Revista Brasileira de Informática na Educação – RBIE
Brazilian Journal of Computers in Education
(ISSN online: 2317-6121; print: 1414-5685)
http://br-ie.org/pub/index.php/rbie

Submission: 10/Jun/2021; 1st round notif.: 17/Sep/2021; New version: 15/Oct/2021; 2nd round notif.: 28/Oct/2021;
Camera ready: 29/Oct/2021; Edition review: 03/Nov/2021; Available online: 10/Dec/2021; Published: 10/Dec/2021;

A Detailed Item Response Theory Analysis of Algorithms and Programming Concepts in App Inventor Projects

Nathalia da Cruz Alves
Graduate Program in Computer Science
Department of Informatics and Statistics
Federal University of Santa Catarina
nathalia.alves@posgrad.ufsc.br

Christiane Gresse von Wangenheim
Graduate Program in Computer Science
Department of Informatics and Statistics
Federal University of Santa Catarina
c.wangenheim@ufsc.br

Jean Carlo Rossa Hauck
Graduate Program in Computer Science
Department of Informatics and Statistics
Federal University of Santa Catarina
jean.hauck@ufsc.br

Adriano Ferreti Borgatto
Graduate Program in Assessment Methods and Management
Department of Informatics and Statistics
Federal University of Santa Catarina
adriano.borgatto@ufsc.br

Abstract

Teaching computing in K-12 is often introduced focusing on algorithms and programming concepts using block-based programming environments, such as App Inventor. Yet, learning programming is a complex process and novices struggle with several difficulties. Thus, to be effective, instructional units need to be designed regarding not only the content but also its sequencing taking into consideration difficulties related to the concepts and the idiosyncrasies of programming environments. Such systematic sequencing can be based on large-scale project analyses by regarding the volition, incentive, and opportunity of students to apply the relevant program constructs as latent psychometric constructs using Item Response Theory to obtain quantitative 'difficulty' estimates for each concept. Therefore, this article presents a more detailed analysis and the interpretation of the results obtained in earlier research of a large-scale data-driven analysis of the demonstrated use in practice of algorithms and programming concepts in App Inventor based on the CodeMaster rubric as well as its error measurement. Based on a dataset of more than 88,000 App Inventor projects assessed automatically with the CodeMaster rubric, we perform an analysis using Item Response Theory. The results demonstrate that the easiness of some concepts can be explained by their inherent characteristics, but also due to the characteristics of App Inventor as a programming environment. These findings are discussed with regard to curricular guidelines for computing education in K-12, especially concerning the adequacy of their proposed learning sequence. These results can help teachers, instructional and curriculum designers in the sequencing, scaffolding, and assessment design of computing education in K-12 using programming environments.

Keywords: *Algorithms and Programming; App Inventor; Item Response Theory; Sequencing; Rubric*

1 Introduction

The importance of computing nowadays for anyone regardless of the area of expertise is widely recognized. Consequently, computing education is making its way into K-12 worldwide, ranging from online MOOCs, extracurricular activities to courses fully integrated into the curriculum (Grover & Pea, 2013; Hubwieser et al., 2015; Lye & Koh, 2014). Several countries have developed guidelines and curricula for K-12 computing education (Webb, Davis, & Bell, 2014). Among those, one of the most prominent models is the K-12 Computer Science Framework (CSTA, 2016) defining a set of core computing concepts and practices to be covered in K-12. The core concepts represent major content areas in the field of computer science, including computing systems, networks, data and analysis, algorithms & programming as well as the impacts of computing. Core practices represent behaviors that computationally literate students should use to engage with the concepts of computing, such as recognizing and defining computational problems and creating computational artifacts. The standard also defines the sequencing of these concepts and practices describing how the students' conceptual understanding and practice of computing should become more sophisticated over time and across educational stages in K-12. Other guidelines and curricula, such as Computing at School (CAS, 2015) or the Brazilian Computer Society Guidelines for Computing Education in K-12 (SBC, 2018), cover similar basic concepts and practices.

There are several approaches to teach computing, yet, in practice, they typically focus on algorithms and programming concepts and related practices as being one of the main knowledge areas of computing (Grover & Pea, 2013; Grover, Basu, & Schank, 2018; Turbak et al., 2014). This comprises the competency to develop algorithms to solve problems in a language that computers can understand including basic programming concepts such as control (e.g., loops and conditionals), modularity, variables, etc. (Figure 1).

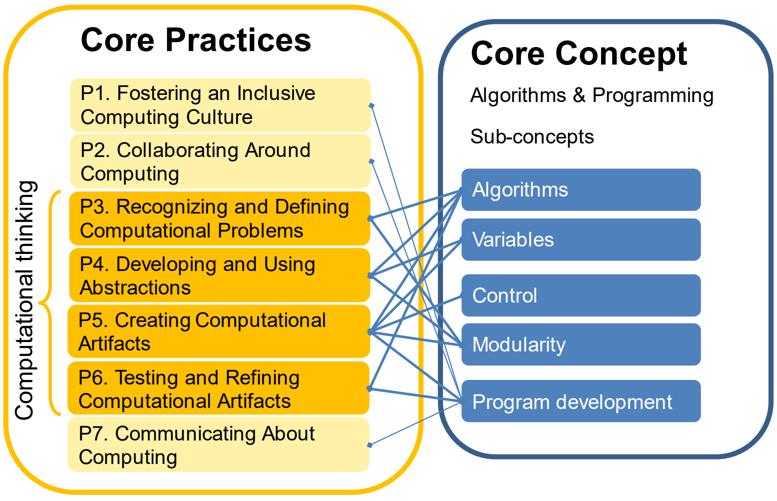

Core Practices
P1. Fostering an Inclusive Computing Culture
P2. Collaborating Around Computing
P3. Recognizing and Defining Computational Problems
P4. Developing and Using Abstractions
P5. Creating Computational Artifacts
P6. Testing and Refining Computational Artifacts
P7. Communicating About Computing
Computational thinking
Core Concept
Algorithms & Programming
Sub-concepts
Algorithms
Variables
Control
Modularity
Program development

Figure 1: Core practices and sub-concepts related to the core concept algorithms & programming concepts (CSTA, 2016).

Variables refer to storing and manipulating data from computer programs. Control concepts specify the order in which instructions are executed within an algorithm or program (e.g., using loops and/or conditionals). Modularity involves dividing complex tasks into simpler tasks and combining them to create something complex. Program development represents the software engineering process that is repeated until acceptance criteria are met. In addition, several core practices are related to algorithms & programming as presented in Figure 1.

In order to introduce programming in K-12, typically visual block-based environments are used. These environments allow to choose and drag-and-drop commands providing visual cues to the user as to how and where commands may be used reducing the cognitive load for novices (Papadakis et al., 2017; Weintrop, 2019). A prominent example is App Inventor (appinventor.mit.edu), an online platform for the development of mobile applications for Android devices. It is used by a wide range of people of all ages and backgrounds with more than 1 million unique monthly active users from 195 countries who created almost 35 million mobile apps as of January 2021. App Inventor projects can be shared via the App Inventor Gallery (MIT, 2020) under the creative commons license. App Inventor is also widely used to teach computing through the development of mobile applications (Wolber, Abelson, & Friedman, 2014) adopting diverse instructional strategies, ranging from well-defined interactive tutorials to open-ended ill-structured activities in a constructivist context following a problem-based learning approach (Patton, Tissenbaum, & Harunani, 2019). These typically aim at teaching students to create their mobile applications to solve real-world issues applying a computational action strategy to make computing education more inclusive, motivating, and empowering for young learners (Fee & Holland-Minkley, 2010; Tissenbaum, Sheldon, & Abelson, 2019). More and more also adaptive learning systems are being adopted (Khosravi, Sadiq, & Gasevic, 2020) providing personalized instruction and feedback tailored to the needs of individual learners.

Yet, learning to program is a highly complex process and novices struggle with a wide range of difficulties (Bennedsen & Caspersen, 2007; 2019). It involves diverse cognitive activities and mental representations concerning the analysis of requirements, design, program understanding, modifying and debugging, as well as the construction of conceptual knowledge on basic operations (such as loops, conditional statements, etc.) (Rogalski & Samurçay, 1990). Learning programming can be considered an exploratory process in which software artifacts are created through an incremental problem-solving process using multiple competencies, i.e., computational concepts, practices, and perspectives (Brennan & Resnick, 2012; Pea & Kurland, 1984; (Resnick et al., 2009).

Thus, in order to be effective, instructional units aimed at teaching programming need to be systematically designed taking into consideration not only the content to be taught but also the sequencing of instruction and the idiosyncrasies of programming environments. This becomes even more important, when there are many types of environments, text-based as well as block-based, which may have different features of cognitive dimensions. Each environment has a strong relationship among its components in such a way that the notation typically cannot be used outside the environment (Green, 1989). As the order and organization of learning activities affect the way information is processed and retained (Patten, Chao, & Reigeluth, 1986), it is important to sequence the content in a way it can be most easily grasped by the student using a particular programming environment (Bruner, 1966) to improve the learners' understanding and to help them to achieve the objectives (Morrison, Ross, & Kemp, 2010). If inadequately sequenced, a learner may be overloaded, which can negatively affect learning, performance, and motivation (Sweller, van Merrienboer, & Paas, 1998). How content is sequenced is determined by the developmental level and current comprehension of the student, the instructional method, and the evolutionary structure of the knowledge on the given subject (Dede, 1986). There are many different ways to sequence content elements (Vainas et al., 2019), as, for example, by adopting a simple to complex sequence strategy according to the main types of knowledge structure (Reigeluth, 1999).

Thus, finding an optimal learning sequence is difficult, especially for different programming environments used to teach algorithms and programming concepts. Therefore, it is important to investigate the factors that lead to students learning difficulty in programming. Several studies already examine the learning of specific concepts when developing apps with App Inventor, including procedural abstraction concepts (Turbak & Mustafaraj, 2017), events (Turbak et al., 2014), programmatic sophistication (Xie, Shabir, & Abelson, 2015), effectiveness (Park & Shin, 2019), or appropriateness (Papadakis et al., 2017) of App Inventor as an educational environment. Others study the learning progression of students in computing courses in K-12, e.g., Xie and Abelson (Xie & Abelson, 2016), who analyze the relationship between the progression of skill in using App Inventor functionality and in using computational thinking concepts as learners create more apps. Other research aiming at investigating the difficulty of content in computing education analyzing how students learn to program is mostly related to higher education (Piech et al., 2012), other block-based languages, such as Scratch (Grover & Basu, 2017; Moreno-León, Robles, & Román-González, 2020; Rich et al., 2017; Rich et al., 2018; 2019; Seiter & Foreman, 2013), LaPlaya (Franklin et al., 2017), and SNAP! based environments (Lytle et al., 2019), object-oriented programming (Krugel et al., 2020), etc.

The assumption in many of these studies is that student progress can be understood through difficulties with specific programming constructs. Thus, the analysis of code created can provide insights concerning the 'difficulty' of learning certain concepts. Depending on the activities (well-defined or ill-defined) the programming ability of a person can be influenced by the volition, incentive, and opportunity to apply computing concepts in a programming environment and those factors should be taken into account.

An alternative is to regard those constructs as latent psychometric constructs and use Item Response Theory (IRT) (De Ayala, 2009) to obtain quantitative 'difficulty' estimates for each content element (De Ayala, 2009). IRT refers to a family of mathematical models that attempt to explain the relationship between latent traits (unobservable characteristics or attributes such as volition, incentive, and opportunity to apply computing concepts, including loop, conditional concepts, etc. in a programming project) and their manifestations (i.e., observed outcomes, performance such as using loop and conditional blocks in App Inventor projects). Typically applied for testing, IRT establishes a link between the properties of the items on an instrument, individuals responding to these items, and the underlying trait being measured. IRT assumes that the latent trait and items of a measure are organized in an unobservable continuum. Therefore, its main purpose focuses on establishing the individual's position on that continuum. IRT is widely used for large-scale assessments (Carlson & van Davier, 2017), such as PISA (https://www.oecd.org/pisa/) or TOEFL (https://www.ets.org/toefl).

Yet, it can also be used to obtain systematic information about the 'difficulty' of concepts and the distribution of the respective competencies among students. This can be done based on the code created by the students as an outcome of the learning process, regarding certain attributes of the code as manifestations of latent psychometric constructs according to the principles of IRT (Berges & Hubwieser, 2015; Kramer, Tobinski, & Brinda, 2016; Santos et al., 2020). The occurrence of certain concepts like loops or conditional statements can be considered as satisfiability on certain items (e.g., "the existence of loops"). Consequently, the probability of such satisfiability depending on the item 'difficulty', the estimated person abilities, and the volition, incentive, and opportunity to apply computing concepts, can be described by certain psychometric models, e.g., the Rasch or Graded Response Model. For example, Berges and Hubwieser (Berges & Hubwieser, 2015) used IRT for assessing coding abilities by analyzing the source code created as an outcome of the learning process in the context of a freshman course at university for text-based object-oriented programming. Similarly, Kramer et al. (Kramer, Tobinski, & Brinda, 2016) used IRT for assessing students' abilities in text-based object-oriented programming in an introductory programming course. Both studies focused on the Java programming language.

Several studies analyze some aspects of algorithms and programming using block-based programming environments, including e.g., Xie and Abelson (2016) analyzing skill progression of students using App Inventor, or other works comparing different block-based programming environments, such as Scratch and App Inventor for example, with regard to their effectiveness

(Park and Shin, 2019) or such as Aivaloglou and Hermans (2016) focusing on Scratch as another popular block-based environment. Yet, so far, no research focusing on the analysis of the difficulty of concepts specifically concerning the block-based programming environment App Inventor considering also practical limitations, such as measurement errors, has been found. Therefore, aiming at understanding the difficulty related to different concepts, we performed an analysis of the 'demonstrated difficulty' in App Inventor projects as part of earlier research (Alves et al., 2021). We analyzed algorithms & programming items based on the CodeMaster rubric (Alves et al., 2020a; Gresse von Wangenheim et al., 2018) by extracting them automatically from the code of App Inventor projects and, adopting IRT, we obtained first results on the 'demonstrated difficulty' of the concepts application and their distribution among the App Inventor projects (Alves et al., 2021). The objective of this article is to further detail this analysis and the interpretation of the results. In addition, we perform an analysis on the test information as well as aspects such as the standard error of measurement of the CodeMaster rubric (Alves et al., 2020a; Gresse von Wangenheim et al., 2018). Thus, the **research question** is: how the 'demonstrated difficulty' of algorithms & programming concepts is perceived in App Inventor projects? We also enhance the discussion of the results relating them to proposed curricular guidelines for computing education in K-12 and revising the adequacy of the proposed learning sequence based on the systematic data-based analysis. The results of this study can be used by instructional and curriculum designers in order to guide the sequencing of programming education in K-12. Furthermore, the results are also expected to guide further research regarding the development and evolution of the block-based programming environment App Inventor.

2 Background

In this section, we present the MIT App Inventor programming environment as well as its characteristics. We also present the mathematical definition and interpretation of the Graded Response Model of the Item Response Theory and the standard error of measurement and test information.

2.1 App Inventor

One of the most prominent block-based programming environments for computing education is App Inventor that allows creating mobile applications (MIT, 2020). It was originally provided by Google and it is currently run by the Massachusetts Institute of Technology. The current version 2.0 of App Inventor runs on a web browser (Figure 2), replacing App Inventor Classic. App Inventor is used by a wide audience, from K-12 to higher education, including end-user developers who write programs to support their primary job or hobbies (Ko et al., 2011; Wolber, Abelson, & Friedman, 2014).

A mobile app can be created in two stages with App Inventor. First, using the Designer Editor, user interface components, such as buttons, labels, etc. are configured (Figure 2). The designer also allows to specify non-visual components such as sensors, social, and media components that access mobile device features. The app's behavior is programmed in a second stage by connecting visual programming blocks in the Blocks Editor. Each block corresponds to abstract syntax tree nodes in traditional programming languages.

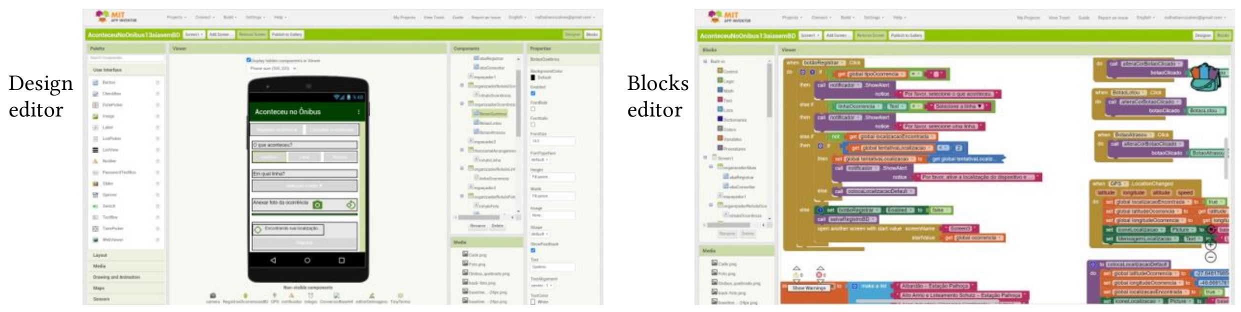


Figure 2: App Inventor Designer and Blocks Editor.

Blocks can represent standard programming concepts like loops, procedures, conditionals, etc., or conditions, events, and actions for a particular component of the app or any component. App Inventor blocks are divided into two categories: built-in blocks and component blocks. Built-in blocks are available for use in any app and refer to overall programming concepts. Component blocks include events, set and get, call methods, and component object blocks that are available for specific design components added to the app (Table 1).

Table 1: Overview of App Inventor blocks.

Type	Category	Description	Examples
Built-in block	Control	Blocks responsible for control commands including important blocks like loops, conditionals, and screen actions.	controls_while controls_if controls_closeScreen
	Logic	Blocks responsible for logic operations on variables including relational and Boolean.	logic_compare logic_operation
	Math	Blocks responsible for creating numbers and performing basic and advanced math operations.	math_add math_cos
	Text	Blocks responsible for creating and manipulating original strings.	text text_length.
	Lists	Blocks responsible for creating and manipulating original lists.	lists_create_with lists_add_items
	Colors	Blocks responsible for creating and manipulating colors.	color_red color_blue
	Variables	Blocks responsible for creating and manipulating original variables.	global_declaration lexical_variable_set
	Procedures	Blocks responsible for definition and call of original procedures.	procedures_defnoreturn procedures_callnoreturn
Component blocks	Events	Blocks responsible for specifying how a component responds to certain events, such as a button has been pressed.	component_event
	Set and Get	Blocks responsible for changing components' properties.	component_set_get
	Call Methods	Blocks responsible for calling component methods to perform complex tasks.	component_method
	Component object	Blocks responsible for getting the instance component.	component_component_block

The source code files of the App Inventor project can be exported as AIA files. An AIA file is a compressed file collection that includes a project properties file, media files that the app uses, and two files are generated for each screen in the app: a BKY file and a SCM file (Figure 3). The BKY file wraps an XML structure including all the blocks of programming used to define the behavior of the app, and the SCM file wraps a JSON structure that contains all the used visual components in the app (Mustafaraj, Turbak, & Svanberg, 2011). This AIA file can be automatically assessed with the algorithms & programming rubric (Table 2) by the CodeMaster tool.

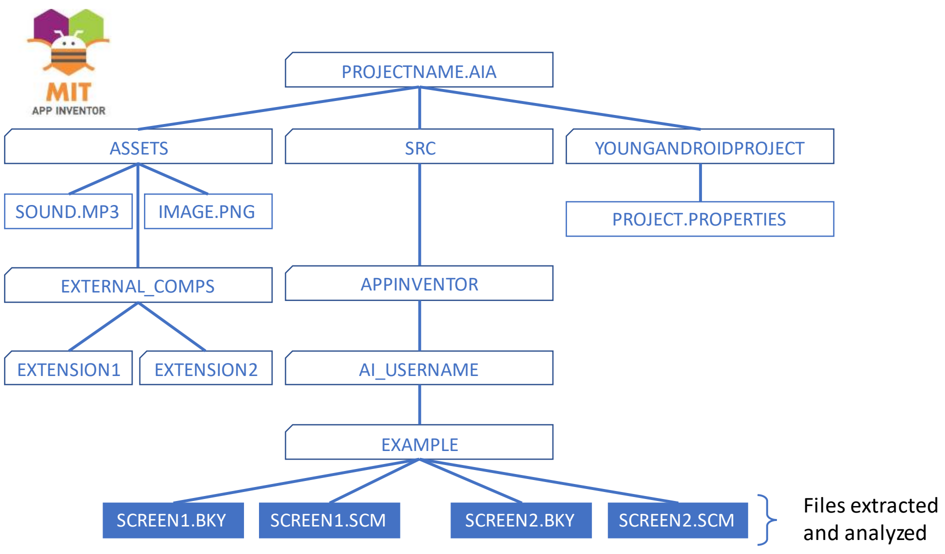


Figure 3: AIA file structure.

2.2 Item Response Theory – Graded Response Model

Item Response Theory (IRT) is a powerful tool in the quantitative processes of educational assessment as it allows analyzing item properties using falsifiable models. There are many mathematical models and to choose the adequate model the number of item response categories must be taken into account. Typically, for polytomous items, such as the CodeMaster rubric with three or more performance levels, the Graded Response Model (GRM) proposed by Samejima (Samejima, 1969) is used. The GRM assumes that an item's response categories (denoted by k) are ordered among themselves and are arranged in order from smallest (1) to largest (m_i+ 1), where m_i+ 1 is the number of categories of the i-th item. Thus, the probability (P) of an individual j with the latent trait θ to satisfy the k-th category from item i is given by the expression:

$$P_{i,k}(\theta_j) = P^{+}_{i,k}(\theta_j) - P^{+}_{i,k+1}(\theta_j)$$

In order to get the probability$P^{+}_{i,k}(\theta_j)$ an expression from the 2-parameter logistic model can be used:

$$P^{+}_{i,k}(\theta_j) = \frac{1}{1 + e^{-a_i(\theta_j - b_{i,k})}}$$

Where:

i (item) = 1, 2, …, I

k (category) = 0, 1, …, m_i

j (individual) = 1, 2, …, n

θ_j represents the latent trait of an individual j

$P^{+}_{i,k}(\theta_j)$ is the probability of an individual j with the latent trait θ to satisfy the k-th category or higher from item i

a_i represents the slope parameter of item i

$b_{i,k}$ is the position parameter of the k-th category from item i, measured on the same scale as the latent trait (θ)

From the definition as categories are arranged in order from smallest to largest, the b's values representing the position parameter should be:

$$b_{i,1} \leq b_{i,2} \leq \ldots \leq b_{i,m_i}$$

Samejima (Samejima, 1969) also defined that $P^{+}_{i,0}(\theta_j)$ — the threshold parameter for the lowest category, equals 1, and $P^{+}_{i,m+1}(\theta_j)$ — the probability of answering above the highest category, is zero:

$$P^{+}_{i,0}(\theta_j) = 1$$

$$P^{+}_{i,m+1}(\theta_j) = 0$$

As a result, the b parameters representing position can be interpreted as the threshold of passing from a lower to a higher performance level (Figure 4).

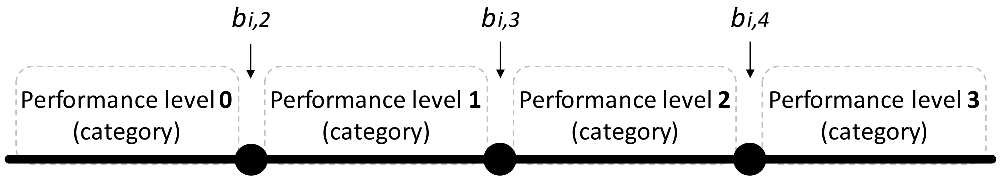

bi,2
bi,3
bi,4
Performance level 0 (category)
Performance level 1 (category)
Performance level 2 (category)
Performance level 3 (category)

Figure 4: Position parameters (b's) for items with 4 adjacent difficulty performance levels (as in the CodeMaster rubric) (Alves et al., 2021).

The position of items and their categories can be analyzed using the estimated values of b parameters on the same scale. Therefore, items that present b parameter values far below the average are considered "easy" as they result in a high probability of an average individual to satisfy the item's category. Similarly, items that present high b parameters far above average are considered "difficult", because of the low probability of an average individual to satisfy the item's category.

2.3 Standard error of measurement and test information

A measure typically used with the ICC is the item's information function. The item's information function allows analyzing how much an item (or test) contains information for the latent trait measure. The information function of an item is given by:

$$I_i(\theta) = \frac{[\frac{d}{d\theta} P_i(\theta)]^2}{P_i(\theta) Q_i(\theta)}$$

Where,

$I_i(\theta)$ is the "information" provided by item *i* at latent trait level *θ*

$P_i(\theta) = P(X_{ij} = 1|\theta)$ and $Q_i(\theta) = 1 - P_i(\theta)$

The information provided by the test is the sum of the information provided by each test's item:

$$I_i(\theta) = \sum_{i=1}^{I} I_i(\theta)$$

Another way of representing the test information function is via the standard error of estimation:

$$EP(\theta) = \frac{1}{\sqrt{I(\theta)}}$$

The standard error of estimation is defined as how accurate is the measure of a and b parameters. In Classical Test Theory, the standard error does not differ. With Item Response Theory the standard error differs depending on the region of the scale, which depends on the amount of information there exists for each region of the scale. For example, on the one hand a test composed of difficult items is expected to be excellent for differentiating top students, but, on the other hand, for the lower half of students its performance on differentiating students will be poorer because they will be confused and lost. Following this idea, an easy test will not help to discriminate knowledgeable from less knowledgeable students.

3 Research Methodology

Adopting the Goal Question Metric approach (Basili, Caldiera, & Rombach, 1994), the objective of this study is defined as to detail the results of the 'demonstrated difficulty' of algorithms & programming concepts of App Inventor projects based on the CodeMaster rubric (Alves et al., 2020a; 2021). Here the term 'demonstrated difficulty' is defined as the volition, incentive, and opportunity to apply programming concepts in an App Inventor project shared via App Inventor Gallery, on which no further background information on the authors is provided.

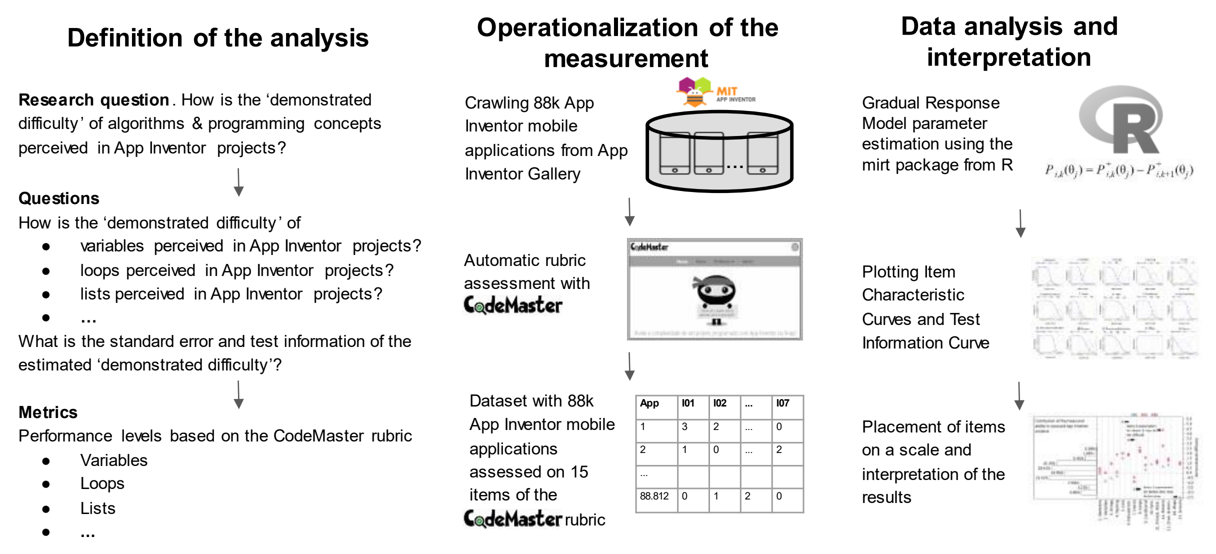

Definition of the analysis
Research question. How is the 'demonstrated difficulty' of algorithms & programming concepts perceived in App Inventor projects?
Questions
How is the 'demonstrated difficulty' of
• variables perceived in App Inventor projects?
• loops perceived in App Inventor projects?
• lists perceived in App Inventor projects?
• ...
What is the standard error and test information of the estimated 'demonstrated difficulty'?
Metrics
Performance levels based on the CodeMaster rubric
• Variables
• Loops
• Lists
• ...
Operationalization of the measurement
Crawling 88k App Inventor mobile applications from App Inventor Gallery
MIT APP INVENTOR
Automatic rubric assessment with CodeMaster
Dataset with 88k App Inventor mobile applications assessed on 15 items of the CodeMaster rubric
App | I01 | I02 | ... | I07
1 | 3 | 2 | ... | 0
2 | 1 | 0 | ... | 2
...
88.812 | 0 | 1 | 2 | 0
Data analysis and interpretation
Gradual Response Model parameter estimation using the mirt package from R
$P_{i,k}(\theta_j) = P^{+}_{i,k}(\theta_j) - P^{+}_{i,k+1}(\theta_j)$
Plotting Item Characteristic Curves and Test Information Curve
Placement of items on a scale and interpretation of the results

Figure 5: Research methodology process.

Adopting a case study design, in which we analyze 88K apps from the App Inventor Gallery, the research methodology includes the following steps (Figure 5).

3.1 Definition of the research question and analysis questions

The **research question** is defined as: How is the 'demonstrated difficulty' of algorithms & programming concepts perceived in App Inventor projects?

From this research question the following analysis questions are derived:

Analysis question 1. How is the 'demonstrated difficulty' of operators, variables, strings, naming, lists, data persistence, events, loops, conditional, synchronization, procedural abstraction, sensors, drawing and animation, maps, and screens perceived in App Inventor projects?

Analysis question 2. What is the standard error and test information of the estimated 'demonstrated difficulty'?

3.2 Operationalization of the measurement

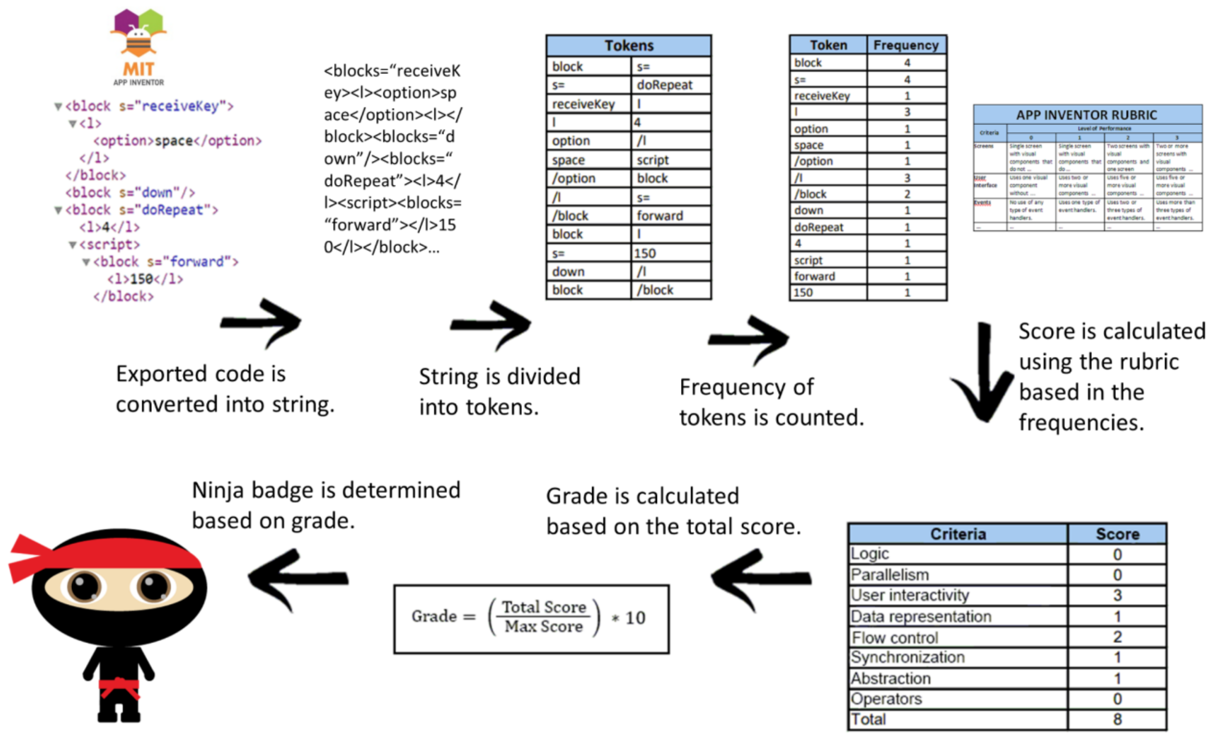
In accordance with the analysis questions, data has been collected based on the source code of App Inventor projects using the CodeMaster rubric. The CodeMaster rubric has been developed based on a systematic mapping study (Alves, Gresse von Wangenheim, & Hauck, 2019) following an instructional design process (Branch, 2010) and the procedure for rubric definition proposed by Goodrich (1997). The rubric is based on the K-12 Computer Science Framework (CSTA, 2016) as well as other rubrics and frameworks, including (Brennan & Resnick, 2012; Grover, Basu, & Schank, 2018; Sherman & Martin, 2015).

Table 2: CodeMaster rubric for assessing algorithms and programming based on the analysis of App Inventor projects (Alves et al., 2021).

Criterion	Performance Level (categories)			
	0	1	2	3
1. Operators	No operator blocks are used.	Arithmetic operator blocks are used.	Relational operator blocks are used.	Boolean operator blocks are used.
2. Variables	No use of variables.	Modification or use of predefined variables.	Creation and operation with variables.	--
3. Strings	No use of strings.	Use of string block to change the text of design components.	Creation and operation with strings.	--
4. Naming	Few or no names are changed from their defaults.	10 to 25% of the names are changed from their defaults.	26 to 75% of the names are changed from their defaults.	More than 75% of the names are changed from their defaults.
5. Lists	No lists are used.	One single-dimensional list is used.	More than one single-dimensional list is used.	Lists of tuples are used.
6. Data persistence	Data is stored only in variables or UI component properties, and does not persist when the app is closed.	Data is stored in files.	Local database is used.	Web database is used.
7. Events	No type of event handlers is used.	One type of event handlers is used.	Two or three types of event handlers are used.	More than three types of event handlers are used.
8. Loops	No use of loops. Simple loops are used.	'For each' loops with simple variables are used.	'For each' loops with list items are used.	--
9. Conditional	No use of conditionals.	Uses 'if' structure.	Uses one 'if then else' structure.	Uses more than one 'if then else' structure.
10. Synchronization	No use of timer for synchronization.	Use of timer for synchronization.	--	--
11. Procedural Abstraction	No use of procedures.	One procedure is defined and called.	More than one procedure defined.	There are procedures for code organization and re-use.
12. Sensors	No use of sensors.	One type of sensor is used.	Two types of sensors are used.	More than two types of sensors are used.
13. Drawing and Animation	No use of drawing and animation components.	Uses canvas component.	Uses ball component.	Uses image sprite component.
14. Maps	No use of city maps.	Use of a city map block.	Use of city map markers blocks.	--
15. Screens	Single screen with visual components, whose state is not	Single screen with visual components, whose state is	Three screens with visual components of which at least one is	Four screens with visual components of which at least two are

	changed programmatically.	changed programmatically.	programmed to change state.	programmed to change state.

The CodeMaster rubric for assessing algorithms and programming concepts is composed of 15 items (Table 2), including general algorithms and programming concepts, as well as, mobile algorithms and programming concepts, such as sensors, screens, etc. For each item performance levels are defined on ordinal scales, ranging from "item is not (or minimally) present" to advanced usage of the item. Aiming at the automation of the assessment, the performance levels are defined for automatically measurable characteristics based on the code of App Inventor projects. The CodeMaster rubric is considered reliable (Cronbach's alpha α=0.84) and there also exists indication of convergent validity based on the results of a correlation and factor analysis (Alves et al., 2020a) enabling a valid assessment of algorithm and programming concepts of App Inventor programs.


MIT
APP INVENTOR
Exported code is converted into string.
String is divided into tokens.
Frequency of tokens is counted.
Score is calculated using the rubric based in the frequencies.
Grade is calculated based on the total score.
Ninja badge is determined based on grade.

```
▼<block s="receiveKey">
  ▼<l>
     <option>space</option>
   </l>
 </block>
 <block s="down"/>
▼<block s="doRepeat">
   <l>4</l>
  ▼<script>
    ▼<block s="forward">
       <l>150</l>
     </block>
```

```
<blocks="receiveK
ey><l><option>sp
ace</option><l></
block><blocks="d
own"/><blocks="
doRepeat"><l>4</
l><script><blocks=
"forward"></l>15
0</l></block>...
```

Tokens	
block	s=
s=	doRepeat
receiveKey	l
l	4
option	/l
space	script
/option	block
/l	s=
/block	forward
block	l
s=	150
down	/l
block	/block

Token	Frequency
block	4
s=	4
receiveKey	1
l	3
option	1
space	1
/option	1
/l	3
/block	2
down	1
doRepeat	1
4	1
script	1
forward	1
150	1

APP INVENTOR RUBRIC

Criteria	Level of Performance			
	0	1	2	3
Screens	Single screen with visual components that do not ...	Single screen with visual components that do ...	Two screens with visual components and one screen	Two or more screens with visual components ...
User Interface	Uses one visual component without ...	Uses two or more visual components ...	Uses five or more visual components ...	Uses five or more visual components ...
Events	No use of any type of event handlers.	Uses one type of event handlers.	Uses two or three types of event handlers.	Uses more than three types of event handlers.
...	...	...	...	...

$$\text{Grade} = \left(\frac{\text{Total Score}}{\text{Max Score}}\right) * 10$$

Criteria	Score
Logic	0
Parallelism	0
User interactivity	3
Data representation	1
Flow control	2
Synchronization	1
Abstraction	1
Operators	0
Total	8

Figure 6: Assessing projects with CodeMaster.

Using publicly available and accessible projects from the App Inventor Gallery obtained in June 2018 containing the source-code from 88,864 App Inventor projects, we automatically assessed these projects using the CodeMaster tool with respect to algorithms & programming concepts by extracting them from the source code through static code analysis. The analysis is done by counting the kind and the number of command blocks used in App Inventor projects with respect to algorithms and programming concepts as defined in the rubric (Figure 6).

Out of the 88,864 projects, 88,812 were successfully assessed with the CodeMaster tool. 52 projects failed to be analyzed due to technical difficulties. The collected data were pooled in a single sample to analyze the difficulty of the items.

3.3 Data analysis and interpretation

To analyze the item properties, we use the IRT Gradual Response Model proposed by Samejima (Samejima, 1969) as presented in Section 2. This analysis is done by estimating the correspondence between an unobserved latent trait (the volition, incentive, and opportunity to apply computing concepts), and observable evidence (the assessed App Inventor projects). The dataset was analyzed using the mirt package from the R programming language (Chalmers, 2012).

In order to use the unidimensional GRM, it is necessary to assure that the instrument can be analyzed by a single predominant dimension. Therefore, we performed a parallel analysis with scree plot and full information factor analysis beforehand for verifying unidimensionality (Figure 7) (Alves et al., 2020a). Wwe used parallel analysis, which assumes that every dimension above the red line can be considered a relevant dimension. Thus, the results suggest that the instrument may contain three dimensions (Figure 7). However, there is a predominant dimension, indicating that the instrument can be analyzed by a single predominant dimension (Alves et al., 2020a). When performing the full information factor analysis (Alves et al., 2020a), we also observed that when considering a single dimension, all factor loadings were greater than 0.3, which indicates that the items are related to this predominant dimension, except by the item "Maps" which presented a 0.262 factor loading (Alves et al., 2020a). This can be explained by the fact that maps are a feature that has been added to App Inventor several years later, and, thus, may be underrepresented in our dataset. Despite the factor loading of this item being slightly less than 0.3 we decided to keep it in the analysis as this item (Alves et al., 2020a). In addition, we calculated the test variance. For acceptable calibration, the first dimension should account for at least 20% of the test variance (Reckase, 1979). We obtained a variance explained by the first-dimension of 53% characterizing the strong unidimensionality of the instrument as required by the IRT model used in this study.

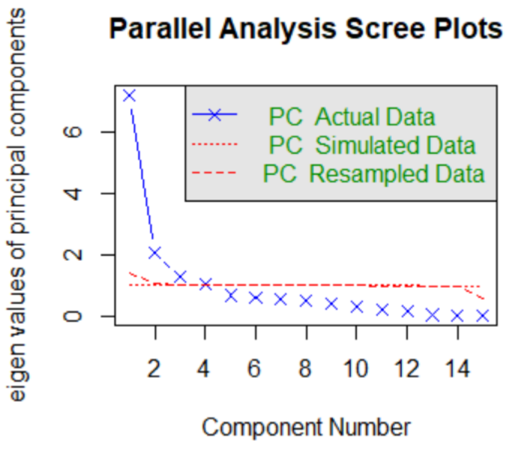


Figure 7: Parallel analysis (Alves et al., 2020a).

In IRT, a and b parameters can theoretically assume any real value between −∞ and +∞. However, a negative value for a parameter is not expected. Typically values above 1.0 are considered good, as they indicate that the item discriminates well from learners with different abilities. In this study, b parameters are the main indicators to be analyzed, as they indicate the position of the item. For b parameters, values close to or within the range [-5, 5] are expected, with negative values indicating that an item is positioned below average and positive values indicating above average.

4 Analysis

In order to detail the properties of the items in the CodeMaster rubric, we use the Gradual Response Model (GRM) (Samejima, 1969) to estimate the slope (a) parameter and position (b's) parameters for each item. The metric is established by setting population parameters to average = 0 and standard deviation = 1.

4.1 Parameters estimation

Since the CodeMaster rubric contains polytomous items, several b parameters are estimated (b_2, b_3, and b_4) to differentiate the passage from one score to another. In this regard, b_2 represents the difficulty of achieving score 1 on any item, b_3 represents the difficulty of achieving score 2 on any item, and b_4 represents the difficulty of achieving score 3 on any item. Consequently, items on a 2-point ordinal scale (without a description for category 3) also do not present a parameter b_4 (e.g., item variables).

Table 3: CodeMaster rubric for assessing algorithms and programming based on the analysis of App Inventor projects (Alves et al., 2021).

Item (i)	a (SE)	b_2 (SE)	b_3 (SE)	b_4 (SE)
1. Operators	3.08 (0.02)	-0.06 (0.01)	0.21 (0.01)	0.47 (0.01)
2. Variables	2.97 (0.02)	-0.83 (0.01)	-0.01 (0.01)	n.a.
3. Strings	1.66 (0.01)	-0.57 (0.01)	0.94 (0.01)	n.a.
4. Naming	1.68 (0.01)	-0.31 (0.01)	0.07 (0.01)	1.89 (0.01)
5. Lists	1.24 (0.01)	1.49 (0.01)	2.00 (0.02)	5.20 (0.07)
6. Data persistence	1.57 (0.02)	1.82 (0.02)	1.90 (0.02)	3.36 (0.04)
7. Events	2.88 (0.02)	-1.65 (0.01)	-0.90 (0.01)	-0.47 (0.01)
8. Loops	1.77 (0.03)	2.14 (0.02)	2.29 (0.02)	2.57 (0.03)
9. Conditional	2.32 (0.02)	0.34 (0.01)	0.80 (0.01)	1.57 (0.01)
10. Synchronization	2.81 (0.03)	0.89 (0.01)	n.a.	n.a.
11. Procedural Abstraction	3.18 (0.03)	0.99 (0.01)	1.08 (0.01)	1.19 (0.01)
12. Sensors	1.53 (0.01)	0.64 (0.01)	2.77 (0.02)	4.39 (0.05)
13. Drawing and Animation	1.32 (0.01)	0.82 (0.01)	1.25 (0.01)	1.45 (0.01)
14. Maps	0.65 (0.14)	11.36 (2.41)	n.a.	12.46 (2.66)
15. Screens	1.19 (0.01)	-2.53 (0.02)	0.89 (0.01)	1.10 (0.01)

In general, most items were well estimated, with slope (a) parameter values above 1 (Table 3). In addition, the values of the position parameters (b_2, b_3, and b_4) are within the range [-5, 5]. Only the item lists and maps presented parameter b_4 values above 5. Standard errors (SE) of each b parameter present similar results and are in low magnitude, therefore, presenting no estimation problem, with exception of the item maps, which presents standard errors in an order of magnitude higher than the standard errors of the parameters of all items. The reason may be that in our dataset map blocks are very rarely used (about 0,1% of the projects) as they had been added more recently to the App Inventor environment. Thus, the parameters of item maps cannot be used for the interpretation of positioning.

Analyzing the results, it can be inferred that the easiest category to satisfy is the first category of item 15 (screens), as it presents the smallest b parameter (b_2 = -2.53). On the other hand, obtaining three points for the item lists is more difficult than any other item as it presents the highest value for a b parameter (b_4 = 5.20). And, although item 14 (maps) presents high b parameters, this item is not considered here as it presents an estimation problem with standard errors in an order of magnitude higher than all other items' standard error.

4.2 Placement of items on a scale

Based on the estimated b parameters (b_2, b_3, and b_4) presented in Table 3 the items are placed on a wright map with a (0.1) scale, i.e., with average = 0 and standard deviation = 1 (Figure 8). The scale is an "arbitrary" scale, where the relations of order and positions between its points are most important and not necessarily its magnitude. The wright map provides a general picture by placing the positioning of demonstrated difficulty of the items on the same measurement scale as the abilities with respect to algorithms and programming concepts based on assessed App Inventor projects used as observable evidence. The left side shows the distribution of the measured ability in App Inventor projects from the most able ones at the top to the least able ones at the bottom. The right side shows items distributed from the most difficult ones at the top to the least difficult ones at the bottom (Figure 8).

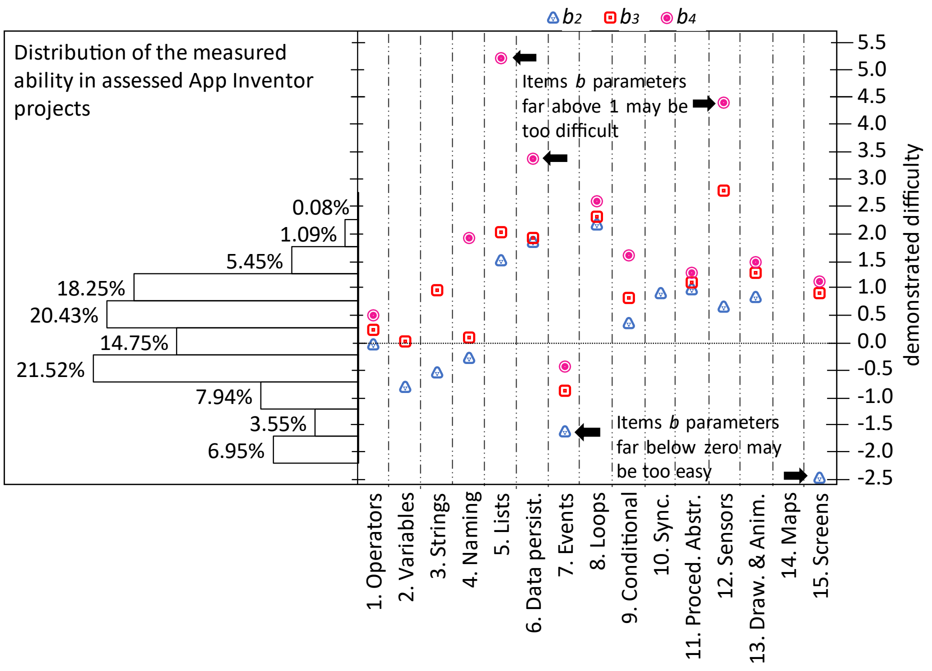
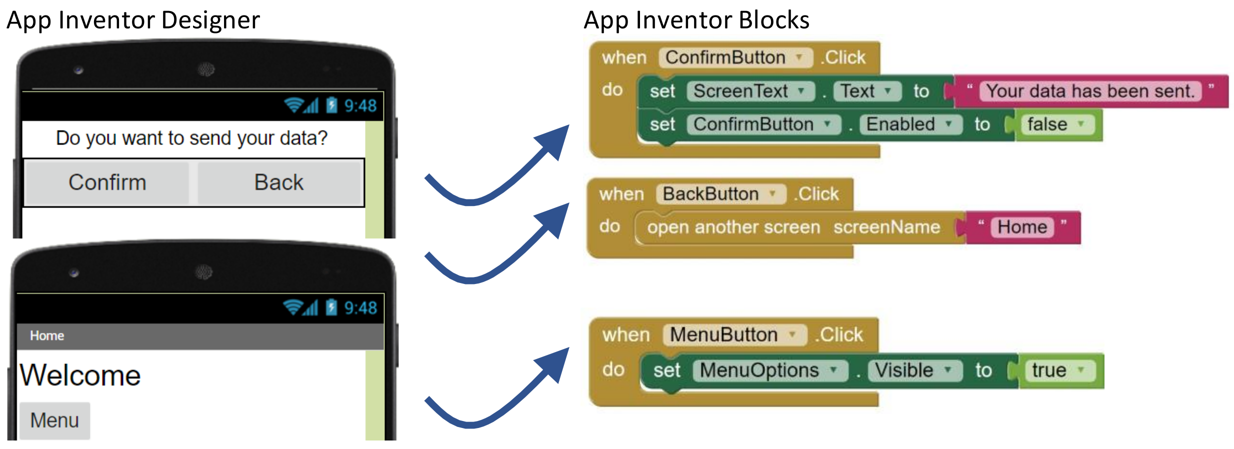


Figure 8: Wright map of the algorithms and programming items in App Inventor projects (Alves et al., 2021).

4.2.1 *Easy items*

From the placement of items on the scale, we can infer that an item with a b parameter estimated at 1.5 is 1.5 standard deviations above the average ability. Thus, such an item is more difficult than all items that are placed below point 1.5 on the scale. In the context of programming with App Inventor, among the easiest items are item 7 (events) and item 15 (screens) (Figure 8), as these items have negative b parameters far below zero. These results are semantically consistent, as App Inventor encourages unlimited use of events and creating screens that change programmatically as an essential functionality of a minimal useful mobile app (Figure 9) (Turbak et al., 2014).

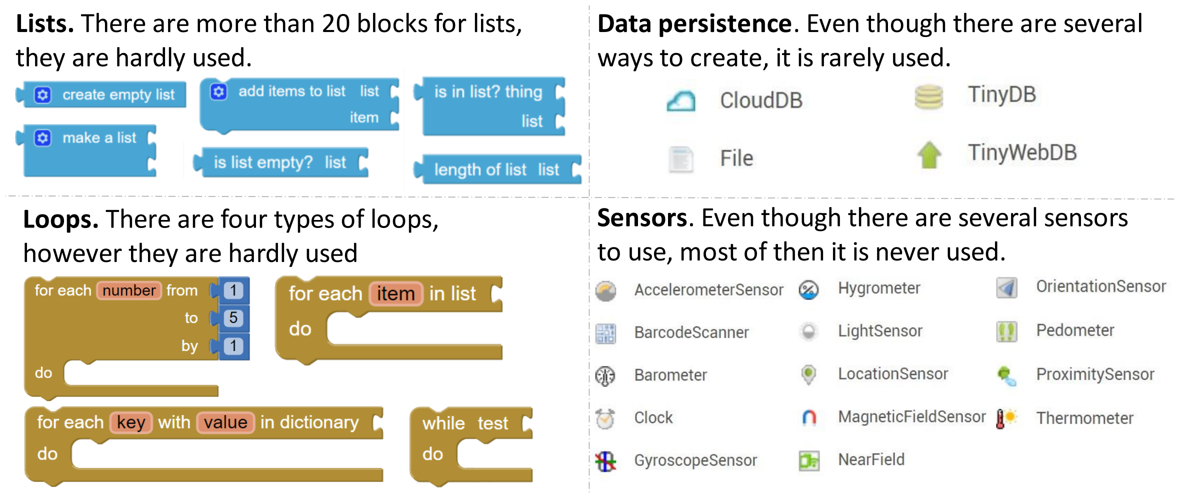


```
when ConfirmButton .Click
do  set ScreenText . Text to " Your data has been sent. "
    set ConfirmButton . Enabled to false

when BackButton .Click
do  open another screen screenName " Home "

when MenuButton .Click
do  set MenuOptions . Visible to true
```

Figure 9: Screens and button events used in App Inventor.

4.2.2 Difficult items

The most difficult items include lists, data persistence, loops, and sensors (items 5, 6, 8, and 12 respectively). For example, achieving the 3rd category with respect to lists has the highest demonstrated difficulty parameter (b_4), being the most difficult to achieve among all items. Item maps parameters are not presented because the values are out of range of the wright map [-2.5, 5.5] and are not considered here.


Lists. There are more than 20 blocks for lists, they are hardly used.

Data persistence. Even though there are several ways to create, it is rarely used.

Loops. There are four types of loops, however they are hardly used

Sensors. Even though there are several sensors to use, most of then it is never used.

Figure 10: Lists, data persistence, loops, and sensors in App Inventor.

Although the loops item is also considered difficult, it is noteworthy that loop blocks in App Inventor programs are rarely used (Figure 10), because many iterative processes that would be expressed with loops in other programming languages are expressed as an event that performs a single step of the iteration every time it is triggered (Turbak et al., 2014; Xie & Abelson, 2016). Thus, the demonstrated difficulty of loops may be poorly represented in the App Inventor dataset, as more than 94% of apps are assessed with 0 points regarding loops (see Figure 11). In other visual programming environments, such as Scratch, the usage of this concept and consequently the demonstrated difficulty may be different.

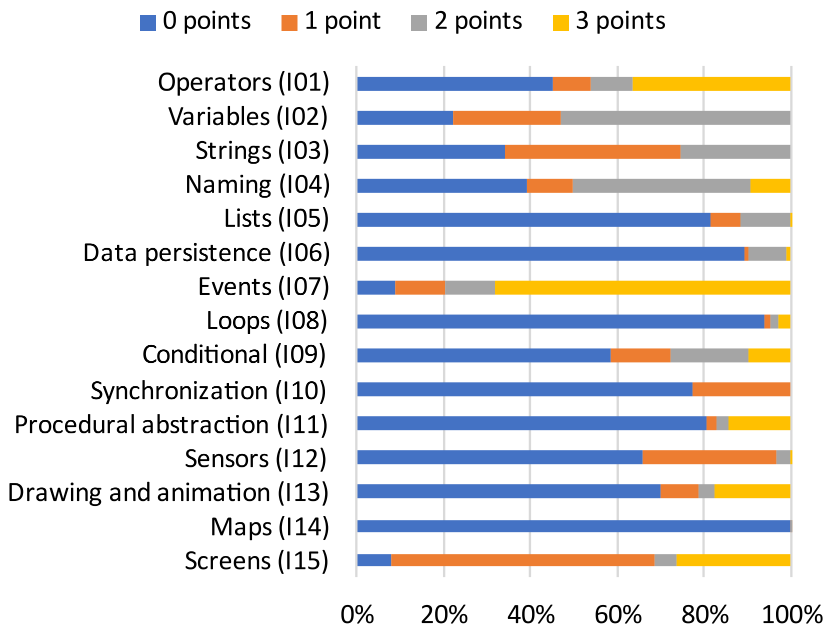


Figure 11: Frequency of the performance level score for each item (Alves et al., 2021).

4.3 Item Characteristic Curves

According to the estimated b parameters, the Item Characteristic Curves (ICC) for each item are plotted (Figure 12). While the theoretical range of a latent trait is from negative infinity to positive infinitive, for practical considerations the range of values can be limited from -4 (low) to +4 (high) on the x-axis. Thus, items with low demonstrated difficulty are placed closer to low latent trait values and high demonstrated difficulty items are placed closer to the high latent trait. Therefore, items that have high b parameters, which indicate high demonstrated difficulty, such as items 5 (lists), 6 (data persistence), 8 (loops), and 12 (sensors) present the curves dislocated to the right (Figure 12). In the same way, items with low demonstrated difficulty, such as items 7 (events) and 15 (screens) present curves dislocated to the left. Although item 14 (maps) presented the highest difficulty parameters (Table 3), and the "curve" is hidden above latent trait 4.0, these parameters presented standard errors in a high order of magnitude (Table 3). Consequently, the ICC for maps cannot be used for difficulty interpretation purposes.

Items with only three performance levels, such as item 2 (variables), 3 (strings), and 14 (maps) have fewer curves than the other items (Figure 12). This is due to the impossibility of satisfying a fourth category as no such performance level has been defined for these items (see Table 2). This also applies to items with two performance levels, such as item 10 (synchronization).

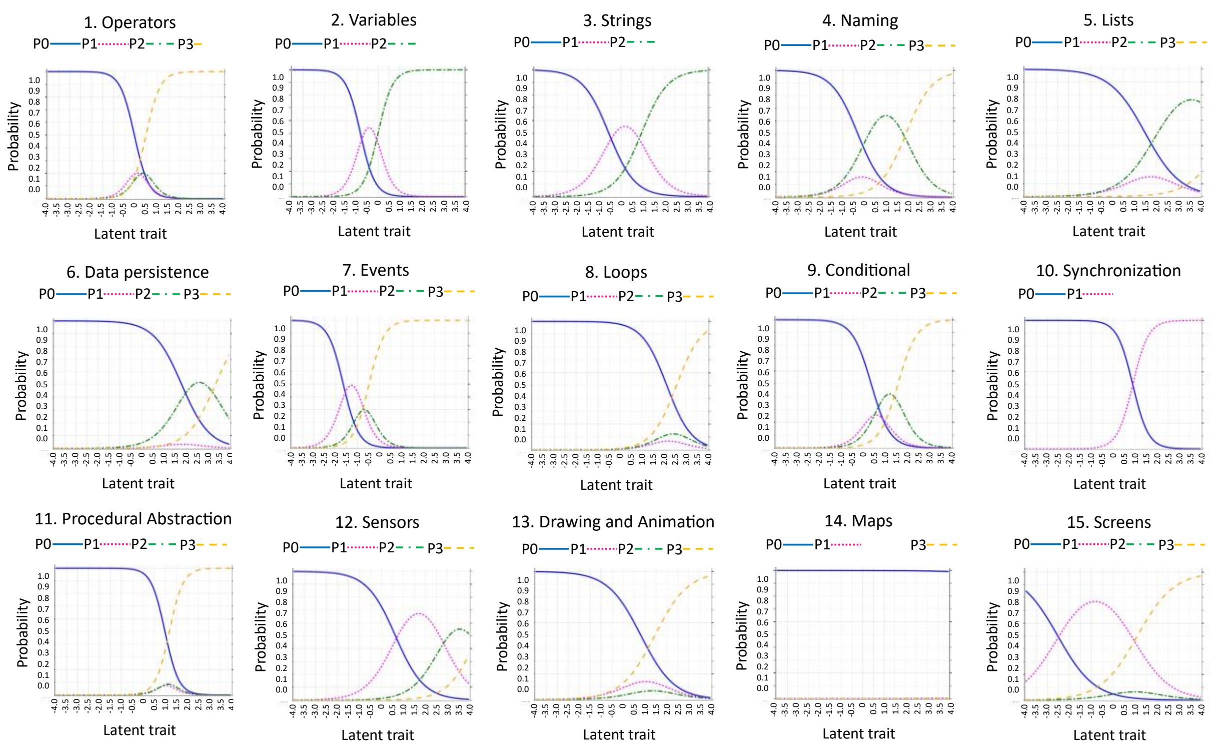


Figure 12: Item Characteristic Curves for each item (Alves et al., 2021).

The P0 curve refers to the probability of satisfying category zero (or achieving score 0) for any item given the latent trait in the x-axis (Figure 12). Similarly, the P1, P2, and P3 curves refer to the probability of achieving scores 1, 2, and 3 respectively given the latent trait in the x-axis (Figure 12). Thus, the P0 is close to 1.0 for low latent trait values, as projects assessed with a "low" latent trait (the volition, incentive, and opportunity to apply computing concepts) have a probability close to 100% of achieving score 0. For example, presenting a latent trait less than -1.0 results in a bigger probability of achieving score 0 in item 3 (strings) than score 1. On the other hand, P0 is close to 0 for high latent trait values, as projects assessed with a "high" latent trait have a probability close to 0% of achieving score 0. For example, presenting a latent trait greater than the average (0.0) results in having a bigger probability in P1 for item 3 (strings), which is related to achieving score 1, than in P0, which is related to achieving score 0 for the same item (Figure 12).

Some items' curves are more attached than others, for example, the curves of item 2 (variables) are more attached than the curves of item 3 (strings) (Figure 12). This occurs because b parameters of variables are less distant than b parameters of strings, as the distance between b_2 and b_3, i.e., b_3 - b_2, of the item variables is 0.82, while their distance for the item strings is 1.51 (Table 3). This means that is easier to progress from "modifying or using predefined variables" to "creating and operating with variables", than progressing from "using string block to change the text of design component" to "creating and operating with strings", as defined in the CodeMaster rubric (Table 2). This is expected as operating with variables is easier than operating with strings, as strings can be broken apart to make new strings, or put together and make longer strings (LEGO, 2018). Similarly, this can also be observed regarding item 1 (operators) and 11 (sensors).

4.4 Standard error of measurement and test information

In order to analyze standard error and test information of the CodeMaster rubric, we used the function *testinfo* from the mirt package from the R programming language (Chalmers, 2012). We considered all 15 items in the CodeMaster rubric and used a practical range of [-4, 4] on the x-axis (Figure 13).

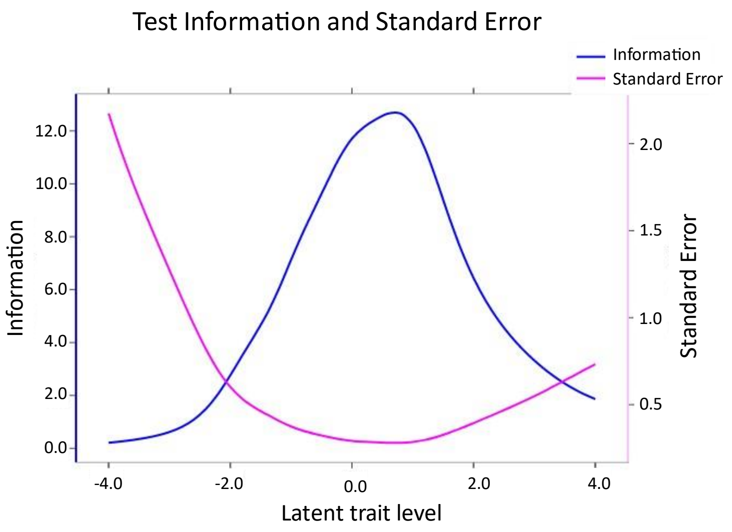


Figure 13: Test Information Curve and Standard Error.

The test information curve is represented by the blue line, while the pink line represents the curve of the standard error (Figure 13). At the extremes of the θ (latent trait) levels, the test produces more information error than legitimate information, because the error curve exceeds the information curve. Below -2.0 and above +3.0 the error curve surpasses the information curve for the CodeMaster rubric (Figure 13). Thus, items that have b parameters estimated at latent trait levels out of the range [-2.0, 3.0] require careful analysis as they may not have real use considering that there may be a measurement error. This is reasonable and may be due to a lesser use of certain items (e.g., lists) not being required in particular App Inventor projects, resulting in fewer data for items considered difficult.

In general, the CodeMaster rubric covers easy, medium and more difficult items with low standard error of measurement. The test information is not symmetric and it is dislocated to the right - to the medium-difficult items (latent trait above 0.0). This is expected as most items have b parameters estimated above 0.0 (Table 3).

5 Discussion

These results of the analysis provide an insight into the degree of demonstrated difficulty concerning algorithms and programming in the context of the development of apps with App Inventor (Figure 14).

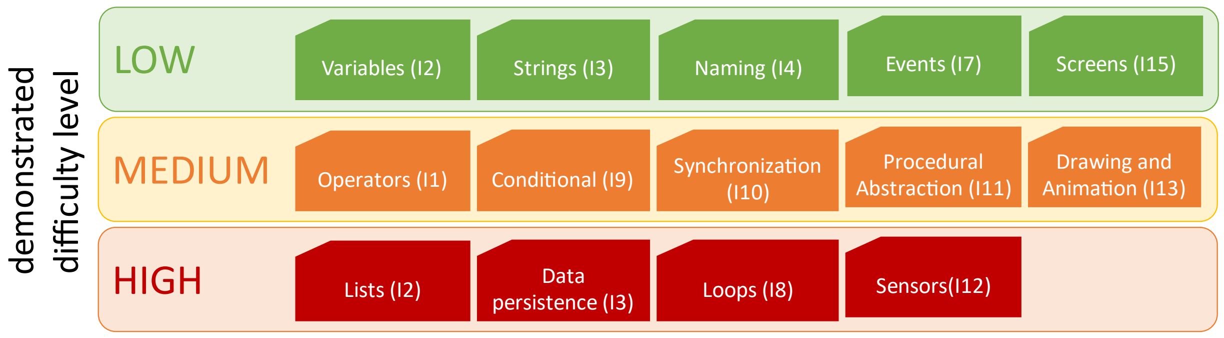


Figure 14: Demonstrated difficulty level for each item in App Inventor projects.

Variables, strings, naming, events, and screens (items 2, 3, 4, 7, and 15 respectively) are the easiest concepts when programming with App Inventor, as all probabilities curves are dislocated to the left (Figure 12). Items with medium demonstrated difficulty include operators, conditional, synchronization, procedural abstraction, and drawing and animation (item 1, 9, 10, 11, and 13 respectively) as the probability curves are close to the average (0.0) latent trait. The most difficult items are lists, data persistence, loops, and sensors (item 5, 6, 8, and 12 respectively) as the probability curves are dislocated to the right (Figure 12). This also confirms results presented by Xie and Abelson (2016) indicating, for example, that apps that require data persistence (e.g., databases) represent more advanced artifacts. Some of the items with estimated high difficulty may be influenced by its infrequent use in App Inventor projects, e.g., loops rather than indicating the difficulty of understanding loops in general, and may be different when using other visual programming environments.

These results can be used as a systematic basis supported by data for the sequencing of computing instruction in K-12 when teaching the development of apps with App Inventor. Rather than proposing such expected learning progression based on the opinions of stakeholders, the results of our analysis provide an evidence-based approach based on data from practice. Based on the results of the scale placement (Figure 8) and the detailed demonstrated difficulty ICC (Figure 12), teaching algorithms and programming concepts with App Inventor should thus start with the creation of screens and events as well as the usage of strings, and variables and naming. Then on the next stage, the instructional design could cover operators and conditionals as well as synchronization and procedural abstraction, while only more advanced students should be presented with problems requiring lists, data persistence, and sensors, allowing them to follow a smooth pathway as they progress toward mastery of the skills with scaffolding support.

The results can also be used to analyze the sequencing of the content proposed by frameworks, guidelines, etc. In this context, we analyzed the appropriateness of curricular guidelines proposed by the SBC (2018) as well as the K-12 Computer Science Framework from the CSTA (2016; 2017) regarding the sequencing of the learning objects proposed for K-12 (Alves et al., 2019; Alves et al., 2020b). The results demonstrate that the difficulty of some concepts can be explained by their particular characteristics, but may also be due to the characteristics of App

Inventor as a programming tool. Although the analysis demonstrates the alignment of the content sequencing of some items with their respective difficulty, we also observed that some concepts related to algorithms and programming are not explicitly covered by the CSTA framework (2016; 2017) nor the SBC guidelines (2018). The CSTA framework (2017) presents a sequencing more aligned to the proficiency scale created according to the estimation of the content's difficulty. Different from the SBC guidelines, the CSTA framework (2016; 2017) proposes teaching items considered easier, such as variables and strings, before more difficult items, such as abstraction. Items considered complex, such as lists, are covered only in the final years of elementary school or in high school and not in the initial years of elementary school as proposed by SBC (2018).

In this respect, the findings of this study provide a systematic basis for the review of curriculum guidelines in order to adjust the sequencing and placement of concepts in accordance to their observed difficulty in order to provide a better support for the student's learning progress. The results can also be used in order to guide the development and evolution of the block-based programming environment as well as instructional units, improving support and guidance provided specifically for concepts observed as more difficult.

5.1 Threats to validity

Our study is subject to several threats to validity which have been handled in order to be minimized. One risk is related to grouping data as App Inventor projects come from various contexts in the worldwide App Inventor community, and no additional information about the creator history is available in the App Inventor Gallery. Another factor that may influence the usage of commands may be the tutorials and instructional units typically used as well as a considerable number of very simple App Inventor projects at the App Inventor Gallery. However, as typically App Inventor is used by novices and/or in the context of computing education in K-12, we consider this acceptable considering the large-scale sample. Another threat regarding the possibility of generalizing the results is related to the sample size and projects from only one repository. Yet, this risk is minimized by using a significant number of apps (over 88,000) from the repository for App Inventor projects used by contributors from around the world.

Another risk is that the analysis based on the created code does not only assess whether a learner is able to achieve a certain item of the rubric, but also whether the learner is willing to do so. Therefore, we also restrict the interpretation of the "difficulty" of items in this study to the "demonstrated difficulty" defined as the volition, incentive, and opportunity to apply programming concepts in an App Inventor project shared via App Inventor Gallery. For measurement we used the CodeMaster rubric that was systematically defined and validated using Classical Test Theory with results reported in Alves et al. (2020a), indicating the reliability and validity of the rubric for the assessment of algorithm and programming concepts of App Inventor projects. Automating the assessment of the App Inventor projects with the CodeMaster tool further reduced the risks of reliability issues which may have been caused through manual assessment. In order to mitigate threats concerning the research methodology, we adopted the GQM approach for measurement (Basili, Caldiera, & Rombach, 1994) and selected appropriate statistical techniques for the analysis (De Ayala, 2009), performing also necessary tests with respect to the characteristics of the dataset to assure their adequacy.

6 Conclusions

In this article we presented the results of an analysis of the demonstrated difficulty of general mobile algorithms and programming concepts based on App Inventor projects. Considering the difficulty of items, we identified that events and variables are the easiest items when programming with App Inventor, while the most difficult items are loops, data persistence, and lists. Overall, App Inventor has proven effective for teaching development of mobile applications. However, several studies also indicate that good visual design is also important. Thus, besides teaching only algorithms and programming, concepts related to visual design can help students to create mobile applications in line with style guide guidelines (Solecki et al., 2020a; Ferreira et al., 2020).

Comparing these results on algorithms and programming to analyses based on other block-based languages, e.g., Scratch (Aivaloglou & Hermans, 2016), we can observe that the difficulty of achieving performance levels of certain items may depend on the specific programming language, and, thus, the programming environment to be adopted has to be explicitly considered in the instructional design of computing education. For example, there exists an expressive difference concerning loops in Scratch and App inventor, as many iterative processes that are expressed through loops in Scratch are represented using event blocks in App Inventor. In addition, the limitation of the use of App Inventor threads discourages frequent use of loops, as they may crash the application (Turbak et al., 2014). Consequently, loops may be easier to be taught using Scratch.

The results of this analysis can be used to systematically discuss and improve the sequencing of instructional units for teaching algorithms and programming with App Inventor by adopting scaffolding techniques in future works. The findings can also support the decision on the selection of block-based programming environments as part of the instructional units depending on the intended learning objects.

Furthermore, our results indicate improvement opportunities of existing curriculum guidelines. For example, some concepts related to algorithms and programming are not explicitly included in the SBC guidelines for K-12 computing education or are positioned too early, e.g., lists, being considered one of the most difficult items. Thus, adjusting curriculum guidelines accordingly may facilitate the student's learning progress as well as improve the learning outcomes contributing positively to the popularization of computing knowledge in K-12.

As future work we also plan to extend the analysis of the difficulty of learning to other knowledge areas and skills, including, for example, the learning of GUI design or Software Engineering concepts when developing mobile applications with App Inventor as well as soft skills such as creativity. In addition, similar studies using IRT can also be conducted with regard to other visual programming environments, such as Scratch and Snap!, enabling also a comparison of the demonstrated difficulty of concepts among different block-based environments.

Acknowledgements

We would like to thank all researchers from the MIT App Inventor team, who provided support for the access to the App Inventor Gallery. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil* (CAPES) - Finance Code 001 and by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil* (CNPq) - Grant No.: 302149/2016-3.

Extended Awarded Article

This publication is an extended version of an awarded paper at the Brazilian Symposium on Computing Education (EduComp 2021), entitled *"An Item Response Theory Analysis of*

Algorithms and Programming Concepts in App Inventor Projects", DOI: 10.5753/educomp.2021.14466.

References

Aivaloglou, E., Hermans, F. (2016). How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the ACM Conference on International Computing Education Research*, Melbourne, Australia, 8–12 September 2016; pp. 53–61. doi: 10.1145/2960310.2960325 [GS Search]

Alves, N. da C., Gresse von Wangenheim, C., & Hauck, J. C. R. (2019). Approaches to assess computational thinking competences based on code analysis in K-12 education: A systematic mapping study. *Informatics in Education*, 18(1), 17-39. doi: 10.15388/infedu.2019.02 [GS Search]

Alves, N. da C., von Wangenheim, C. G., Hauck, J., Borgatto, A., & Andrade, D. (2019, November). Uma análise do sequenciamento pedagógico no ensino de computação na educação básica. *Brazilian Symposium on Computers in Education* (*Simpósio Brasileiro de Informática na Educação-SBIE*) (Vol. 30, No. 1, p. 1). doi: 10.5753/cbie.sbie.2019.1 [GS Search]

Alves, N. da C., von Wangenheim, C. G., Hauck, J. C. R., & Borgatto, A. F. (2021, April). An Item Response Theory Analysis of Algorithms and Programming Concepts in App Inventor Projects. In *Anais do Simpósio Brasileiro de Educação em Computação* (pp. 01-11). Jataí, Goiás. SBC. doi: 10.5753/educomp.2021.14466 [GS Search]

Alves, N. da C., Gresse von Wangenheim, C., Hauck, J. C. R., & Borgatto, A. F. (2020a, February). A large-scale evaluation of a rubric for the automatic assessment of algorithms and programming concepts. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 556-562). Association for Computing Machinery, New York, NY, USA, 556–562. doi:10.1145/3328778.3366840 [GS Search]

Alves, N. da C., von Wangenheim, C. G., Hauck, J. C. R., Borgatto, A. F., & Andrade, D. F. (2020b). An Item Response Theory Analysis of the Sequencing of Algorithms & Programming Concepts. Proceedings of International Conference on Computational Thinking Education (CoolThink@). Hong Kong: The Education University of Hong Kong. [GS Search]

De Ayala, R. J. (2009). *The theory and practice of item response theory*. Guilford Press. [GS Search]

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*, 528-532, John Wiley & Sons. [GS Search]

Bennedsen, J., & Caspersen, M. E. (2007). Failure Rates in Introductory Programming, *ACM SIGCSE Bulletin*, 39(2), 32-36. doi:10.1145/1272848.1272879 [GS Search]

Bennedsen, J. & Caspersen, M. E. (2019). Failure rates in introductory programming: 12 years later. *ACM Inroads* 10(2), 30–36. doi:10.1145/3324888 [GS Search]

Berges, M., & Hubwieser, P. (2015, June). Evaluation of Source Code with Item Response Theory. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 51-56), Association for Computing Machinery, New York, NY, USA. doi:10.1145/2729094.2742619 [GS Search]

Branch, R. M. (2010). *Instructional Design: The ADDIE Approach*. Springer. [GS Search]

Brennan, K. & Resnick, M. (2012, April). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the Annual Meeting of the American Educational Research Association*, Vancouver, Canada (Vol. 1, p. 25). [GS Search]

Bruner, J. S. (1966). *Toward a theory of instruction* (Vol. 59). Harvard University Press. [GS Search]

Carlson, J. E., & von Davier, M. (2017). Item Response Theory. In *Advancing Human Assessment* (pp. 133-178), eds. Bennet & van Davier, Springer, Cham. [GS Search]

CAS. (2015). *Computing at School*. Retrieved October 29, 2021, from https://www.computingatschool.org.uk/

Chalmers, R. P. (2012). Mirt: A multidimensional item response theory package for the R Environment. *Journal of Statistical Software*, 48(6), 1–29. [GS Search]

CSTA. (2016). *K-12 Computer Science Framework*. Retrieved October 29, 2021, from https://k12cs.org/ [GS Search]

Dede, C. (1986). A review and synthesis of recent research in intelligent computer-assisted instruction. *International Journal on Man-Machine Studies*, 24(4), 329-353. doi:10.1016/S0020-7373(86)80050-5 [GS Search]

Fee, S. B. & Holland-Minkley, A. M. (2010). Teaching computer science through problems, not solutions. *Computer Science Education*, 20(2), 129-144. doi:10.1080/08993408.2010.486271 [GS Search]

Ferreira, M. N. F.; da Cruz Pinheiro, F.; Gresse Von Wangenheim,C.; Missfeldt Filho, R.; Hauck, J. C. R. Ensinando Design de Interface de Usuário de Aplicativos Móveis no Ensino Fundamental. *Revista Brasileira de Informática na Educação*, vol. 28, 2020. doi: 10.5753/rbie.2020.28.0.48 [GS Search]

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D., & Harlow, D. (2017, March). Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. In *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 231-236). Association for Computing Machinery, New York, NY, USA, 231–236. doi:10.1145/3017680.3017760 [GS Search]

Goodrich, H. (1997). Understanding Rubrics. *Educational Leadership*, 54(4), 14–17. [GS Search]

Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.), People and Computers V. Cambridge, UK: Cambridge University Press, pp. 443-460. [GS Search]

Gresse von Wangenheim, C., Hauck, J. C. R., Demetrio, M. F., Pelle, R., da C. Alves, N. da C., Barbosa, H., Azevedo, L. F. (2018). CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs. *Informatics in Education*, 17(1), 117-150. doi:10.15388/infedu.2018.08 [GS Search]

Grover, S., & Basu, S. (2017, March). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean Logic. In *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 267–272). Association for Computing Machinery. New York, NY, USA. doi:10.1145/3017680.3017723 [GS Search]

Grover, S., Basu, S., & Schank, P. (2018, February). What We Can Learn About Student Learning From Open-Ended Programming Projects in Middle School Computer Science. In

Proceedings of the 49th ACM Technical Symposium on Computer Science Education (pp. 999-1004). Association for Computing Machinery, NY, USA. doi:10.1145/3159450.3159522 [GS Search]

Grover, S. & R. Pea. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43. doi:10.3102/0013189X12463051 [GS Search]

Hubwieser, P., Giannakos, M. N., Berges, M., Brinda, T., Diethelm, I., Magenheim, J., Pal, Y., Jackova, J., & Jasute, E. (2015). A Global Snapshot of Computer Science Education in K-12 Schools. In *Proceedings of the ITiCSE on Working Group Reports* (pp. 65-83). Association for Computing Machinery, New York, NY, USA, 65–83. doi:10.1145/2858796.2858799 [GS Search]

Khosravi, H., Sadiq, S., & Gasevic, D. (2020, February). Development and Adoption of an Adaptive Learning System. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 58-64), Association for Computing Machinery, New York, NY, USA, 58–64. doi:10.1145/3328778.3366900 [GS Search]

Ko, J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., & Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3), Article 21. doi:10.1145/1922649.1922658 [GS Search]

Kramer, M., Tobinski, D. A., & Brinda, T. (2016, November). On the way to a test instrument for object-oriented programming competencies. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 145-149). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2999541.2999544 [GS Search]

Krugel, J., Hubwieser, P., Goedicke, M., Striewe, M., Talbot, M., Olbricht, C., Schypula, M. & Zettler, S. (2020, April). Automated Measurement of Competencies and Generation of Feedback in Object-Oriented Programming Courses. In *Proceedings of the IEEE Global Engineering Education Conference (EDUCON)* (pp. 329-338), Porto, Portugal. doi:10.1109/EDUCON45650.2020.9125323 [GS Search]

LEGO. (2018). Lego Education Documentation. Retrieved October 29, 2021, from: https://makecode.mindstorms.com/types/string

Li, I., Turbak, F., & Mustafaraj, E. (2017, October). Calls of the Wild:Exploring Procedural Abstraction in App Inventor. In *Proceedings of the IEEE Blocks and Beyond Workshop* (pp. 79-86). Raleigh, NC, USA. doi:10.1109/BLOCKS.2017.8120417 [GS Search]

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41, C, 51–61. doi:10.1016/j.chb.2014.09.012 [GS Search]

Lytle, N., Cateté, V., Boulden, D., Dong, Y., Houchins, J., Milliken, A., Isvik, A., Bounajim, D., Wiebe, E., & Barnes, T. (2019, July). Use, Modify, Create: Comparing Computational Thinking Lesson Progressions for STEM Classes. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 395-401). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3304221.3319786 [GS Search]

MIT. (2020). MIT App Inventor. About us. Retrieved October 29, 2021, from http://appinventor.mit.edu/explore/about-us.html

Moreno-León, J., Robles, G., & Román-González, M. (2020). Towards data-driven learning paths to develop computational thinking with Scratch. *IEEE Transactions on Emerging Topics in Computing*, 8(1), 193-205. doi:10.1109/TETC.2017.2734818 [GS Search]

Morrison, G. R., Ross, S. M., & Kemp, J. E. (2010). *Designing Effective Instruction*, 6th ed. John Wiley & Sons. [GS Search]

Mustafaraj, E., Turbak, F., & Svanberg, M. (2017, May). Identifying Original Projects in App Inventor. In *Proceedings of the 30th International Florida Artificial Intelligence Research Society Conference*, Marco Island, FL, USA. 567-572. [GS Search]

Papadakis, S., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2017). The appropriateness of Scratch and App Inventor as educational environments for teaching introductory programming in primary and secondary education. *International Journal of Web-Based Learning and Teaching Technologies*, 12(4), 58-77. doi:10.4018/IJWLTT.2017100106 [GS Search]

Park, Y., & Shin, Y. (2019). Comparing the Effectiveness of Scratch and App Inventor with Regard to Learning Computational Thinking Concepts. *Electronics*, 8(11), 1269. doi:10.3390/electronics8111269 [GS Search]

van Patten, J., Chao, C. I., & Reigeluth, C. M. (1986). A review of strategies for sequencing and synthesizing instruction. *Review of Educational Research*, 56(4), 437-471. doi:10.3102/00346543056004437 [GS Search]

Patton, E. W., Tissenbaum, M., & Harunani, F. (2019). MIT App Inventor: Objectives, Design, and Development. Kong SC., Abelson H. (eds), *Computational Thinking Education* (pp. 31-49). Springer, Singapore. doi:10.1007/978-981-13-6528-7_3 [GS Search]

Pea, R. D. & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*, 2(2), 137–168. doi:10.1016/0732-118X(84)90018-7 [GS Search]

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012, February). Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 153-160). Association for Computing Machinery, New York, NY, USA, 153–160. doi:10.1145/2157136.2157182 [GS Search]

Reckase, M. D. (1979). Unifactor latent trait models applied to multifactor tests: Results and implications. *Journal of educational statistics*, Sage Publications CA: Thousand Oaks, 4(3), 207–230. [GS Search]

Reigeluth, C. M. (1999). The Elaboration theory: Guidance for scope and sequence decision. In C. Reigeluth (ed.) *Instructional-Design Theories and Models* (vol.II), Erlbaum Associates. [GS Search]

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM* 52(11), 60–67. doi:10.1145/1592761.1592779 [GS Search]

Rich, K. M., Strickland, C. T., Binkowski, T. A., Moran, T. A., & Franklin, D. (2017). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the ACM Conference on International Computing Education Research* (pp. 182-190), Association for Computing Machinery, New York, NY, USA. doi:10.1145/3105726.3106166 [GS Search]

Rich, K. M., Strickland, C. T., Binkowski, T. A., & Franklin, D. (2018). Decomposition: A K-8 computational thinking learning trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (pp. 124-132), Association for Computing Machinery, New York, NY, USA. doi:10.1145/3230977.3230979 [GS Search]

Rich, K. M., Strickland, C. T., Binkowski, T. A., & Franklin, D. (2019). A K-8 debugging learning trajectory derived from research literature. In *Proceedings of the 50th ACM Technical*

Symposium on Computer Science Education (pp. 745-751), Association for Computing Machinery, New York, NY, USA. doi:10.1145/3287324.3287396 [GS Search]

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J.M.Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (eds.), *Psychology of programming*, Academic Press. doi:10.1016/B978-0-12-350772-3.50015-X [GS Search]

Samejima, F. A. (1969). Estimation of latent ability using a response pattern of graded scores. *Psychometric Monograph*, 34(4), 2-17. [GS Search]

Santos, J. S., Andrade, W. L., Brunet, J., & Araujo Melo, M. R. (2020, October). A Systematic Literature Review of Methodology of Learning Evaluation Based on Item Response Theory in the Context of Programming Teaching. In *Proceedings of the IEEE Frontiers in Education Conference (FIE)* (pp. 1-9). Uppsala, Sweden. doi:10.1109/FIE44824.2020.9274068 [GS Search]

SBC. (2018). Brazilian Computer Society *Guidelines for Computing Education in K-12 (Diretrizes para ensino de Computação na Educação Básica)*. Retrieved October 29, 2021, from https://www.sbc.org.br/educacao/diretoria-de-educacao-basica

Seiter, K., & Foreman, B. (2013, August). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research* (pp. 59-66), Association for Computing Machinery, New York, NY, USA, 59–66. doi: 10.1145/2493394.2493403 [GS Search]

Sherman, M., & Martin, F. (2015). The assessment of mobile computational thinking. *Journal of Computing Sciences in Colleges*, 30(6), 53–59. [GS Search]

Sweller, J., van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive Architecture and Instructional Design. *Educational Psychology Review*, 10(3), 251–296. doi:10.1023/A:1022193728205 [GS Search]

Tissenbaum, M., Sheldon, J., & Abelson, H. (2019). From Computational Thinking to Computational Action. *Communications of the ACM*, 62(3), 34-36. doi:10.1145/3265747 [GS Search]

Turbak, F., Sherman, M., Martin, F., Wolber, D., & Pokress, S. C. (2014). Events First Programming in App Inventor. *Journal of Computing Sciences in Colleges*, 29(6), 81-89. [GS Search]

Vainas, O., Ben-David, Y., Gilad-Bachrach, R., Ronen, M., Bar-Ilan, O., Shillo, R., Lukin, G., Sitton, D. (2019). Staying in The Zone: Sequencing Content in Classrooms Based on The Zone of Proximal Development. In *Proceedings of the 12th International Conference on Educational Data Mining* (pp. 659-662), Montreal, Canada. [GS Search]

Webb, M., Davis, N., Bell, T., Katz, Y. J., Reynolds, N., Chambers, D. P., & Sysło, M. M. (2017). Computer science in K-12 school curricula of the 2lst century: Why, what and when?. *Education and Information Technologies*, 22(2), 445–468. doi:10.1007/s10639-016-9493-x [GS Search]

Weintrop, D. (2019). Block-based Programming in Computer Science Education. *Communications of the ACM*, 62(8), 22-25. doi: 10.1145/3341221 [GS Search]

Wolber, D., Abelson, H., & Friedman, M. (2014). Democratizing Computing with App Inventor. *GetMobile: Mobile Computing and Communications*. 18(4), 53–58. doi:10.1145/2721914.2721935 [GS Search]

Xie, B., & Abelson, H. (2016, September). Skill progression in MIT app inventor. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 213-217). Cambridge, GB. doi:10.1109/VLHCC.2016.7739687 [GS Search]

Xie, B., Shabir, I., & Abelson, H. (2015). Measuring the programmatic sophistication of app inventor projects grouped by functionality. Retrieved October 29, 2021 from https://dspace.mit.edu/bitstream/handle/1721.1/98913/measuring%20usability.pdf [GS Search]