Secondary education students' difficulties in algorithmic problems with arrays: An analysis using the SOLO taxonomy

Euripides Vrachnos, Athanassios Jimoyiannis
evrachnos@gmail.com, ajimoyia@uop.gr

Department of Social and Educational Policy, University of Peloponnese, Greece

Abstract. Developing students' algorithmic and computational thinking is currently a major objective for primary and secondary education in many countries around the globe. Literature suggests that students face at various difficulties in programming processes, because of their mental models about basic programming constructs. Arrays constitute the first data structure students have to cope with in introductory programming courses. This paper presents the results of an empirical study on secondary education students' misconceptions and mental representations of the array data structure. Students' responses to written tasks regarding short code segments were mapped to the different levels of the SOLO taxonomy, in order to identify how students use arrays to solve programming problems. The analysis of the results showed that the majority of the students tended to manifest responses assigned to the lower SOLO levels, i.e. prestructural, unistructural and multistructural. The findings indicate that many students in the sample had incomplete or faulty representations of the array concept, which seem to be connected to their misconceptions about the programming variable concept.

Keywords: arrays, introductory programming, misconceptions, mental representations, SOLO taxonomy

Introduction

In the recent years a growing debate is evolving among academics, educators and educational policy authorities, around the world, with regards to significant reforms in secondary education curricula aiming to enhance students' algorithmic and computational thinking, as well as their problem solving abilities when using various programming tools and environments. Furthermore, recent reports, published in Europe and the USA, suggest that the students should be exposed to both, the principles of computer science and the fundamental concepts and procedures of programming (ACM, 2013; CSAT, 2011; Informatics Europe & ACM, 2013). In this context, new standards and pedagogical approaches for programming and algorithmic thinking, have been introduced in the K-12 curricula; for example, in Australia (ACARA, 2013), England (Department for Education, 2013), Greece (National Curriculum, 2011) and New Zealand (Bell et al., 2012).

Introductory programming and algorithms are generally considered as demanding tasks for both, teaching and students' learning perspectives. Existing research has revealed many important aspects of students' programming thinking. Studies on students' programming ability have shown that they have difficulties in designing algorithms as well as in writing and tracing programs (De Raadt, 2007; McCracken et al., 2001; Robins, Rountree & Rountree, 2003). Novice programmers face at difficulties to grasp basic programming concepts, such as variables, conditional structures and loops. It seems that students find very difficult to manipulate abstract programming entities, e.g., logical data structures, nested IFs, loops and counter initialization, recursion etc. In addition, these constructs have limited relation to students' pre-existing knowledge or everyday experience (De Raadt, 2007; Jenkins, 2002; Kaczmarczyk et al., 2010; Lahtinen et al., 2005).

One of the most influential books, edited by Soloway and Spohrer (1989), has provided a sound theoretical and research orientation for studying students' common misconceptions and mental models about fundamental programming constructs. Following, a range of studies have been conducted regarding, for example, the variable concept and the assignment statement (Jimoyiannis, 2011; Ma et al., 2011), the conditional and loop structures (Sajaniemi & Kuittinen, 2005; Bonar & Soloway, 1985), and the recursion concept (Putnam et al., 1989). In a more recent study, Sirkiä & Sorva (2012) used visual programming exercises to identify a wide variety of programming misconceptions held by university students studying computer science.

As a common conclusion, students appear to have faulty or fragile mental models about programming constructs, objects, attributes and methods (Eckerdal & Thune, 2005; Garner, Haden & Robins, 2005). While expert programmers are able to develop algorithmic thinking mechanisms and models, novices usually show surface understanding and misconceptions about fundamental domain concepts (Madison & Gifford, 2003; Ma et al., 2011; Jimoyiannis, 2011; Lister et al., 2009). The majority of the students exhibit poor performance in using effective strategies, even when they try to solve elementary programming problems. They lack the skills necessary to function in an abstractive way, to consolidate a program as a single entity, to authentically compose new algorithms and to effectively adapt statements or procedures that harness their previous programming knowledge (Jimoyiannis, 2011).

In addition, a significant number of investigations have been directed to misconceptions that are related to object-oriented programming concepts, such as objects, classes, methods, constructors etc. (Boustedt, 2012; Fleury, 2000; Holland et al., 1997; Hristova et al., 2003; Ragonis & Ben-Ari, 2005). However, only few of them were focused on students' difficulties and misconceptions about data structures, such as heaps and trees (Seppälä, 2006; Danielsiek, 2012; Wolfgang & Vahrenhold, 2013; Karpierz & Wolfman, 2014).

Investigating how students transform their ideas about programming constructs, from concrete representations to abstract mental models, is still an open research topic in computer science education (Cortney et al., 2011; Lister et al., 2009; Ma et al., 2011). In addition, addressing new instructional approaches that could facilitate the development of students' algorithmic thinking, is another issue of major interest in computer science education. In particular, searching novices' ability to use abstract concepts, like arrays and data structures, in programming problems is still a research topic of interest (Ma et al., 2011; Lister et al., 2009; Sheard et al., 2008).

In conclusion, literature review has indicated that research findings regarding students' difficulties and misconceptions about the array concept are rather limited. This survey was designed to extend previous results and provide new information on how secondary education (K-12) students typically approach short code programs with arrays. At the same time, we looked into factors that may influence students' representations and effective use of arrays to solve programming problems. The students' written responses to programming problems were mapped to the different levels of the Structure of the Observed Learning Outcome (SOLO) taxonomy, which proposed by Biggs and Collis (1982).

Thus, the following questions were used to guide our research:

1. What are students' dominant representations about the array concept? Are there any critical misconceptions regarding the array concept?

2. To what extent the participant students have developed their ability to solve algorithmic problems with arrays? Can they identify the type of problems that require the use of the array data structure?
3. To what extent SOLO taxonomy offers a coherent framework to analyse students' representations of the array concept as well as their ability to use arrays in order to solve programming problems?

The research findings showed that the students participated in the present study had incomplete and faulty representations of the array concept, which determined students' performance and their difficulties to use the array structure in solving simple programming problems. An interesting finding, reported for the first time according to our knowledge, was that students' misconceptions and conceptual barriers with regards to arrays appeared to be originated in faulty representations about the variable concept. The paper concludes with suggestions for educational practice and further research in introductory programming with regards to the concept of arrays.

Theoretical Background

Arrays misconceived

Teaching computer programming to novices includes many more things than the syntactic details and the semantics of the specific programming language used. What differentiates expert and novice programmers is that the latter need to think about algorithms and data in ways that are different to other subject areas in the curriculum (for example, mathematics or physics). The students, as novices, need to manipulate many abstract entities that have no or little relation to their experience and pre-existing knowledge (logical type, arrays, data structures, recursion etc.). This justifies why students have many difficulties when they try to express algorithmic solutions that do not come spontaneously, i.e. as a natural consequence of abilities and knowledge transfer from other cognitive areas to algorithms or specific programming environments (Jimoyiannis, 2011; Robins, Rountree & Rountree, 2003).

Another source of major difficulties is related to students' representations about the role of the machine (computer environment) during the program execution. Du Boulay (1986) introduced the term of *'notional machine'* to provide a theoretical concept describing the abstractive role of the computer in programming. The notional machine needs to be simple and to include concrete tools that allow students to overcome their faulty or incomplete perceptions regarding the computational machine and its internal operation during the execution of a program (Robins, Rountree & Rountree, 2003; Sorva, 2013).

In computer programming, an array is an indexed set of data elements that are stored in successive locations in the computer memory. Despite that arrays are the most widely data structure used by students in introductory programming, existing research findings indicate that their performance is particularly problematic, even in typical procedural languages (Du Boulay, 1989; Garner, Haden & Robins, 2005). A survey on the ideas of computer science educators has shown that loops and arrays constitute the most difficult programming concepts for novice students (Dale, 2006). Students usually fail to grasp in an integrated manner the key data structure concepts related to arrays; for example, array indexes, array elements, element value, array declaration, initialization statements etc.

Therefore, only a short number of investigations have been directed to students' mental representations and misconceptions about arrays. Detailed description of student programming abilities in problems with arrays is less common and the underlying factors/causes of learning difficulties are not discussed in detail.

An array is a homogeneous data structure, i.e. it consists of data elements of the same type. In addition, an array is a random-access data structure. In order to denote an individual array element, the name of the entire structure A is augmented by the index i that indicates the particular element (A[i]). In an early study, Du Boulay (1989) has identified that students' misconceptions in arrays were related to confusion between the index i of the array element and the value of the array element A[i]. In other words, the students had difficulties to differentiate between the concepts of *index* and *content* of an array element. In addition, Du Boulay (1989) pointed out that many students are not able to realize that the expressions a1, a[1], a[i] are conceptually the same construct, i.e. a single variable.

Student difficulties and misconceptions, similar to the concept of array index, were also reported with regards to the concept of pointer in C, which is also related to the notional machine, i.e. a memory location (Adcock et al., 2007; Craig & Petersen, 2016).

Du Boulay (1989) also reported student difficulties when an integer array element is used as an index in an array cell. For example, in the following assignment command

$$A[4] \leftarrow A[A[3]] + 5$$

the value of the third element A[3] is used as an index in the integer array A. In this case, the critical issue to be conceptualised is that an array element can be used as a single variable.

The theoretical foundations underpinning the present study were anchored in constructivist approaches of learning, which are expected to support students' development of programming skills and computational thinking (Ben-Ari, 2001). Programming is a constructive activity by nature as well. In this context, learning is thought as a contiguous process of refinement and extension of students' prior programming knowledge and, consequently, building coherent and viable mental models of the programming constructs. Therefore, the hypothesis that directs this particular study is that new knowledge (i.e. the array concept) is built upon prior knowledge (i.e. the variable concept) since an array is a sequence of variables. Arguably, students' misconceptions about variables are expected to be inherited or transferred to the array concept.

A previous study in Greece, revealed that the majority of K-12 students have a fuzzy representation of the concept of programming variable, which is rooted in their conceptions of the mathematical variable. Therefore, they difficulties to understand the main difference between the two concepts, i.e. a programming variable can hold only one value at a time. A common misconception among students is that *"a variable can store more than one values at a time"* or that *"a variable can 'remember' the history of the previous values"*. This faulty mental model has been described as the 'box' or 'stack' model of the programming variable concept (Jimoyiannis, 2011).

Effective use of variables is fundamental to students' achievement in algorithms and computer programming. Early research findings have shown that secondary education students have various difficulties to manipulate variables when they try to solve simple programming problems (Du Boulay, 1989; Soloway & Spohrer, 1989; Jimoyiannis, 2011). Therefore, students' efficient models about variables constitute an essential prerequisite towards building other abstract programming constructs, like counters, loops and arrays.

SOLO taxonomy in analyzing students' algorithmic thinking

Researchers and computer science educators have applied various educational taxonomies, in order a) to better analyse students' algorithmic thinking and the knowledge required to successfully solve programming problems and b) to design assessment tools of students' performance in programming tasks or problems. Bloom taxonomy, in its original (Bloom, 1956) and revised form (Anderson et al., 2001), and the Structure of the Observed Learning Outcome (SOLO) taxonomy (Biggs & Collis, 1982) are the most popular and widely adopted taxonomies.

However, the application and the interpretation of both Bloom taxonomies in computer programming appeared to be problematic (Shuhidan et al. 2009; Thompson et al. 2008). Existing research has shown that the interpretation of the revised Bloom taxonomy in computer science tasks is not straightforward (Fuller et al., 2007; Thomson et al., 2008) while experts consider it difficult to agree on a common interpretation (Johnson & Fuller, 2006). Another critical problem is related to the inherent difficulties to achieve an accurate classification of a code writing task, because of the different views regarding the task complexity, the magnitude of the code, and the sophistication of the cognitive processes required to solve a particular problem (Whalley & Kasto, 2014).

By adopting a constructivist pedagogical philosophy, Biggs & Collis (1982) proposed SOLO taxonomy to describe a hierarchy of learning, in terms that each partial construction (level) becomes a foundation upon which learning is further enhanced and extended. SOLO includes five levels of sophistication which reveal the structural complexity of students' knowledge: *prestructural, unistructural, multistructural, relational,* and *extended abstract*. These levels are ordered in terms of various characteristics that represent students' movement from the concrete to the abstract and from surface to conceptual understanding.

SOLO taxonomy can be used to define intended learning outcomes, instructional ways that support them, and forms of assessment that evaluate to what extent the outcomes were achieved. Biggs (2003) suggested that SOLO is applicable to measuring the learning outcomes achieved in different cognitive areas and subjects, among different levels of students and in different types of assignments. Therefore, SOLO taxonomy provides a qualitative way to classify cognitive processes. It has been applied to many different subjects, such as mathematics (Chick, 1998), biology (Campbell et al., 1998), language studies (Lake, 1999), poetry and history (Biggs & Collis, 1982).

Lister et al. (2006) were first that used SOLO taxonomy to classify students' responses to computer programming problems. In the last decade, SOLO was widely used in computer science education to classify students' responses to code reading tasks (Clear et al., 2008; Sheard et al., 2008) and their solutions to programming assignments (Whalley et al., 2011; Corney et al., 2014; Seiter, 2015).

A novel combination of the revised Bloom taxonomy and SOLO taxonomy was proposed by Meerbaum-Salant, Armoni & Ben-Ari (2013). It was also used to design programming assignments in Scratch. More recently, Ginat and Menashe (2015) presented a framework of SOLO utilization to assess algorithmic design features of code writing by focusing on selection, flexible manipulation and composition of basic design patterns. In addition, Seiter (2015) used SOLO to measure computational thinking skills of fourth grade students while Lavy & Yadin (2014) used an extended version of SOLO taxonomy in software-based projects.

Using SOLO taxonomy in programming tasks about arrays

The lower SOLO levels focus on the quantity of knowledge (i.e. the amount of programming details a learner is able to use) while the higher levels focus on the development of relationships among programming concepts and their integration into a coherent construct (Thompson, 2007). Therefore, they could determine the boundaries between surface and deep learning in algorithmic and computer programming. Independent researchers reported that SOLO taxonomy can be reliably applied to classify both, code comprehension and writing questions, and the student responses to those questions, as long as the classifiers have a shared understanding of applying this taxonomy to code comprehension tasks (Clear et al., 2008; Sheard et al., 2008).

An initial set of guidelines and SOLO descriptors to classify code writing solutions were proposed by Lister et al. (2009). However, classifying student answers to code writing tasks using this interpretation of the SOLO levels proved difficult (Lister et al., 2009). The higher the SOLO level of a question is, the most difficult is to measure student performance to this question (Whalley et al., 2011).

On the other hand, Petersen, Graig & Zingaro (2011) argued that any specific taxonomy level is related to the particular examples and the exercises used during course instruction. They advocated that students' answers should be categorized in the unistructural level of the SOLO taxonomy when a specific problem or programming situation has been encountered in the class. Similarly, Izu & Weerasinghe (2016) agreed that any SOLO classification is highly dependent on course activities and the examples used.

In this context, the main idea driving the use of SOLO taxonomy in this research project was that students' answers were classified not so much according to the code correctness but according to the level of programming knowledge integration demonstrated by the students (Lister et al., 2009). The classification of students' responses in SOLO levels was mainly based on the logic and the structure of each particular algorithm rather than the correctness of the code written. In other words, an integrated answer is a strong/convincing indicator of student's understanding of the code and, consequently, of his ability to apply new programming concepts to similar problems. As Lister et al. (2006) argued, the students who are not able to read and describe a programming code relationally do not possess the skills needed to autonomously create their own code to solve similar problems.

In order to provide a consistent framework of classifying students' responses, we have adopted a series of criteria that expected to be used in the analysis of students' responses to the programming tasks of this study. A complete analysis schema was achieved and used in the present study to describe the cognitive and structural complexity regarding code explaining and code writing tasks. Table 1 presents an example of the SOLO levels regarding students' responses to the bubble sort sorting algorithm.

```
for i←2 to N do
  for j←N downto i do
    if ( array[j] < array[j–1]) then
      temp ← array[j]
      array[j] ← array[j–1]
      array[j–1] ← temp
    end if
  end for
end for
```

Table 1. SOLO categories and mapping descriptions

SOLO category	Description	Response examples
Prestructural	• The response has little or no relevance to the given task or problem • Substantial lack of knowledge in terms of the programming constructs involved • Significant programming misconception	• *"The given algorithm searches for a value in the array"* • *"The given algorithm finds the maximum value of the array"* • Response indicating confusion of the index of an array element with the content of the array element
Unistructural	• The response focuses on a single statement or concept • The response just describes one part of the code • Student manifests a correct grasp of some but not all aspects of the problem • Code reiteration/plagiarism from other programming tasks or educational material	• *"The algorithm performs comparisons between elements of the array"* • *if array[j] is less than array[j–1] then* *temp is assigned to array[j]* *array[j] is assigned to array[j–1]*
Multistructural	• The response describes the largest part of the code (line by line) with minor flaws in the description • The response describes the largest part of the code (line by line) without focusing on the relations among parts/statements	• *"This code part compares every element in the array with the next one. If an element is greater than the next one, the two elements will be swapped."* • *"This algorithm compares adjacent elements. If they are not in ascending order they will be interchanged"*
Relational	• The response shows understanding of the purpose and the functionality of the code • The response shows integration of programming concepts and constructs • Understanding how to apply the key programming concept/idea to a familiar/similar problem	• *"This code segment sorts an array in descending order"* • *"This code segment sorts an array in ascending order"* • *"This code segment sorts the elements of the array"*

Extended abstract	• Questioning and going beyond key programming principles or the problem assigned
	• The response relates a programming concept or principle in a way that indicates students' ability to handle new or unseen problems
	• The response indicates proper use of constructs and concepts beyond the task requirements, in order to provide an improved solution
		• The algorithm is optimized so that execution stops when the array has been sorted, by using a logical variable as event flag, for example:
		i ← 2
		do
		AscendingOrder ← True
		for j=N downto i do
		    if (number[j] < number[j–1]) then
		    swap(number[j], number[j–1])
		    AscendingOrder ← False
		    end if
		end for
		i ← i + 1
		while not AscendingOrder

Our SOLO analysis schema has adopted the approach of Lister et al. (2006), which suggested that a relational response may be either correct or incorrect. The classification of each response was examining whether a student is able to identify the purpose of a code segment rather than focusing on the correctness of that code. Therefore, the response "the algorithm sorts the array in descending order" is classified into the relational level, since it indicates a stable representation of the sorting algorithm, despite the wrong order.

Students' performance at the highest SOLO level of algorithmic thinking is a very interesting case with regards to both, educational and research, perspectives. The extended abstract example, given in Table 1, could be a possible answer to a code writing task that requests a sorting algorithm. Ideally, the classification of responses into the level of extended abstract indicates students' ability a) to use programming principles and constructs that were not explicitly taught and/or b) to move beyond the problem assigned. Extended abstract responses are expected to appear in writing code tasks that require students to use programming constructs from scratch, in order to provide a new solution or an improved algorithm regarding an existing/given solution (Ginat & Menashe, 2015; Whalley et al., 2011; Whalley & Kasto, 2014).

Similarly, Azu, Weerasinghe & Pope (2016) adopted the same approach using tasks focused on loop design and array code writing rather than code reading. However, while they revealed common errors and misconceptions of students about arrays and loops, their study was focused on the design of an evaluation framework based on SOLO taxonomy to assess students' performance in programming tasks. Murphy et al. (2012) followed the same approach using tasks asking the students to get in comparing and explaining short code writing questions. The present study was designed with the objective to analyse and interpret, using the SOLO taxonomy, the various mental models and misconceptions of K-12 students' about arrays. Therefore, the tasks included in the present study were not expected to record students' performance in the extended abstract level.

Research Methodology

Research context

According to the Greek National Curriculum for upper secondary education (CF, 1998), third grade students (K-12) in the technological direction are attending an obligatory course, entitled Development of Applications in Programming Environments (DAPE). This course is taught for 2 hours per week and aims at students' development of algorithmic thinking and acquiring basic knowledge and skills to solve programming problems. The students have the opportunity to use pseudocode and procedural programming environments (e.g. Pascal) for exploration and writing algorithms by using the basic programming structures (variables, control and loop structures, arrays, procedures and functions).

This introductory programming course was continually offering, since 2000, while the subject content is included in the national university entrance examinations. The study presented here is of particular interest considering that computer programming has already a long history in Greek secondary education and the existing research data regarding secondary education students are limited.

The sample

The present survey was administered in three upper secondary schools in the urban area of Athens. A total of 100 third grade (K-12) students, attending this introductory programming course, participated in the investigation on a voluntary basis. However, 10 students did not respond to all the programming tasks in the instrument. Thus, the data used in our analysis concern 90 students. The participants were aged between 17-18 years while they were coming from medium socioeconomic levels. The majority of them (93%) reported that they have computer access at home. In addition, 24% of the students reported that, before entering this course, they had no opportunity to develop a complete program using a programming environment.

No intervention took place before the survey. All students received regular instruction in the classroom with regards to the related topic (arrays) in the context of this course. They were introduced to variables, control and loop commands, arrays and user-defined functions using a Pascal-like algorithmic language. Students were also informed about the purpose of the study and they were assured that data analysis would have no impact to their grades.

The procedure

The survey was administered two months after the array unit was taught in these schools. Normal exam conditions were applied during the survey procedure. The students answered to the tasks anonymously using paper and pencil. In addition, they were requested to add a short written explanation justifying their responses. Researcher's role was restricted to answering students' questions and clarifying the programming tasks under study. In order to reduce students' possible anxiety and familiarize them with the whole process, three simple questions-tasks were given as a starting point. Data from these questions were not included in the analysis.

No time limit was set to the students to respond to the main research tasks; however the whole process was completed within one hour. The students were also informed of their rights to withdraw from completing the questionnaire at any time during the data collection.

The instrument

The research instrument was created by the researchers and included three open-ended tasks. The tasks and the related questions were worded in a way that the students had to explain the purpose of the programming tasks and, in addition, to give a concise explanation of its functionality rather than trying to describe a code segment line by line. Lister et al. (2009) suggested a hierarchy of essential programming skills, which depicts students' ability to trace, explain and write code. The bottom level is related to students' knowledge and understanding of elementary programming constructs, like basic terms, data types and control structures. The intermediate level concerns students' ability to accurately read and trace a code segment and, as a result of the above, their ability to explain the outcome of a piece of code in an abstractive level. On the top of the proposed hierarchy is students' ability to write, correct and improve non-trivial code segments with the aim to solve a specific problem.

Towards determining a hierarchy of students' programming skills, which could be applied to student tasks about arrays, we followed an approach similar to Tan et al. (2009) and Whalley et al. (2006). Therefore, the tasks given to the participants were classified to the following types:

a) Explain code: A segment of code was given; the students were asked to explain, in plain text, what is the outcome of this piece of code (Task 1).

b) Skeleton-code: Students were given a piece of code; they were asked to extend the functionality of this code by redesigning the given program in order to achieve a new outcome (Task 2).

c) Code tracing: Students were given a piece of code; they were asked to determine the expected output after code execution (Task 3).

Students' answers were independently analysed and classified, according to the SOLO categories and mapping descriptions presented in Table 1, by two computer science educators. The inter-rater reliability achieved an agreement level of 95%. The discrepancies among the two educators were solved by a third ratter.

Results and analysis

Task 1

You are asked to write a code segment which can execute the following: a) reading the grades of N students, b) checking their validity (value between 1 and 20), and c) storing the grades in an array. Which one of the code segments below would be your choice? Explain your answer.

```
Algorithm A1
for i=1 to N
   repeat
      read grade
   until grade >= 1 and grade <=20
   G[i] ← grade
end for
```

```
Algorithm A2
for i=1 to N
   repeat
      read G[i]
   until G[i] >= 1 and G[i] <=20
end for
```

Table 2 presents students' responses according to the SOLO taxonomy levels. We classified into the prestructural level the responses which were unrelated to the task problem, they indicate critical deficiencies in programming thinking and/or difficulties in using basic programming structures. Indicative examples of responses in this category were:

"I do not understand this problem"

"The validity condition is false".

In the unistructural level were identified responses based on justifications indicating that the students recall a familiar data validation technique and reiterate it uncritically, i.e. with no logical argumentation related to the context of the specific problem. In addition, some students used this code segment without deeper understanding of its functionality. Typical responses assigned to this SOLO level were the following:

"Algorithm A1 looks more familiar"

"Algorithm A1 is the proper validation technique".

Table 2. Students' responses to Task 1

SOLO level	Percentage %	Students (N=90)
Prestructural	16.7	15
Unistructural	20	18
Multistructural	28.9	26
Relational	34.4	31

Into the multistructural level we have classified the responses of the students which describe the largest part of the code (line by line) without focusing on the logic and the relations among its parts (i.e., validity checking and value assignment to the array elements). Typical responses in this SOLO level were like

"My choice is algorithm A1, because the validity checking should be before the command of value assignment to an array element."

The majority of the students in the sample did not choose algorithm A2 in order to check the validity of data and store them in an array. Rather they prefer to use an auxiliary variable (named grade) to read input data and, following, to proceed to the successive assignments of the auxiliary variable's content to the corresponding array elements.

Finally, 34% of the responses were classified in the relational level. Typical responses of this type were:

"Algorithm A2 requires less memory"

"Algorithm A2 is a more efficient algorithm".

Firstly, their justifications indicate that these students have built effective mental models of the concepts of variable and array. In addition, they have organized their representations regarding this particular code segment into a larger programming structure, thus achieving an overall understanding beyond its functionality, i.e. in terms of *algorithm efficiency* and *working memory* needs.

In conclusion, the analysis of students' justifications in this task revealed a) difficulties to use arrays effectively and b) a potential misconception with regards to the notion of array, i.e. the students were not able to use an array element (G[i] in this case) as a single variable.

Task 2

Consider the following segment of code that reads the grades of 50 students and computes the maximum value. You are asked to modify this algorithm, in order to compute the number of students who received the highest grade.

```
read grade
max ← grade
  for student=2 to 50
     read grade
     if grade > max then
        max ← grade
     end if
end for
```

The objective of this task was to investigate students' ability to recognize the need of using arrays in programming problems. The algorithm above computes the best grade and it is a common example used by the teachers in their instruction; it is familiar to the majority of the students. However, in order to compute the number of students with the highest grade, a second pass on the data is necessary. Therefore, it was expected from the students to identify the need of using an array and make appropriate modifications of the given algorithm in order to solve this problem.

It should be noted that the algorithm presented below offers a solution which is not based on the array data structure. Since no student in this study gave the following one-pass algorithm, it seems that the array solution is potentially close to students' thinking.

```
read grade
max ← grade
count ← 1
for student=2 to 50
    read grade
    if grade > max then
        max ← grade
        count ← 1
    else if grade = max then
        count ← count + 1
    end if
end for
```

Table 3 summarizes students' responses to Task 2. It is worthy to note that 10 students gave no complete response to this task. Into the prestructural level we have classified responses that, more or less, reiterate the given algorithm (12%).

```
read grade
max ← grade
count ← 0
for student=2 to 50
    read grade
    if grade > max then
        max ← grade
        print count + 1
    end if
end for
```

These students have just rewritten the initial code segment and they added a variable that is supposed to be a counter. Therefore, they recognized the need of introducing a counter type variable and they were able to correctly apply the counter initialization command. Despite that, the *count* variable was not used appropriately within the loop command in order to compute the number of grades with the maximum value.

Into the unistructural level we have classified similar responses that indicate students' ability to use the counter variable appropriately. However, responses like the following piece of code do not constitute valid solutions of the Task 2.

Table 3. Students' responses to Task 2

SOLO level	Percentage %	Students (N=90)
No answer	11.1	10
Prestructural	12.2	11
Unistructural	22.2	20
Multistructural	30	27
Relational	24.4	22

```
read grade
max ← grade
count ← 0
for student=2 to 50
    read grade
    if grade > max then
        max ← grade
        count ← count + 1
    end if
end for
```

Into the multistructural level we have classified the responses that indicate students' understanding of the logic of the code, i.e. appropriate use of the counter variable and correct calculation of the maximum value. For example,

```
read grade
max ← grade
for student=2 to 50
    read grade
    if grade > max then
        max ← grade
    end if
end for
count ← 0
for student=1 to 50
    if grade = max then
        count ← count + 1
    end if
end for
```

However, these students were not able to realise that the data, upon which their computation is operated, are not available. In other words, they believe that the variable *grade* keeps all the 50 values. It seems that they have a faulty representation of the programming variable, based on the box/stack model (Jimoyiannis, 2011), which directed them to believe that they can recover all the grade values. This faulty representation prevented the students performed in this SOLO category to identify the need of using an array in their modification of the algorithm. This is a clear example that students' misconceptions regarding the concept of programming variable affect their representations of the array concept and induce serious cognitive difficulties.

It should be noted that that 8 students gave different solutions which were also classified into the multistructural level. Their algorithm uses correctly the counter variable and the related computation commands; however, the code segment of the successive grades' reading is included again. Students' choice to input again the 50 grades, in order to compute the maximum value, provided strong evidence that they have understood that a variable can currently hold only one value. However, according to this particular problem, the grades were given once from the standard input. It seems that these students have adjusted the given problem to a familiar solution. They exhibited a clear understanding of the main parts of the code but they failed to identify the relationship between these parts and use an array to store the given data.

Finally, one out of four students gave responses classified into the relational level. These students presented completely correct and justified solutions based on the appropriate use of both, the array and counter concepts, like the following:

```
read grade[1]
max ← grade[1]
for student=2 to 50
    read grade[student]
    if grade[student] > max then
        max ← grade[student]
    end if
end for
count ← 0
for student=1 to 50
    if grade[student] = max then
        count ← count + 1
    end if
end for
```

The responses above could be classified to the extended abstract level of the SOLO taxonomy as well, since the students had to use a new programming construct (the array) in order to design a new algorithm. However, this solution was assigned to the relational level because the students were not asked to solve the problem from scratch, in an ad hoc manner, but to modify/redesign a given solution by using the array structure.

Task 3

Consider the following segment and the array X containing six integer values as following

6	5	4	3	2	1

```
i ← 1
while i < X[i] do
    X[X[i]] ← X[i]
    i ← i + 1
    print i, X[i], X[X[i]]
end while
```

What do you expect to be displayed on the screen when the program finishes? Explain your answer.

The structure of the algorithm above is quite simple, since it includes only a while…do loop and three statements therein. Task 3 appeared to be the most difficult problem, since one out of three students in the sample did not try to give any answer (Table 4). During the responding process, many students reported to the researcher that they were not able to understand this particular task and the meaning of X[X[i]] when figuring an answer. So, they decided to leave this task unanswered. Jakwerth & Stancavage (2003) identified three main reasons why students participating in research studies leave questions unanswered: a) difficulties in understanding, b) lack of knowledge, c) lack of adequate time to respond. In this experiment, the students had enough of time to complete the programming tasks. Our interpretation of the missing responses is that this task appeared to be of major difficulty for the students and, therefore, 30% of them decided to leave it unanswered.

Table 4. Students' responses to Task 3

SOLO level	Percentage %	Students (N=90)
No answer	33.3	30
Prestructural	16.7	15
Unistructural	38.9	35
Multistructural	11.1	10

We have classified as prestructural, the responses that indicate substantial lack of complete mental models regarding the concepts of array and the array index (16.7%). Following we present some typical examples of students' responses in this category:

"*You cannot use X[i] inside brackets. An index should be there*"

"*X[i] cannot be an array index*"

"*X[i] is an array and not a variable*".

The students above have the idea that the element X[i] in an array of integers cannot be used as a variable. This is an indicator of incomplete or faulty representations regarding the array concept. We conclude that students' difficulty to use an integer array element as an array index is rooted in their inability to conceive an array element X[i] as a variable.

In addition, we have also classified into the prestructural level the responses given by seven students, which revealed the perception that X[i] represents the array as a whole structure. When they were referring to the array as a whole, the students above used the term "array X[i]" instead of "array X".

It is a common practice, among educators in the introductory programming lessons, that the first problems about arrays students deal with have a typical mathematical nature. In addition, the programming names used in the first programs to describe arrays are capital letters (A, B, X etc.), and i, j for the array indices. These symbols are quite similar to those used in mathematical problems and, probably, constitute a source of difficulties that the students need to overcome. Despite that the mathematical analogy is actually helpful for the instructors, it appeared that the static-functional view of the array concept is not adequate for the students to build a coherent representation of the dynamic nature of the array concept as a data structure.

Into the unistructural level we have classified students' responses (38.9%) that showed an incomplete representation of the array concept and the connection between the index of an array element and its value. The majority of the students in this category used X[i] as a static variable and not as the ith element of the array. They figured their responses using successive calculations of the array elements and putting the related values in tables like Table 5. Therefore, these students had the opportunity to create a correct view of the overall computation process using a, more or less, mechanical way, i.e. when i is increased, X[i] refers to the next array element. It is quite clear in Table 5 that these students were able to update the value of the index i, i.e. i=3, but the variable X[i] still refers to the previous index X[2] and the print statement outputs 3, X[2] and X[X[2]] respectively. This response revealed a faulty-static mental model behind students' approach, i.e. they lack a representation that reflects the dynamic nature of the relationship among the index of an array element and the content (value) of that array element.

Table 5. Calculation table of successive array elements (Task 3)

i	X[i]	X[X[i]]	Output
1	6	6	2 6 6
2	5	5	3 5 5
3	4	3	4 4 3
4	3	5	

There is also an analogy with the concept of mathematical function; i.e. the relation between an array element X[i] and the corresponding index i and, on the other hand, the relation of the value of a real function f(x) and the variable x. According to Carlson (1998) and Thomson (1994), students lack the conceptual structures to model functional relationships, for example when the function value (output variable) is changing in relation to the continuous changes of an input variable. Moreover, students exhibit many difficulties in complex concepts like the function composition, i.e. f(f(x)). In our case, there is a functional representation or analogy in students' mind regarding the expression X[X[i]], i.e. the array element X[i] is dependent on the value of i. Therefore, the students were not able to update X[i] and X[X[i]] and they do not appear to attain this dynamic relation, i.e. to understand that X[i] is referring to a variety of array elements during algorithm execution; i.e., first it refers to X[1], then to X[2], and so on.

Finally, 11% of the students gave a correct response that was categorized to the multistructural level. These students showed their understanding of the code line by line. An indicative example of students' multistructural responses is presented in Table 6. The students have updated every element X[i] by using the previously updated value of the index i. In this case, they were able to identify that the variable X[i] in the print statement is not the same variable X[i] used in the previous assignment. This response shows complete understanding of the dynamic relations between array index i and the corresponding array element X[i].

Table 6. Example of students' responses classified in multistructural level (Task 3)

i	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[X[i]] ← X[i]	print i, X[i], X[X[i]]
1	6	5	4	3	2	1		
	6	5	4	3	2	**6**	X[6] ← X[1]	
2	6	5	4	3	2	6		2 5 2
	6	5	4	3	**5**	6	X[5] ← X[2]	
3	6	5	4	3	5	6		3 4 3
	6	5	4	**4**	5	6	X[4] ← X[3]	
4	6	5	4	4	5	6		4 4 4

Discussion and conclusions

The analysis presented in this paper used SOLO taxonomy to map students' responses to three programming tasks about arrays. The tasks were designed to examine the extent that students have developed the ability to solve algorithmic problems with arrays. The analysis has shown that the majority of the participants had serious difficulties to effectively respond to short code problems; they tended to give responses assigned to the lower SOLO levels, i.e. prestructural, unistructural and multistructural. The findings also revealed that many students in the sample had incomplete or faulty representations about the array concept which do not support them to connect the notions of array, array index and array element, in a meaningful and an effective way to solve programming problems. It seems that the cognitive barriers exhibited by many students are originating from persistent misconceptions and faulty mental representations of the programming variable concept.

The main findings drawn from this investigation are further discussed below, in relation to the research questions. The first difficulty we revealed indicates that the students are not familiar with using an array element as a single variable. They prefer to use simple variables in programming expressions rather than using arrays in order to handle data of the same type. Our findings confirm previous results of Hazzan, Lapidot & Ragonis (2011) about students' difficulty in understanding that each array-cell functions as any other variable. A recent study regarding pointers in C programming language provided empirical evidence that students prefer also to work with basic elements (single variables) rather that array objects (Craig & Petersen, 2016).

Many students were not able to identify which type of problems require the use of arrays structure. In the second task, instead of using an array the students used a variable as a stack that could 'remember' the history of all its previous values. The so-called *stack model* representation of the programming variable concept (Jimoyiannis, 2011) seem to play an important role in students' decision about the necessity of using an array. This is a strong indicator that some difficulties and misconceptions of students regarding arrays are connected or even caused by misconceptions about the variable concept. This note looks reasonable, since an array is a sequence of variables. However, the main conclusion is that, eventually, the array concept is constructed upon the concept of variable.

The above students' difficulties have a common origin which is related to their incomplete representations of the array concept as well as the perceived relations among the constitutional array elements. It is quite apparent that students are not able to build a common mental model for the concepts of variable and array element. The dominant representation of the array concept that students construct is a compact (single) programming entity rather that a coherent, linear structure of variables of the same type. Similar findings are reported by Hazzan, Lapidot & Ragonis (2011) who pointed that "many students think that when an array is used all its cells should be scanned".

It seems that many students cannot understand that an array element, its index and a variable share the same operational properties. The majority of the students failed to perceive the *dynamic nature* of an array and the relations among its constitutional entities, as well as to apply proper connections between indexes i, array elements X[i], and cell values X[i] in their algorithmic solutions. Our results provided evidence that the *mathematical representation* of the array construct, as a whole, and the relation between index i-array and element X[i] was prevalent among the students in the sample. Many students failed to grasp the dynamic features and they had a static perception of the array features. As a consequence, they handle an array as a function or reals or integers; i.e., when i is increased to i+1 in a loop, X[i] is still related to the previous value of the index i.

Table 7. Main findings regarding students' perceptions of the array concept

Students' difficulties	Misconception	Representation
To perceive and use an array element as a single variable	X[i] represents the whole array structure (not a single element)	An array element is a part of a compact-single entity (a set of data of the same type)
To judge the type of problems that require to use an array in their solution algorithm (when and why)	A variable can store all the previous values that were given in input; therefore there is no need to use an array	The stack/box model for the concept of programming variable
To properly use an array in a particular algorithm solving a computational problem	Confusion between the value of an array element X[i] and its index i	Incomplete mental representations of the array entities (X[i] and i) and the relations among them
To understand the dynamic relations between array indexes, array elements X[i], and the values X[i]	Mathematical view of the relation between the index i and the array element X[i]	Static representation of the relations among the array constitutional entities
To perceive the dynamic nature of an array		Static representation of the array structure

The critical misconceptions detected in this study concern a) students' confusion between the value of an array element X[i] and the index I, b) X[i] represents the whole array structure (not a single element corresponding to the index i) and c) the mathematical view of the relations between index i and array element X[i]. Table 7 summarizes the key findings revealed in this study in terms of students' difficulties, misconceptions and representations of the array concept in algorithmic problems.

With regards to the third research question, the results contributed to the existing literature and offered significant empirical evidence regarding the efficacy of SOLO taxonomy to achieve an insight of and explain students' mental models and their ability to use arrays in order to solve programming problems. It was reflected on the SOLO levels of cognitive complexity that students' understanding of the programming concepts involved in arrays is progressing from surface to deeper constructs. Confirming previous research studies, SOLO taxonomy allowed us to analyse the students' responses in terms of abstraction levels they achieved of the array concept (Lister et al., 2006; 2009; Sheard et al., 2008; Whalley et al., 2011; Corney et al., 2014; Seiter, 2015). Therefore, SOLO taxonomy offers a robust and coherent framework for analysing and understanding students' mental models and their misconceptions in arrays.

Implications for educational practice and future research

This study provided information about students' mental models and representations of the array construct as well as the difficulties they are facing when solving programming problems with arrays. The findings may be related to the particular sample and the educational context in Greek upper-secondary schools. Therefore, generalizations to other educational environments internationally are not obvious and should be done with cautiousness. However, the results presented in this paper could be of value for the design and the successful implementation of new learning environments that support students' algorithmic thinking, as well as the development of appropriate mental models about basic programming entities.

As an overall conclusion, the results indicated that secondary education students require time to progressively develop the appropriate mental models of the array concept. The typical programming languages and the traditional instructional approaches appear to be inefficient to support students to develop the necessary programming skills in arrays. Effective teaching interventions should be focused on revealing the conceptual structure of arrays and the dynamic features of many related programming concepts, i.e. array, variable, array index, array element, loop command etc. Teachers should better organize their teaching, in both classroom and laboratory sessions, and systematically engage their students in *problem-solving* activities requiring specific *code tracing* and *code explaining* tasks.

The traditional representation tools and approaches seem to be inefficient to support students' cognitive efforts towards transforming their faulty or incomplete models into sound and viable representations of the array concepts. For example, the graphical representation of the array concept used in traditional instruction and textbooks is a series of contiguous boxes that are supposed to form a larger construct (the array). For the majority of the students, this is probably a misapplied analogy (Soloway & Spohrer, 1989), which directs them to consider an array element not as a variable but as a part of set of data (the array).

The analysis concludes by adopting computer-based algorithm simulations as efficient learning environments for algorithmics and introductory programming courses (Sorva, Karavirta & Malmi, 2013; Urquiza-Fuentes & Velazquez-Iturbide, 2009). Properly designed algorithm simulations are expected to a) offer dynamic visual representations of the key data structures, b) provide students with opportunities for experimentation in order to unfold the dynamic features of the array construct, and c) support computer science teachers to design and implement in their classrooms inquiry and constructivist practices through problem-solving activities.

Our current research is directed to the design of a Web-based environment with the affordances to facilitate a) students' experimentation with dynamic algorithm visualizations and b) students' development of efficient mental models about the logic of basic array algorithms. The array cells need to be visualized as separate variables and not as contiguous boxes in order to help students perceive an array not as a single entity but as a sequence of objects-variables of the same type. The system should also allow students' to modify array algorithms in terms of both, the code and the input data, and reflect on the screen outcomes. In addition, we are currently working to develop a coherent framework about arrays and other data structures with the aim to harness the affordances of algorithm visualization towards improving students' mental models about complex programming concepts.

References

ACARA (2013). *Draft Australian Curriculum: Technologies Foundation to Year 10 Consultation Report*. Retrieved 27 May 2016, from http://www.acara.edu.au/verve/_resources/Draft_Australian_Curriculum_Technologies_-_Consultation_Report_-_August_2013.pdf

ACM (2013). *Computer Science Curricula 2013. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York: ACM and IEEE Computer Society. DOI: 10.1145/2534860.

Adcock, B., Bucci, P., Heym, D. W., Hollingsworth, E. J., Long, T., & Weide, B. (2007). Which pointers errors do students make? *ACM SIGCSE Bulletin*, *39*(1), 9-13.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich P. R., Raths, J., & Wittrock M. C. (2001*). A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. New York: Longman.

Bell, T., Newton, H., Andreae, P., & Robins, A. (2012). The introduction of computer science to NZ high schools: An analysis of student work. *Proceedings of the 7th Workshop in Primary and Secondary Computing Education*, (pp. 5-15). Hamburg: ACM.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, *20*(1), 45-73.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning. The SOLO taxonomy*. New York: Academic Press.

Biggs, J. (2003). *Teaching for quality learning at university*. Buckingham: The Society for research into Higher Education and Open University Press.

Bloom, B. S. (1956). *Taxonomy of educational objectives, Handbook 1: cognitive domain*. New York: Addison Wesley.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, *1*(2), 133-161.

Boustedt, J. (2012). Students' different understandings of class diagrams. *Computer Science Education*, 22(1), 29–62.

Campbell, J., Smith, D., & Brooker, R. (1998). From conception to performance: how undergraduate students conceptualise and construct essays. *Higher Education*, *36*(4), 449-469.

Carlson, M. P. (1998). A cross-sectional investigation of the development of the function concept. In A.H. Schoenfeld, J. Kaput, & E. Dubinsky (Eds.), *Research in Collegiate Mathematics Education. III. CBMS Issues in Mathematics Education* (pp. 114-162). Providence, RI: American Mathematical Society.

Chick, H. (1998). Cognition in the formal modes: research mathematics and the SOLO taxonomy. *Mathematics Education Research Journal*, *10*(2), 4-26.

Clear, T., Whalley, J., Lister, R., Carbone, A., Hü, M., Sheard, J., Simon, B., & Thomson, E. (2008). Reliably classifying novice programmer exam responses using the SOLO taxonomy. In S. Mann & M. Lopez (Eds.), *21st Annual Conference of the National Advisory Committee on Computing Qualifications* (pp. 23-30). Auckland: NACCQ.

Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., & Murphy, L. (2014). 'explain in plain english' questions revisited: data structures problems. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (pp. 591-596). New York: ACM Press.

Craig, M., & Petersen, M. (2016). Student difficulties with pointer concepts in C. *Proceedings of the Australasian Computer Science Week Multiconference* (Art. 8). New York: ACM.

CSAT (2011). *K–12 Computer Science Standards.* New York: Computer Science Teachers Association and ACM.

Dale, N. B. (2006). Most difficult topics in CS1: results of an online survey of educators. *SIGCSE Bulletin*. *38*(2), 49-53.

Danielsiek, H. (2012). Detecting and understanding student's misconceptions related to algorithms and data structures. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE' 12)* (pp. 21-26). New York: ACM.

Department for Education (2013). *National Curriculum in England: Computing Programmes of Study*. Retrieved 27 May 2016, from https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study.

De Raadt, M. (2007). A review of Australian investigations into problem solving and the novice programmer. *Computer Science Education*, *17*(3), 201–213.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57-73.

Eckerdal, A., & Thune, M. (2005). Novice java programmers' conceptions of "object" and "class", and variation theory. *ACM SIGCSE Bulletin*, *37*(3), 89-93.

Fleury, A. E. (2000). Programming in java: student-constructed rules. *ACM SIGCSE Bulletin*, *32*(1), 197-201.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukiermanm, D., Hernán-Losada, I., et al. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, *39*(4), 152-170.

Garner, S., Haden, P., & Robins A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In A. Young, & D. Tolhurst (Eds.), *Proceedings of the Australasian Conference on Computing Education (ACE '05)* (pp. 173-180). Darlinghurst: Australian Computer Society, Inc.

Ginat, D., & Menashe, E. (2015). SOLO taxonomy for assessing novices' algorithmic design. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE' 15)* (pp. 452-457). New York: ACM.

Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to teaching computer science: an activity-based approach*. London: Springer.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin*, *29*(1), 131-134.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, *35*(1), 153-156.

Informatics Europe & ACM (2013). *Informatics education: Europe cannot afford to miss the boat*. Report of the joint Informatics Europe & ACM Europe Working Group on Informatics Education, Retrieved 27 May 2016, from http://europe.acm.org/iereport/ACMandIEreport.pdf

Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 251-259). New York: ACM.

Jakwerth, P., & Stancavage, F. (2003). *An investigation of why students do not respond to questions*. Working Paper No. 2003-12. Washington, DC: US Department of Education, National Center for Education Statistics.

Jimoyiannis, A. (2011). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science & Technology Education*, *4*(2), 53-74.

Johnson, C., J., & Fuller, U. (2006). Is Bloom's taxonomy appropriate for computer science? *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006* (pp. 120-123). New York: ACM.

Kaczmarczyk, L., Petrick, E., East, P., & Herman, G. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)* (pp. 107-111). New York: ACM.

Karpierz, K., & Wolfman, S. (2014). Misconceptions and concept inventory questions for binary search trees and hash tables. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)* (pp. 109-114). New York: ACM.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin, (37)*3, 14-18.

Lake, D. (1999). Helping students to go SOLO: teaching critical numeracy in the biological sciences. *Journal of Biological Education*, *33*(4), 191-198.

Lavy, I., & Yadin, A. (2014). Extending the SOLO model for software–based projects. *International Journal of Modern Education and Computer Science*, *6*(3), 1-10.

Lister, R., Simon, B., Thomson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin, 38*(3), 118–122.

Lister, R., Clear, T., Simon, Bouvier, D., Carter, P., Eckerdal, A., Jackova, J., Lopez, M., McCartney, R., Robbins, P., Seppällä, O., & Thomson, E. (2009). Naturally occurring data as research instrument: analysing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, *41*(4), 156–173.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating the viability of mental models held be novice programmers. *Computer Science Education*, *21*(1), 57-80.

Madison, A., & Gifford, J. (2003). Modular programming: novice misconceptions. *Journal of Research on Technology in Education*, *34*(3), 217-219.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B-D., Cary Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-180.

National Curriculum (2011). *Curriculum of Informatics and ICT.* Athens: Institute of Educational Policy, Ministry of Education. Retrieved 27 May 2016, from http://ebooks.edu.gr/new/ps.php.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. M. (2013). Learning computer science concepts with Scratch. *Computer Science Education*, *23*(3), 239-264.

Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to 'explain in plain english' linked to proficiency in computer-based programming. *Proceedings of the 9th Annual International Conference on International Computing Education Research (ICER '12)* (pp. 111-118). New York: ACM.

Petersen, A. Graig, M., & Zingaro, D. (2011). Reviewing CS1 exam question content. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11*) (pp. 631-636). New York: ACM.

Putnam, R. T., Sleeman, D., Baxter, J., & Kuspa, L. (1989). A summary of misconceptions of high school Basic programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 301-314). Hillsdale, NJ: Lawrence Erlbaum Associates.

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*(3), 213-221.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137-172.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, *15*(1), 59-82.

Seiter, L. (2015). Using SOLO to classify the programming responses of primary grade students. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)* (pp. 540-545). New York: ACM.

Seppälä, O., Malmi, L., & Korhonen A. (2006). Observations on student misconceptions – A case study of the Build-Heap algorithm. *Computer Science Education*, *16*(3), 241–255.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thomson, E., & Whalley, J. L. (2008). Going SOLO to assess novice programmers. *ACM SIGCSE Bulletin*, *40*(3), 209-213.

Shuhidan, S., Hamilton, M., & D' Souza, D. (2009). A taxonomic study of novice programming summative assessment. In M. Hamilton, & T. Clear (Eds.), *Proceedings of the 11th Australasian Conference on Computing Education (ACE '09)* (pp. 147-156). Darlinghurst: Australian Computer Society, Inc.

Sirkiä, T., & Sorva, J. (2012). Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (pp. 19-28). New York: ACM.

Soloway, E., & Spohrer, J. C. (1989). *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education, 13*(2), Art.8.

Sorva, J., Karavirta, V., & Malmi L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions of Computing Education*, *13*(4), Art. 15.

Thompson, P. W. (1994). Images of rate and operational understanding of the fundamental theorem of calculus. *Educational Studies in Mathematics*, *26*, 229-274.

Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In S. Hamilton & M. Hamilton (Eds.), *Proceedings of the 10th Conference on Australasian Computing Education* (pp. 155-161). Darlinghurst: Australian Computer Society, Inc.

Tan, G., & Venables, A. (2009). Wearing the Assessment 'BRACElet'. *Journal of Information Technology Education*, *9,* 25-34.

Urquiza–Fuentes, J., & Velazquez–Iturbide, J.A. (2009). A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Transactions of Computing Education*, *9*(2), Art. 9.

Whalley, J., Lister, R., Thomson, E., Clear, T., Robbins, P., Kumar, P.K.A., & Prasard, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO Taxonomies. In D. Tolhurst, & S. Mann (Eds.), *Proceedings of the 8th Australasian Computing Education Conference* (pp. 243-252). Darlinghurst: Australian Computer Society, Inc.

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In J. Hamer, & M. de Raadt (Eds.) *Proceedings of the Thirteenth Australasian Computing Education Conference* (pp. 37-46). Darlinghurst: Australian Computer Society, Inc.

Whalley, J., & Kasto, N., (2014). How difficult are novice code writing tasks? A software metrics approach. In J. Whalley & D. D'Souza (Eds.), *Proceedings of the Sixteenth Australasian Computing Education Conference*, (pp. 105-112). Darlinghurst: Australian Computer Society, Inc.

Wolfgang, P., & Vahrenhold, J. (2013). Hunting high and low: instruments to detect misconceptions related to algorithms and data structures. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)* (pp. 29-34). New York: ACM.

To cite this article: Vrachnos, E., & Jimoyiannis, A. (2017). Secondary education students' difficulties in algorithmic problems with arrays: An analysis using the SOLO taxonomy. *Themes in Science and Technology Education*, 10(1), 31-52.

URL: http://earthlab.uoi.gr/theste