeed improves significantly the quality of the solutions of the students who used it.

Key-Words: - Problem Solving, Recursion, educational technology.

1 Introduction

Recursion has always been one of the most difficult concepts to understand and apply by computer science students. While typical algorithm has straightforward and trackable steps to follow, a recursion algorithm is built in a way that in order to solve a problem, one has to solve the same smaller-scale problem up until the problem becomes very simple that a solution can be provided without further calls to smaller problems. Once the solution to the simple problem is return, it is possible to solve the higher-scale problem which in turn enable the solving of higher-scale problem and so on until the original problem can be solved. The recursive algorithm is much less intuitive, and the reader has difficulties to track its steps [1,2]. Recursive solutions are essential in the field of computer science, and many times a problem can be solved only using such an algorithm (i.e., Hanoi towers), and therefore understanding well the concepts involved, and being able to plan and apply correctly recursive algorithm is an obvious goal of introductory course in computer science.

In order to overcome the above difficulties, few metaphors were developed to assist the learner to understand the execution of recursive algorithms, among them are the little-man metaphor [3], and the frame model [4]. These visual metaphors demonstrate the advance process of a recursive function by illustrating the recursive call as a package delivered forth and back from one little man to the next one in the chain (e.g., little-man model) or as series of frames each located inside a larger one. Indeed, these metaphors were found to be quite effective in explaining the way linear recursive functions behave. However, not all recursive algorithms are linear (i.e., form a simple chain of recursive calls), and there are many multi-dimensional recursive algorithms which form complex non-linear chains of recursive calls. Since the above models are linear, they cannot be adapted to more complex forms of recursion (e.g., Inorder tree traversal).

In this study we developed an interactive software tool that enhances the understanding of recursion concepts (linear and non-linear) by tracking the recursive calls visually, running them step by step, tracking variables and return values of each call, and continue running until the algorithm stops. In addition, we examined the tool's effectiveness as perceived by the students who participated in the research.

2 Background

Recursive functions can be linear or multi-dimensional. The most common recursive functions are linear ones, in which the function makes a single call to itself each time it runs. The factorial function appears in Figure 1 is a good example of such a function. In some cases, as shown in Figure 1, the recursive call is the last command in the functions (called tail recursion). In other cases, as shown in Figure 2 (reversing an integer number) there are more commands to be executed after the recursive call returns with or without a value. A double recursion is shown in Figure 3 (calculating a Fibonacci number), in which multiple recursive calls are made. A more complex form of recursion is indirect recursion, in which a function f does not call itself, but rather call another function g, which in turn calls yet another function k, that calls f again. Such a mutual recursion is shown in Figures 4 and 5, where two functions *is_odd()* and *is_even*() that are mutually call each other.

```
int factorial(int n)

    if (n==1) return 1;
    else return
    num*factorial(num-1);
```

Fig. 1. Tail Linear Recursion

```
void reversePrint(int n)

    if (n<=0) return;
    reversePrint (n/10);
    System.out.print(n%10);
```

Fig. 2. Non-Tail Linear Recursion

```
int fibonacci(int n)

    if (n<=1) return n;
    else return fibonacci(n-1) +
    fibonacci(n-2) ;
```

Fig. 3. Double Recursion

```
boolean is_even( int n )

    if (n == 0) return true;
    else return odd(abs(n)-1)
```

Fig. 4. Mutual Recursion (part 1)

```
boolean is_odd( int n )

    if (n == 0) return false;
    else return even(abs(n)-1);
```

Fig. 5. Mutual Recursion (part 2)

The little-man metaphor [3] and the frames model [4] are effective when tail linear recursion is discussed. The factorial algorithm is demonstrated with the little-man metaphor in Figure 6, and with the frame metaphor in Figure 7 for the input value *n=4*. As shown, the learner sees an illustration of the recursion, and able to track its steps. However, given more complex linear recursions (e.g., non-tail), multi-dimensional recursions (e.g., double, multi), not to mention indirect recursion (e.g., mutual), these models would not promote the learner with understanding of the functions' behavior.

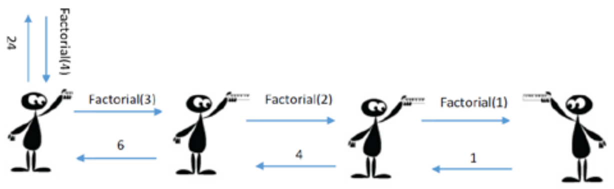


Fig. 6. Little Man Model

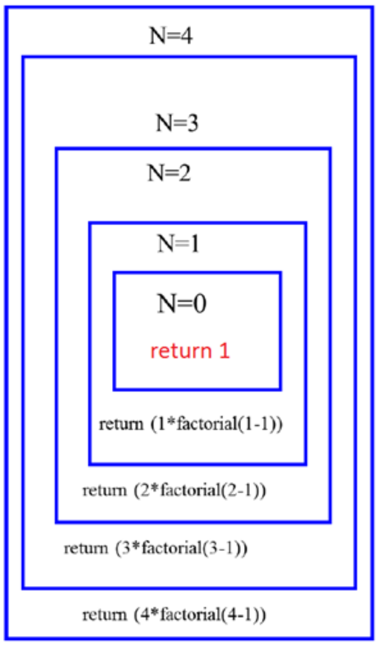

N=4
N=3
N=2
N=1
N=0
return 1
return (1*factorial(1-1))
return (2*factorial(2-1))
return (3*factorial(3-1))
return (4*factorial(4-1))

Fig. 7. Frame Model

3 Related work

Various teaching strategies were suggested and recommended in the literature as to recursion algorithms, starting with recurrence relations from the theory of mathematical inductions [5,6], through concur-and-divide methods [7], and even algebraic substitution techniques [8]. However, experiments have shown that concrete conceptual models assist learner better than abstract ones [9]. The use of visualization technology in class has made a great impact on learners, and promoted significantly the understanding of recursion concepts [10]. Sa & Hsin [11] have developed RGraph, a tool that visualizes a recursive function calls, forth and back. A tutorial on recursion exploration based on RGraph was developed and used to teach recursion with initial encouraging results about better understanding [12]. However, RGraph is currently a tool with a few pre-defined problems, all of them are linear. It does not enable the learner to run and explore user-defined recursive functions, neither it supports the visualization and understanding of more complex recursive functions (e.g., multi-dimensional and/or indirect recursions).

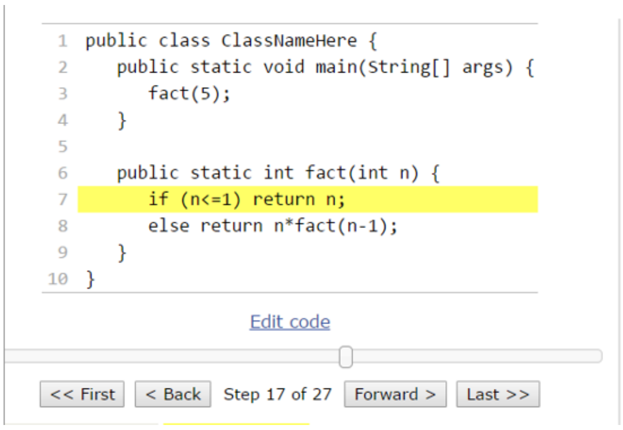
4 The Study

A new and novel tool was developed, aiming to provide learners and developers with an interactive environment for the exploration of recursive functions of all kinds. After the completion of the development process, we plan to examine its effectiveness as regard to the understanding and implementation of recursion concepts in problem solving as perceived by both the students and the teaching staff. Then, we plan to build a tutorial, which is based on the implementation of the tool in introductory computer science course and advanced data structures and algorithms courses.

4.1 The tool

The tool operates in a similar fashion to software development environment (e.g., Eclipse, Visual Studio). The user writes a recursive function/s (See Figure 8), and run it using the tool, while providing the necessary initial inputs. Once the function has been compiled successfully (using background processes) the user will be able to control its running, in a similar fashion to typical debugging. The user is able to trace the program step-by-step, back and forth, and explore its variables. In addition to standard debugging, the user will be provided with the opportunity to track the function calls visually.

Each recursive call will open new icon on the screen with all the information relevant to the exploration of this call: parameters and the current state of the call, the value returned, the line of code that was executed and the recursion depth.



```
public class ClassNameHere {
   public static void main(String[] args) {
      fact(5);
   }

   public static int fact(int n) {
      if (n<=1) return n;
      else return n*fact(n-1);
   }
}
```

Fig. 8. Environment

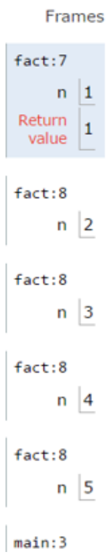


Fig. 9. Frames

In Figure 9 we can see the result of running the *factorial* function with *n*=5. The first (lowest) frame refers to the main method, calling the *fact()* function on line 3, the frame above refers to the first call to *fact()*, with *n=5* as a parameter. The subsequent frames refer to the successive calls to *fact()* till the last call to *fact()* with *n=1* (base case). The user can track the recursion, and whenever a new recursive call is made a frame with all the necessary information required (i.e., current line, parameter value, calling functions).

The above frames can address linear recursion when each function calls itself at most once. However, for more complex recursions such as double or mutual recursions, the linear representations of the frames as shown in Figure 9 might not be sufficient. For these kind of recursion, we provide a more sophisticated visualizer, in which the hierarchical structure of the recursion is revealed.

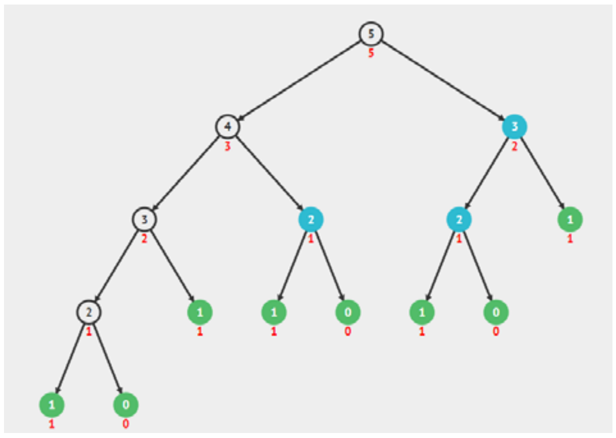


Fig. 10. Tree-like structure

In Figure 10 we see the result of running the *Fibonacci* function (shown in Figure 3). In this function, two recursive calls are made from each function calls. A tree-like structure is more trackable, as shown in figure 10. Each node represents a function call, with the value of the parameter inside, and the return value below. Another way to track the recursion is available via graph-like representation, as shown in Figure 11, in which calls to similar copies (a function call with identical parameter values) are shown as incoming edges, enabling the user to better track the complexity of her recursion.

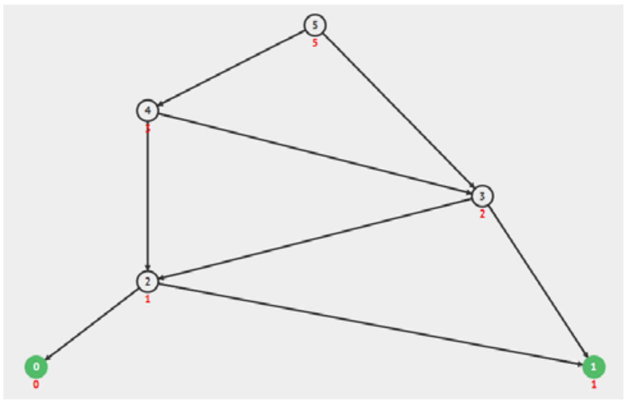


Fig. 11. Graph-like structure

The tool was developed in a web-based environment. It provides the user with information about the number of recursive calls, enabling her to estimate the complexity of the recursive function. The output is presented graphically, plotting the recursive calls for each input size. The output is shown gradually, not all at once. This way the user can explore the code along with the output nodes, tracking thoroughly the recursion.
For instance, if the user run the Fibonacci function (see Figure 3) with initial input of n=5, the diagram will plot for every recursive call the number of recursive calls derived: for n=0 and n=1 the number of calls is zero, for n=2 it is two, for n=3 it is three, for n=4 it is five, and last for n=5 it is eight. Actually, in this example, as the input size rise, the number of derived recursive calls grows exponentially, and the user is able to view this complexity via the graphical diagram.
The visualization process start with analysis of the input function, embedding breaking commands inside the function that enables the debugging operations, tracking and saving the current call's state, and managing the whole running of the recursive function.

4.2 Environment and population

We tested the tool in the course "data structures and algorithms". The study subjects were Information Systems (IS) students in their second year of studies in a regional academic college. 78 students participated in the courses, divided into two lecture-groups.

4.3 Data collection and analysis tools

As regards to the examination of the tool's effectiveness, we used an empirical comparative study in which two groups were involved. The students were divided into two equal-size groups. The experimental group study recursion using the tool, while the control group study recursion using classical methods (e.g., frame model, little-man model) and a standard IDE. Both groups were presented with the recursion problems presented in figures 1-3. The experiment group were presented with the tool we developed, and the students could run the solutions using the debugger, while exploring the solutions using the visualization shown in figures 9-11.
After studying the recursion concepts, all students from both groups were given a series of problems that require recursive solutions. We expected that students who learned recursion using the proposed tool will be able to perform better than the students from the control group given that they were permitted to use the tool while solving the given problems. During the solutions we were observing the students to see whether and how they use the tool, and we asked them to report on their use while solving each of the given problems.
When we checked the solutions, we divided them into the following four categories: correct solutions, faulty base cases, faulty recursive call, and faulty return command. Solutions to problems that work perfectly on any legal input were classified as

correct ones. Solutions with base case other than expected, even partially correct, were classified as faulty base case. Solutions that had problems with the recursive call (e.g., incorrect parameters) were classified as faulty recursive call. Solutions with errors in the return command (e.g., return too early) were classified as faulty return command.
After checking the solutions, we also made observations and interviews with selected participants, in order to gain better understanding of the tool advantages and shortcomings. With these essential feedbacks, we intend to further improve the tool and add desired functionality.

4.4 The problems

The students were provided with the following three problems:

(1) Calculate recursively the sum of the first *n* integers, *n* is given as a parameter. For instance *sum(5) = 5+4+3+2+1 = 15.* Assume non-negative *n.*
(2) Reverse a string recursively. For instance, *reverse("hello") = "olleh".* Assume non-empty string.
(3) Given the formula given in Figure 12, calculate recursively how many combinations there are when choosing k elements out of a set of n elements. Assume non-negative *k* and *n*.

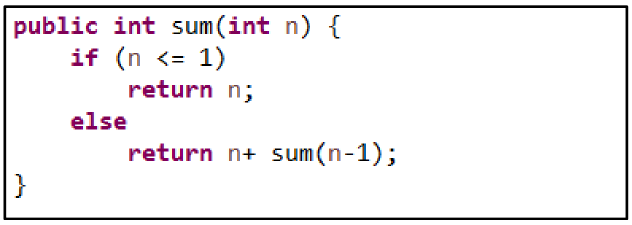
The three problems above were given with increasing difficulty, addressing tail recursion, non-tail recursion, and double recursion, respectively.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \le k \le n-1$$

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers } n \ge 0$$

Fig. 12. K out of N formula

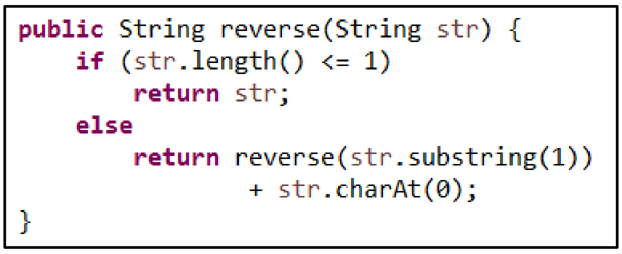
The correct solutions for these problems are given in figures 13-15.



```
public int sum(int n) {
    if (n <= 1)
        return n;
    else
        return n+ sum(n-1);
}
```

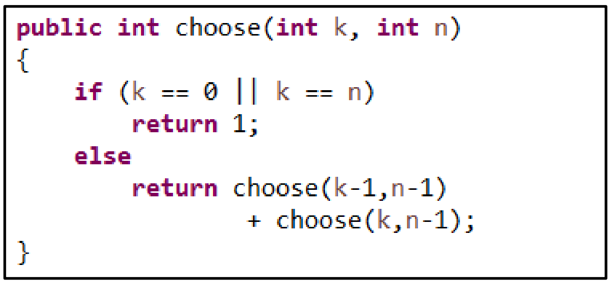
Fig. 13. Problem 1 solution

Both groups, were allowed to use the regular IDE (Eclipse Neon) to write and test their solutions. The experiment group was provided also with a link to a web page in which the tool presented above was implemented. They were told that if they want they can use the tool while developing solutions to the given problems. They were given 60 minutes to address the problem, and were instructed not to consult with each other. Also, in order to prevent cheating, we took all cellular phones, and blocked all network communication except the debugger web page.



```
public String reverse(String str) {
    if (str.length() <= 1)
        return str;
    else
        return reverse(str.substring(1))
                + str.charAt(0);
}
```

Fig. 14. Problem 2 solution



```
public int choose(int k, int n)
{
    if (k == 0 || k == n)
        return 1;
    else
        return choose(k-1,n-1)
                + choose(k,n-1);
}
```

Fig. 15. Problem 3 solution

4.5 Results

In this section we first present summative results, comparing the experimental and the control group achievements. Then we present common errors performed by the study participants, and how the tool provides them with assistance.

4.5.1 Summative Results

A summary of the results is shown in Table 1. As expected, most of the participants were able to provide a correct solution to the first problem. Since it was very simple, one could address the problem without using a debugger. As to the second problem, we observe a decrease in the number of the students who provided correct solutions. This is also expected as the solution is not so simple, and it requires an understanding of the recursion structure. In the third problem we see an increase in the number of correct solutions, probably because this problem was provided with a formula, which can be translated easily to a recursive method. When comparing the results of the experiment group and the control group we observe that the experiment group outperformed the control group in all three problems. We also see that the as the problem gets harder, the difference is more notable. While in the first problem there is a difference of 2% in the number of correct answers, in the second problem there was a difference of 17%, and in the third problem 25% difference. The participants of the experiment group indeed used the tool extensively. All of them were using the tool to solve the second and third problems, while only 52% of them have used it also with the first problem. As to the control group, only 3 out of 38 sketched some kind of frame model or little man model to monitor their solutions.

Table 1: Percentage of correct answers

Problem	Experiment group	Control group
1	85%	83%
2	62%	45%
3	77%	51%

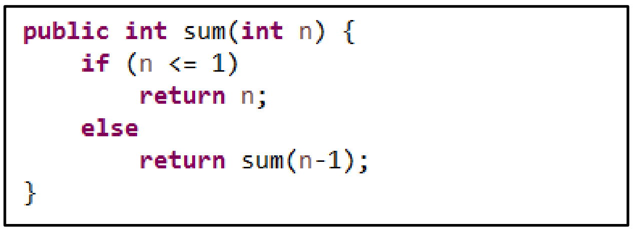
The percentages of errors according to these types are presented in table 2. As shown, in the experiment group the percentages of errors referring to base cases, and return commands is lower than the control group, while the percentages of recursion calls category is higher in the experiment group.

Table 2: Percentage of errors' types

Error	Experiment group	Control group
Base case	19%	34%
Method call	58%	45%
Return	25%	21%

4.5.2 Common Errors

As to the first problem, the most common error was a faulty recursive call neglecting the addition of n to the returned value, as shown in Figure 16.

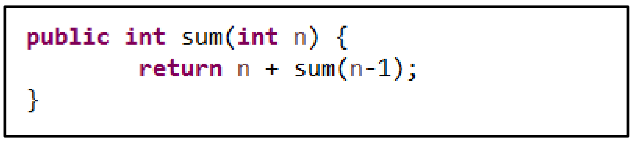


```
public int sum(int n) {
    if (n <= 1)
        return n;
    else
        return sum(n-1);
}
```

Fig. 16. Problem 1 – faulty return

Students from the experiment group who used the interactive debugger could follow frames (Fig. 9) the tree-like structure (Fig. 10) and observe the return values underneath the nodes, identify the bug, and fix it. Students from the control group could only notice the problem if they tested their solution using the standard IDE, but did not had any clues regarding the bug's source.

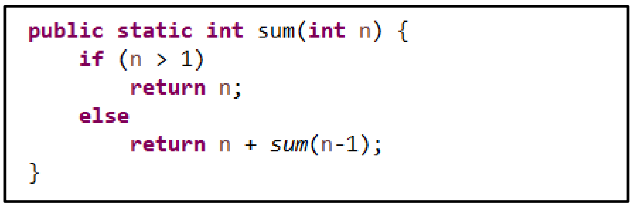
The second common error related to the first problem was a missing base case, as shown in Figure 17.



```
public int sum(int n) {
        return n + sum(n-1);
}
```

Fig. 17. Problem 1 – missing base case

Students from the experiment group who used the interactive debugger could observe the message that the debugger cannot run the recursion since it is too deep. Students from the control group could also observe the 'Stack Overflow Error' raised by the IDE, and fix the problem. The third common error related to the first problem was a faulty base case, as shown in Figure 18.

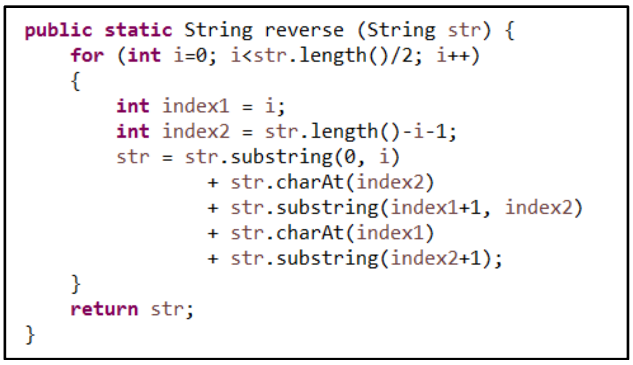


```
public static int sum(int n) {
    if (n > 1)
        return n;
    else
        return n + sum(n-1);
}
```

Fig. 18. Problem 1 – faulty base case

Students from the experiment group who used the interactive debugger could track the frames or the tree and see that only one call was performed before the recursion stopped with faulty results. Students from the control group could only observe a faulty result, however, they could not see the fact that only one call to *sum()* was made.

The solution to the second problem was a bit more complicated than the one to the first problem. It involves decomposition and assembly of the input string each call. Many participants failed to provide a proper solution, some of them provided iterative one as shown in Figure 19, and then tried to convert it into recursive one demonstrated in Figure 20.



```
public static String reverse (String str) {
    for (int i=0; i<str.length()/2; i++)
    {
        int index1 = i;
        int index2 = str.length()-i-1;
        str = str.substring(0, i)
                + str.charAt(index2)
                + str.substring(index1+1, index2)
                + str.charAt(index1)
                + str.substring(index2+1);
    }
    return str;
}
```

Fig. 19. Problem 2 – Iterative Solution

```java
public static String reverse (String str, int i) {
    if (i == str.length()/2)
        return str;

    int index1 = i;
    int index2 = str.length()-i-1;
    str = str.substring(0, i)
            + str.charAt(index2)
            + str.substring(index1+1, index2)
            + str.charAt(index1)
            + str.substring(index2+1);

    return reverse(str, ++i);
}
```

Fig. 20. Problem 2 – Recursive Solution

Among all the students who provided a recursive solution similar to the one shown in Figure 20, only few succeeded to complete a correct version. All other students made various mistakes while converting the iterative solution to a recursive one. Some of them did not add *i* as a parameter to the method, some other made concatenation errors.

The solution to the third problem was a bit more complicated than the one to the second problem, since it includes two recursive calls. Many of the students who provided faulty solution did not understand that the recursive function must have both *k* and *n* as parameters, and neglected one of them. Some others could not figure out what the base case should look like, and provided a faulty one. some others forgot to indicate a base case or provided a faulty one, and some others made mistakes while invoking the recursive call referring to the parameters sent**.**

4.5.3 Tool assistance

We were watching the students while they solved the problem, and specifically we tracked the experiment group to see if and how they use the interactive debugger. As we expected, almost all of them indeed used the tool to run their solutions and test them for correctness. The first problem was simpler than the other two, therefore merely half of them did not use the interactive debugger at all. Among those who did use the tool, almost everyone was satisfied with the 'Frames Visualization' (See Figure 9) and only few tried to run it with the more complex visualization modes. Some of them noted that their implementation is faulty while watching the frames and immediately fixed the code until run correctly. As to the second and third problems, all the students in the experiment group used the tool. In the second problem most of them used the 'Frames visualization' while few also tried the 'Tree Visualization' (See Figure 10) although it did not add much more information. However, while solving the third problem, many of the students have used both 'Tree Visualization' and 'Graph Visualization' (See Figure 11) to test their solutions. Among the ones who provided correct solution to the second and third problems, there were many who ran the above models many times, until getting to the correct version. The visual feedback assisted them to identify their errors (faulty base case, faulty recursive call, etc.) and fix them before handing over the solutions. The students from the experiment group who did not provide a correct solution to the second and third problems also used the tool to explore their solutions, but nevertheless they were not able to fix the errors completely. Many of these students did not design a proper function, and as a result the visualization did not help the, much. For instance, if the function designed to address the second problem did not include *i* as a parameter, the debugger will not highlight the problem. Same for the third problem, when the function has only one parameter for *k* or *n*. The tool assists only solutions who are 'close enough' to the correct solution. If the student did not get the idea of the required recursion, and as a result design a faulty function (i.e., faulty signature), then the tool cannot assist.

As to the control group, they merely used the metaphors of the 'Little Man' (See Figure 6) or the 'Frame Model' (See Figure 7) while solving the given problem. Most of them ran their solutions in the IDE and tested the final result shown on screen. If it was correct the moved on, otherwise they fixed the code accordingly. While fixing the code some of them used the IDE's built-in debugger who can track the recursion via step-by-step commands enabling tracking the parameters values on each call. However, such a method requires focusing on the debugging process, remembering the values of previous calls, and complex track of returning values. Although possible, it takes much more time to follow a solution in this way, and we see that the success percentages of the control group are significantly lower than those of the experiment group.

4.6 Interviews

After the completion of the assignment described above, we conducted interviews with five students from the experiment group, that were observed while making intensive use of the tool. We asked them to describe the benefits it provided them. We

also asked about their criticism on the tool and asked for suggestions to improve it.
In what follows we provide few excerpts given by the interviewees.

4.6.1 Benefits the tool provides

" The tool made for me a visualization of the recursive process. Without it, it is more difficult for me to follow the development of the recursion and the logic involved. "

" What I loved in the tool is the ability to track the hierarchy of the recursion calls, and to follow the return values. That was very helpful. "

" The tool helped me find an infinite recursion I made by mistake. It just didn't run… It took me only a while until I noticed the error. "

"I used the graph-like visualizations when I solved the third problem. I think that the solution I gave was correct but not very efficient. Many nodes had plenty of incoming edges. I tried to think of a better solution but I ran out of time. "

"running the recursion in a step-by-step manner, forward and back, while watching all the recursion calls on screen, including the calls that were already ended, was of a great value. "

4.6.1 Improvement suggestions

"I would like to have these abilities in the regular IDE I'm using. It can help a lot when solving recursion problems. "

" I would like to add a conditional breakpoint, so I will be able to stop the running and watch the current state visually upon the case I want to explore. Now I have to run it step-by step. "

" You should consider hover-event over the nodes, so that if one passes over a node, the relevant line of code will be painted. "

" I would add statistics to each node, for instance how long did it take from the start until return, how many calls with the same values occurred, and alike. "

4.7 Discussion

The results presented in section 4.5 support our assumption that a visualizer tool can effectively improve the understanding of students concerning recursion concepts. The results show that if visualization is used, the results are better and there are fewer errors. Moreover, the results show that regarding to base cases and return parts of the recursion, fewer mistakes are made by the students, as the visualizer make it more easy to capture such errors. The fact that only 3 participants from the control group have tried to draw the recursion call's hierarchy indicate that in the absence of a visualization tool, the student will not make an extra effort to visualize the solution, and accordingly the number of faulty solutions grow.
From the participants' excerpts we learn that indeed the tool was helpful. Recursion is an abstract concept, and many students find it very difficult to understand. Visualization has always been [13] a mean to improve the understanding of complex concepts, including recursion algorithms. It assists the user to track the calls, the logic behind the recursion, the convergence towards the base cases, and the process of returning from the recursive calls. It even helps one who cares about the complexity of the algorithm (depends on the number of repeating calls).

Based on the students' suggestions for the tool improvements, we plan to make few changes to make the tool even better, and then we intend to build a tutorial on recursion teaching, based on the tool and its exploration capabilities. The tutorial will include complete lessons that can assist educators with the instruction of all related issues including linear and tail recursion, double and multi-dimensional recursion, direct and indirect recursion, recursive calls, base condition, running a recursion forth and back etc. We believe that using our tutorial will contribute to the understanding and the ability to apply recursive solution among students and learners, and we also believe that such a tool can be valuable as well to practitioners in the industry when testing and debugging complex recursive algorithms in various fields (e.g., computational biology, machine learning, enterprise systems etc.)

5 Conclusions

To address students' difficulties to implement recursive algorithms in problem solving relating to programming, we developed an interactive tool that enable to run and debug recursive functions and track them visually. The tool enables tracking of user-defined, direct and indirect, linear and multi-dimensional recursive functions. We tested the tool empirically, and our findings support our assumption that a visual debugger for recursive

algorithms might assist in understanding better recursion and promote higher-quality solutions with fewer errors.
In the future, we plan to expand the tool further with features related to multi-thread recursion and test it in additional academic institutes, as well as in the industry.

References:

[1] Gal-Ezer, J., & Harel, D. 1998. "What (else) should CS educators know?". Communications of the ACM, (41:9), pp. 77-84.

[2] Dann, W., Cooper, S., & Pausch, R. 2001. "Using visualization to teach novices recursion". ACM SIGCSE Bulletin, (33:3), pp. 109-112.

[3] Harvy, B. (1985). Computer science Logo style. Volume 1: Intermediate programming. MIT Press.

[4] Roberts, E. 2006. Thinking recursively with Java. Hoboken, NJ: John Wiley.

[5] Ford, G. 1984."An implementation-independent approach to teaching recursion". ACM SIGCSE Bulletin, (16:1). pp. 213–216.

[6] Wilcocks, D., and Sanders, I. 1994. "Animating recursion as an aid to instruction". Computers & Education (23:3) pp. 221-226.

[7] Ginat, D., and Shifroni, E. 1999. "Teaching recursion in a procedural environment—how much should we emphasize the computing model?". ACM SIGCSE Bulletin (31:1), pp. 127-131.

[8] Lewis, C. M. 2014. "Exploring variation in students' correct traces of linear recursion". In Proceedings of the tenth annual conference on International computing education research. pp. 67-74. ACM.

[9] Wu, C. C., Dale, N. B., & Bethel, L. J. 1998. "Conceptual models and cognitive learning styles in teaching recursion". In ACM SIGCSE Bulletin (30:1), pp. 292-296.

[10] Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. 2002. "A meta-study of algorithm visualization effectiveness". Journal of Visual Languages & Computing, (13:3), pp. 259-290.

[11] Sa, L., & Hsin, W. J. 2010. "Traceable Recursion with Graphical Illustration for Novice Programmers". InSight: A Journal of Scholarly Teaching (5), pp. 54-62.

[12] AlZoubi, O., Fossati, D., Di Eugenio, B., Green, N., Alizadeh, M., and Harsley, R. 2015. "A Hybrid Model for Teaching Recursion". In Proceedings of the 16th Annual Conference on Information Technology Education. pp. 65-70. ACM.

[13] Dann, W., Cooper, S., & Pausch, R. 2001. Using visualization to teach novices recursion. ACM SIGCSE Bulletin, 33(3), 109-112.