UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PEDRO HENRIQUE SOUSA DE MORAES

Willow: A Tool for Interactive Data Structures and Algorithms Visualization

Recife

2020

PEDRO HENRIQUE SOUSA DE MORAES

Willow: A Tool for Interactive Data Structures and Algorithms Visualization

Dissertation presented to the Postgraduate Program in Computer Science at the Federal University of Pernambuco, as a partial requirement to obtain the degree of Master at Computer Science.

Concentration Area: Software Engineering and Programming Languages

Advisor: Leopoldo Motta Teixeira

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

M827w Moraes, Pedro Henrique Sousa de
Willow: a tool for interactive data structures and algorithms visualization / Pedro Henrique Sousa de Moraes. – 2020.
113 f.: il., fig., tab.

Orientador: Leopoldo Motta Teixeira.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências e apêndices.

1. Engenharia de software. 2. Linguagem de programação. I. Teixeira, Leopoldo Motta (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 207

Pedro Henrique Sousa de Moraes

"Willow: A Tool for Interactive Data Structures and Algorithms Visualization"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 29/10/2020.

BANCA EXAMINADORA

ACKNOWLEDGEMENTS

This work was accomplished with the help of several people. I would like to express my deeply appreciation to my advisor Leopoldo Motta Teixeira, who instructed me along my trajectory and provided me with encouragement and patience throughout the duration of this project. I would also like to extend my gratitude to the professor Marcelo d'Amorim, whose help in several stages of the research with practical suggestions cannot be overestimated. I gratefully acknowledge the assistance of professor Waldemar Neto, who provided a good amount of assistance during the development of the research.

All the gratitude to my parents (Genilda Teófilo and Pedro Saraiva), for all the support and always ensuring that I had everything to complete this stage of life.

We acknowledge support from FACEPE (IBPG-0751-1.03/18 and APQ-0570-1.03/14), and CNPq (409335/2016-9). This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

ABSTRACT

Teaching Introductory Programming and Data Structures and Algorithms is an important part of Information Technology courses. Both disciplines include essential concepts for software development. Preparing lessons for these courses can be time demanding and tedious as instructors often need to create and modify examples using slides and sketches on a board. Students may also have difficulties due to the high level of abstraction of the content taught in both courses. Educational visualization tools, such as Python Tutor exist, but they provide rigid choices of visualization schemes used to represent the data. Most educational tools are discontinued or have limited support to the visualization of data structures and algorithms. Other tools create visualizations of several algorithms, but lack the ability to edit the source code or inputs. This work proposes WILLOW, a web-based interactive tool to visualize program state. WILLOW enables the user to customize visualizations and to walk through the code in both directions to facilitate code understanding. The sensible features of WILLOW are its ability to change data representations, jump to any point of a program with visual support during debug sessions, and detection and animation of common data structures such as lists and trees. To evaluate WILLOW, we conducted two studies, a survey with instructors of several universities, and a follow up experiment with programmers of a freelancing platform. We obtained positive feedback from 91% of the survey participants, suggesting that WILLOW can be used as an teaching aid tool by instructors. In the follow up experiment with programmers, we could not find significant difference between participants that used WILLOW and participants that did not, the results of the experiment were not conclusive. Nevertheless, we obtained positive results after considering a subset of the experiment tasks, participants also reacted positively to the tool and many would like to use it again.

Keywords: Program Visualization. Algorithm Visualization. Educational Tool.

RESUMO

O ensino de Introdução a Programação, e Algoritmos e Estruturas de Dados é parte importante da formação de alunos em cursos de computação. Ambas as disciplinas incluem conceitos essenciais para o desenvolvimento de software. No entanto, preparar as aulas para esses cursos pode ser demorado e tedioso, pois os professores geralmente precisam criar ou modificar exemplos de algoritmos executando passo a passo, usando apresentações de slides ou esboços em um quadro. Os alunos também podem ter dificuldades, devido ao alto nível de abstração do conteúdo ministrado em ambos os cursos. Existem ferramentas de visualização educacionais, como o Python Tutor, mas essas ferramentas fornecem visualizações rígidas de esquemas usados para representar os dados. Várias ferramentas educacionais foram descontinuadas ou tem suporte limitado à visualização de estruturas de dados e algoritmos. Outras ferramentas criam visualizações de vários algoritmos, mas não têm a capacidade de editar o código-fonte ou entradas. Este trabalho propõe WILLOW, uma ferramenta interativa baseada em tecnologias web para visualizar o estado de programas. WILLOW permite que o usuário personalize visualizações e navegue pelo código em ambas as direções para facilitar a sua compreensão. As principais características de WILLOW são sua capacidade de alterar representações de dados, saltar para qualquer ponto de um programa com suporte visual durante as sessões de depuração, e detecção e animação de estruturas de dados comuns, como listas e árvores. Para avaliação de WILLOW, realizamos dois estudos, um survey com professores de várias universidades, seguido de um experimento com programadores de uma plataforma de freelancers para resolução de problemas com e sem WILLOW. Obtivemos feedback positivo de 91% dos participantes do survey, que sugere que WILLOW pode ser usado como uma ferramenta de auxílio no ensino pelos professores. Com relação ao estudo com programadores, não foi encontrada diferença significativa nas respostas entre participantes que usaram WILLOW e participantes que não usaram. Contudo, foram obtidos resultados positivos ao considerar um subconjunto das tarefas do experimento, participantes também reagiram positivamente à ferramenta e muitos gostariam de usa-la novamente.

Palavras-chaves: Visualização de Programas. Visualização de Algoritmos. Ferramenta Educacional.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Introductory Programming (IP) and Data Structures and Algorithms (DSA) are important courses in the Computer Science curriculum. Teaching and Learning these courses can be challenging for instructors and students. Abstract concepts and programming languages can be overwhelming and lead students to misconceptions and difficulties to understand the concepts. To help students, instructors typically prepare examples on lecture slides or create sketches on the board to illustrate algorithm behavior (GUO, 2013).

This task can be daunting for instructors and ineffective for students. Lecture slides require a great deal of planning and preparation from the instructors and whiteboard sketches take time to draw during class, and the diagrams can get confusing (ORSEGA; VANDER ZANDEN; SKINNER, 2012). Moreover, both approaches require the instructor to create new examples every time new inputs and modifications are proposed.

Another option to illustrate how DSAs work is through the use of graphical tools. Some of these tools, (e.g., Python Tutor) were already adopted in introductory CS courses in Universities in the US and Canada such as Berkeley and Toronto with good acceptance of instructors, who indicated the tool helped clarify basic programming concepts (GUO, 2013). Unfortunately, the currently available graphical tools generate "rigid" graphical representations—they adopt the same visual elements to represent different types of data structures and do not preserve the positioning of objects. Hence, many of the algorithms usually seen through a DSA course can not be consistently represented (SORVA et al., 2012; FOUH; AKBAR; SHAFFER, 2012). Python Tutor, for example, can not create good visualizations for data structures more complex than linked lists, due to its restrictions in the positioning of objects.

To overcome these limitations we propose Willow, a tool for generating interactive examples of algorithms, data structures and other basic programming concepts. Willow allows users to create visualizations for programs written in Python or Java, with few restrictions. In order to create better representations of algorithms, Willow also allows users to change how objects are represented, by doing so, users can achieve better quality examples which favor understanding.

Willow's main focus is to provide a tool that can help instructors and students through the visualization of more complex DSAs, since the best current available tools such as Python Tutor (and derivatives) (GUO, 2013) and Kanon (OKA; MASUHARA; AOTANI, 2018) are not capable of effectively showing a good variety of algorithms.

To be able to create such visualizations, Willow allows users to manipulate a base visualization, changing how objects are represented, creating animations for data structures, and navigating to any point of a program.

1.1 MOTIVATION

In the past decades, several tools had been developed with the purpose of visualizing generic programs, focusing mainly in didactic visualizations to teach students of introductory courses. However, very few educational tools targeted more complex algorithms commonly taught in DSA (SORVA; KARAVIRTA; MALMI, 2013; VELÁZQUEZ-ITURBIDE, 2019). Another factor that motivated the creation of WILLOW is that almost all educational tools are discontinued for several reasons, such as the use of old technologies, closed source development and abandonment.

WILLOW was created to be flexible and extensible, being able to support multiple languages (an extremely rare feature among the older tools) and create visualizations for any source code the user provides. WILLOW's implementation also uses the most recent web technologies and is open-source, its repository is available at: `https://github.com/pedro00dk/willow`.

1.2 RESEARCH

To evaluate WILLOW, we carried two studies, a survey with IP and DSA instructors of several universities and an experiment with amateur programmers from the freelancing platform Upwork (UPWORK, 2020). Our studies aimed at answering the following research questions:

- RQ1: What are the most common practices used by instructors when teaching coding-related courses?
- RQ2: What are the perceptions of these instructors towards WILLOW?
- RQ3: What are the benefits WILLOW can offer to programmers when solving data structures and algorithms problems, with regard to time to solve, confidence and correctness?

In the following chapters we discuss the background of our research (Chapter 2) and some of the most recent related work (Chapter 3). We then describe our proposed tool (Chapter 4). We detail all the evaluation process and results of both studies applied in the evaluation of WILLOW(Chapter 5). Finally, we conclude this dissertation (Chapter 6).

2 BACKGROUND

This chapter introduces an overview of the difficulties that Information Technology students suffer in initial courses, such as Introductory Programming, and Data Structures and Algorithms. These difficulties are categorized, detailed and associated with the concept of notional machines. Notional machines are abstractions of how students comprehend code execution dynamics. Educational visualization tools provide support for students by showing correct models of notional machines, making abstractions more clear and, hence, helping them to better understand the programming concepts.

2.1 THE LEARNING CHALLENGE

The challenges that Information Technology students have to overcome are presented since the beginning, when they have to attend to their potentially first Introductory Programming courses. Introductory Programming courses can be very challenging to novices. The primary goal is to make students learn how to create their own programs, and solve simple problems using some programming language. This is a demanding task for students, and is often not achieved. As a result of that, courses fail to teach programming worldwide, exhibiting high failure rates (SORVA et al., 2012).

One of the main causes of failures are misconceptions and other difficulties exhibited by novices throughout the courses, drastically reducing students ability to learn and make progress (QIAN; LEHMAN, J., 2017). Still, misconceptions are only one of many possible causes that may lead to students difficulties in learning. There are several other reasons that may impact the students learning process, such as social, personal or institutional problems (SILVA RIBEIRO; BRANDÃO; BRANDÃO, 2012),

2.1.1 Misconceptions

The early studies in the area of misconceptions in Computer Science and programming arose in 1980s (BAYMAN; MAYER, 1983; DU BOULAY, 1986). Many studies emerged from researchers in the Computer Science education area, analysing students imprecise or incomplete understandings of programming concepts. Since there is no commonly used definition to students difficulties, studies use misconception or similar terms, such as "difficulties", "errors", "mistakes" and so on (QIAN; LEHMAN, J., 2017).

Misconceptions are deficient or inadequate understandings for many practical contexts. In programming, misconceptions includes syntax errors, confusions with control flow primitives, wrong interpretation of concepts, difficulties in using learned constructs, planning and debugging problems and others (QIAN; LEHMAN, J., 2017; SORVA; KAR-

AVIRTA; MALMI, 2013; JACKSON; COBB; CARVER, 2005).

Misconceptions with Syntax

Novices in Introductory Programming courses often exhibit syntactic errors in their programming activities. A study on large volumes of student source code in the Java language (ALTADMRI; BROWN, 2015), showed that the most frequent errors presented by students are mismatches of parenthesis, brackets and quotation marks. Other common Java syntactic errors committed by novices are missing semicolons, unresolved symbols and illegal expressions (JACKSON; COBB; CARVER, 2005). Semicolon is a character used to terminate statements in Java and other programming languages, and it is often forgotten by novices. The unresolved symbol error is frequently the result of accessing the value of a undeclared variable. And illegal expression errors are often caused by malformed boolean expressions due to unfamiliarity with operators. For example, often novices mistakenly use the assignment operator (=) as a comparison operator (==), causing syntactic errors and in some cases semantic errors, which are harder to be fixed by novices (ALTADMRI; BROWN, 2015; SORVA et al., 2012).

Syntactic errors are among of the most frequent mistakes students make (JACKSON; COBB; CARVER, 2005; ALTADMRI; BROWN, 2015). However, they can be easily fixed, because most modern integrated development environments (IDE) and language compilers can easily spot syntax problems and present the user with error messages about the problems and sometimes hints for correction (QIAN; LEHMAN, J., 2017).

Misconceptions with Semantics

Most of the errors students make are related to syntax of programming languages (JACKSON; COBB; CARVER, 2005; GARNER; HADEN; ROBINS, 2005), they are the most frequent type, but also superficial, and easy to fix. However, students misconceptions with program semantics have a much greater impact on their mental models of a program runtime. These misconceptions prevent the student to correctly express their solutions for problems, because they cannot understand the behavior of the programs they make (BAYMAN; MAYER, 1983; SORVA et al., 2012).

There are many sources of misconceptions in the semantic context of programming languages, such as variables, for instance, variables are one of the fundamental building blocks for any program, used to store inputs and outputs of program operations. Many different types of misconceptions involving variables were reported by studies. Students may not understand that variables only store a single value, causing then to try assigning and retrieving multiple values at the same time for a single variable (DOUKAKIS;

GRIGORIADOU; TSAGANOU, 2007). Novices can also misunderstand how both the statement order and the expression order (e.g. `A=B; B=A;`) influence in the result of the assignment (DU BOULAY, 1986). Another source of confusion around variables is the meaning of their names, which can cause novices to misinterpret the value of a variable. Variable names, even being arbitrary, may cause students to think the value contained in a variable is always the meaning of the variable name (KACZMARCZYK et al., 2010; QIAN; LEHMAN, J., 2017).

Other difficult concepts for novices that cause many misconceptions, are control flow primitives (e.g. `if-else, while, for`). For instance, students may believe that both `if` and `else` conditional blocks are executed, or that if the condition for a conditional statement evaluates to `false`, the program execution stops. Loop constructs are also a challenge to novices, since they still did not develop a concise understanding of variable scopes, the use of loop structures becomes too confusing. Novices may not know which lines of the loop scope are executed, how many times the loop executes, initialization, stop and increment statements, and so forth (QIAN; LEHMAN, J., 2017). Another misconception involving loop statements is about how the loop code executes. Some students fail to understand that loop code executes sequentially and that information can be propagated to the next loop executions (DU BOULAY, 1986). In programming languages with multiple loop constructs such as Java, many novices avoid using some of these constructs, as they believe that one construct is better than the others. This is a result of the lack of understanding of the benefits of each loop construct and how to use them to help solving different problems (QIAN; LEHMAN, J., 2017).

Since the increase in popularity of the object-oriented paradigm (OOP) in the 1990s, studies conducted with novices reported that students often struggle with many OOP principles (GUZDIAL, 1995; RAGONIS; BEN-ARI, 2005; SORVA et al., 2012). Concepts such as classes and objects are among the most confusing in OOP, novices often misunderstand what these elements are for, and how they relate to one another (KACZMARCZYK et al., 2010). Methods and functions are other sources of misconceptions. These concepts introduce new ways to interpret already seen concepts, such as variable scopes, making it even more confusing when mixed with classes and instances attributes. Some students may exhibit difficulties in understanding where the function parameters come from, as well as side effects that may or may not happen on parameters values, how the return statement works and where the return value goes (RAGONIS; BEN-ARI, 2005). Novices may not even understand the role of the main method, or the relationship between methods, objects and classes (SAJANIEMI; KUITTINEN; TIKANSALO, 2008). When objects are introduced, novices have to deal with references, a new type of variable value. Although in most languages the assignment semantics of a reference value is identical to the semantics of primitive values, novices may not distinguish between references and objects, causing them to misunderstand the result of copying references (KACZMAR-

CZYK et al., 2010). This misinterpretation may cause students to build different mental models about reference assignments (QIAN; LEHMAN, J., 2017).

Besides misconceptions with general object-oriented programming concepts, novices may also exhibit difficulties in developing decentralized solutions, required for OOP programming (GUZDIAL, 1995), and correct construction of a object oriented mental model and notional machines (SORVA et al., 2012).

Misconceptions with Strategies

Strategies refer to knowledge in programming activities such as planning, writing, testing and debugging programs. Different terms were adopted to describe programmer's strategic knowledge, such as plans, patterns, schemas and others (EBRAHIMI, 1994; LOPEZ et al., 2008; QIAN; LEHMAN, J., 2017).

The first barrier novices have to overcome to develop good strategy knowledge is a correct understanding of programming language syntax and semantics. Studies report that students misconceptions in strategic knowledge are highly correlated with difficulties in syntactic and semantic knowledge (EBRAHIMI, 1994; LOPEZ et al., 2008). Since novices only have small programming knowledge, their lack in strategies and patterns to solve programming problems. This lack of strategies is associated to their capacity to interpret the problem objectives and decompose the problem, which influences in the planning, testing and debugging (MULLER, 2005).

Most novices in Introductory Programming are capable of creating programs that "work". However their program often will not check program invariants, conditions and other edge cases which can cause the program to fail at runtime (SAJANIEMI; KUITTINEN, 2005). In some cases, students may not know how to check the correctness of their programs, and believe that they can obtain a partially correct result from the program if part of the code they wrote is correct.

Another problem is the use of debugging tools. Novices are unfamiliar with the information provided by debuggers. By combining the unfamiliarity with debuggers and semantic elements of the program, the result is a poor and local analysis of the program runtime behaviour (QIAN; LEHMAN, J., 2017). After all, most of the problems novices confront in debugging tasks are not fixing the program errors, but rather understanding the program behaviour and finding the error (MCCAULEY et al., 2008). After identifying and locating the errors in the program, novices can fix most of them (FITZGERALD et al., 2008).

Misconceptions on Data Structures and Algorithms

Not only students of Introductory programming suffer from misconceptions of programming topics. Some studies reported misconceptions with topics of Data Structures and Algorithms courses (DANIELSIEK; PAUL; VAHRENHOLD, 2012; PAUL; VAHRENHOLD, 2013; KARPIERZ; WOLFMAN, 2014; ZEHRA et al., 2018; ZINGARO et al., 2018; VELÁZQUEZ-ITURBIDE, 2019). These studies showed that students of Data Structure and Algorithms (DSA) courses commonly exhibit misunderstandings of the subjects studied, some students also show misunderstandings on Introductory Programming topics.

One study reported that students exhibited confusion in understanding the differences between heaps and binary search trees (BST). This confusion was caused because students developed restricted mental models of heaps, where the data structure representation must be similar to a tree. Another misconception that contributed to the confusion was the unawareness of the left-completeness property of heaps (PAUL; VAHRENHOLD, 2013). Another study reported source of confusion involving binary search trees was that students struggled with the possibility of inserting duplicate keys (keys already present in the data structure) (KARPIERZ; WOLFMAN, 2014). Students also exhibited misconceptions in greedy algorithms, they did not understand the design decisions that lead to efficient implementations, which led to bad greedy algorithm implementations (VELÁZQUEZ-ITURBIDE, 2019).

2.2 FACTORS CONTRIBUTING TO MISCONCEPTIONS

Many factors can contribute to novices misconceptions and other difficulties. Previous research reported previous math knowledge, understanding of the English language, task complexity, instructor knowledge and teaching methods, and more, as factors that may contribute to novices misconceptions (ROBINS; ROUNTREE; ROUNTREE, 2003; URQUIZA-FUENTES; VELÁZQUEZ-ITURBIDE, 2009).

Prior math knowledge is a great source of misconceptions among novices. It is specially true for students that had a deficient basic formation in the area of exact sciences (SORVA; LÖNNBERG; MALMI, 2013). Due to the lack of math knowledge, students may exhibit difficulties when abstracting information or to interpret syntax and semantics of programming languages. Difficulties related to algebraic expressions are common, and students often forget to declare variables before using it, because it is not necessary in high school (JACKSON; COBB; CARVER, 2005). Other misconceptions caused by math knowledge is about integer and floating point variables and numeric precision. Students may believe that variables are capable of holding numbers of any precision, which may cause confusion when applying operators with different data types or due to unexpected floating point operation results due to limited precision (DOUKAKIS; GRIGORIADOU; TSAGANOU, 2007).

Many programming language constructs are based on natural languages, specially in the English language. A study with Chinese high school students reported that the ability with English was the best predictor of students success rates (QIAN; LEHMAN, J. D., 2016).

Task complexity is a factor that affects novices' cognitive load, causing confusion mainly among early beginners that are still unfamiliar with programming language keywords and syntax. Beginners may forget the most basic programming constructs such as parenthesis, operators or semicolons when solving problems. A study reported that students submissions for the first Introductory Programming activities were mostly flawless, but when activities become increasingly challenging, students started to present more syntactic errors (ANDERSON; JEFFRIES, 1985). Students may also suffer when debugging code, tracing skills require high demand on concentration, and due to limited knowledge novices often use wrong variable values and miss errors when debugging (VAINIO; SAJANIEMI, 2007).

Occasionally, teaching methods adopted by instructors may contribute to students' misunderstandings. Instructors may use inadequate analogies or metaphors. A common example in Introductory Programming is to describe a variable as a box, students may believe that a variable may hold more than one value because boxes can hold many objects (CLANCY, 2004). The use of analogies can contribute to students' understanding, especially for complex concepts, but their inadequate use may create barriers for novices, preventing then from building correct knowledge and progress in learning.

2.3 MENTAL MODELS

Mental models are interpretations of the thinking processes about the behavior of anything in the real world (RAMALINGAM; LABELLE; WIEDENBECK, 2004). These interpretations shape how a person understands the relationships, opportunities and consequences of his or her actions. A mental model can be seen as descriptions of the processes of how something works, since it is impossible to remember all details. These models allow us to simplify complexity and evaluate which things are more relevant, defining how we reason. Mental models play a major role in the learning process, particularly in many activities related to problem solving, such as cognition, reasoning, decision-making and event anticipation (GÖTSCHI; SANDERS; GALPIN, 2003).

2.3.1 Notional Machines

In the Computer Science context, a notional machine is a mental model for how a person understands the the behaviour of programming constructs. Programmers think of the notional machine as a conceptual computer, whose operations and properties are implied by their knowledge of the programming language being used (SORVA et al., 2012).

Students in Introductory Programming start developing their own notional machines since the beginning of the course. However, novices have fragile knowledge in programming languages syntax and semantics, which tends to affect their notional machines, hence, developing misconceptions (LOPEZ et al., 2008).

2.3.2 Visualizing Notional Machines

To help students comprehend programming concepts, instructors often use visualizations of some sort in their classes (NAPS; RÖSSLING, et al., 2002). With the help of visualizations, instructors can show details of program concepts, bringing to the surface the program runtime, which was hidden from the students focusing on the program only at the code level. Visualization can be of any sort, and depending on the instructor objectives, different levels of abstractions may be used to represent a program runtime. Usually, these representations are visualizations of instructors' own notional machines, and serve as conceptual models to help students build their programming knowledge (SORVA; KARAVIRTA; MALMI, 2013).

A common strategy adopted by instructors to show visualizations is the chalk-and-talk approach (BECKER; WATTS, 2001), where instructors draw sketches of visualizations on the blackboard. Another alternative to only drawing sketches, is involving novices in this activity. Some studies experimented with novices participation by making them draw their own perceptions of program runtimes (HERTZ; JUMP, 2013; HOLLIDAY; LUGINBUHL, 2004). However, drawing sketches may take a long time, which limits the amount of examples a instructor can show in a lecture (SORVA; KARAVIRTA; MALMI, 2013). The drawing can also get messy and possibly confuse novices (GUO, 2013). It is also common for instructors to use lecture slides containing pictures and diagrams to illustrate programming concepts, data structures or algorithms. Still, the creation of presentation material of good quality for lectures requires a long time of planning and preparation (ORSEGA; VANDER ZANDEN; SKINNER, 2012).

2.4 EDUCATIONAL PROGRAM VISUALIZATION

To assist novices and instructors in providing examples of programming concepts, several tools that aim to make visualization more practical were developed since the 1980s. The idea behind most of these tools is to show a program or algorithm runtime steps. The level of abstraction may vary with the tool, as well as if the steps are displayed automatically or require some interaction (SORVA; KARAVIRTA; MALMI, 2013). With the help of these tools, instructors can demonstrate programming concepts and algorithms, and students can analyse behavior.

There are some studies that define different taxonomies to classify software visualization (SV) tools (NAPS; COOPER, et al., 2003; HUNDHAUSEN; DOUGLAS; STASKO, 2002; MALETIC; MARCUS; COLLARD, 2002; KELLEHER; PAUSCH, 2005). Figure 1 shows a diagram of how Software Visualization tools for education are organized, the diagram shows a common taxonomy used by review studies to categorise software visualization tools by form, and contain two main categories, which are *Algorithm Visualization* (AV) and *Program Visualization* (PV) (PRICE; BAECKER; SMALL, 1993). Algorithm Visualization tools show highly specialized visualizations for a limited set of algorithm it supports, the algorithms cannot be modified. Program Visualization tools require users to implement their own code, the purpose of visualization in these tools vary, some tools abstract the code representation, other create visualizations of the program runtime dynamics.

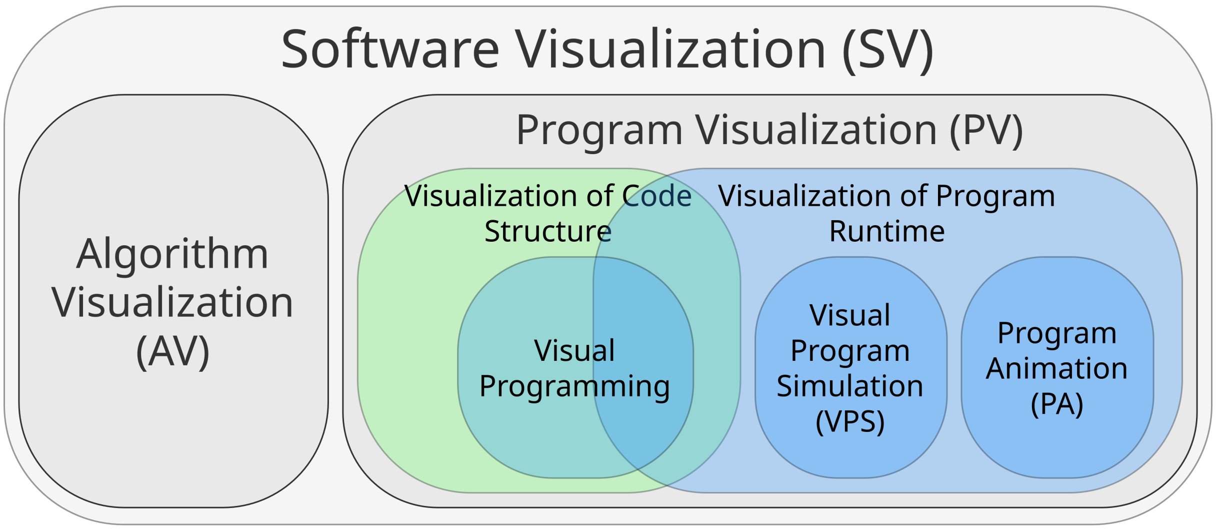

Software Visualization (SV)
Algorithm Visualization (AV)
Program Visualization (PV)
Visualization of Code Structure
Visualization of Program Runtime
Visual Programming
Visual Program Simulation (VPS)
Program Animation (PA)

Figure 1 – Software Visualization Categories.

The Software Visualization term (SV) refers only to educational tools. Diagram adapted from (SORVA; KARAVIRTA; MALMI, 2013). The area of the categories do not matter.

2.4.1 Algorithm Visualization

Algorithm Visualization tools are by far the most common kind of tool studied in the field of educational program visualization (SORVA; LÖNNBERG; MALMI, 2013). The Algorithm Visualization category includes tools that automatically execute predefined programs. The provided programs cannot be changed, and for many tools, users cannot even change program inputs. The tools in this category provide very low control of their visual elements. Users usually only have access to the program execution controls. Another property of AV tools is their high level of abstraction. This makes AV tools less interesting for novices, because it makes learning fundamentals of program runtime more difficult (SORVA; KARAVIRTA; MALMI, 2013).

Since AV tools only provide a single or a small set of programs, they generate visualizations with detailed elements, which can be useful for instructors to use in demonstrations. Algorithm Visualizations can still be used by more advanced students alone, but it requires sufficient attention to the visualizations. However, simply viewing generated visualizations may not be enough to help the student understand the content (SORVA; LÖNNBERG; MALMI, 2013).

AV tools are an interesting approach to demonstrate data structures and algorithms from a visual and abstract point of view. However, some studies propose that having students interacting with visualizations rather than only viewing is more beneficial from a learning point of view (HUNDHAUSEN; DOUGLAS; STASKO, 2002; NAPS; COOPER, et al., 2003). This shifted the trend of educational program visualization from Algorithm Visualization to Program Visualization (SORVA; LÖNNBERG; MALMI, 2013). Examples of AV tools are provided in the Chapter 3.

2.4.2 Program Visualization

The Program Visualization (PV) category includes tools that allow the user to manipulate program source code. PV tools are further divided into two main groups showed in Figure 1.

Some tools use visual abstractions of program source code structures, which allow users to create programs using simplified visual components that represent code elements such as variables, operators, flow control flow primitives and so forth.

Other tools focus on providing visualizations for program runtime dynamics. These tools work like debugger applications, the code provided is analysed while in execution, the collected information is used to generate visualizations which are displayed to the user (SORVA; KARAVIRTA; MALMI, 2013). However, this flexibility comes with a cost. Generated visualizations are good for explaining simple programming concepts, but due to tool limitations, they lack detail to explain more complex data structures and algorithms.

Examples of PV tools are provided in the Chapter 3.

Visual Programming

Within the program visualization category, Visual Programming tools attempt to provide new ways to create programs by dropping common text based formats in favor of using graphic components. One of the Visual Programming most adopted strategies is the use of programming blocks (ROQUE, 2007; WEINTROP; WILENSKY, 2015; BAU et al., 2017). Another strategy adopted by tools is the use of programming nodes for data processing pipelines, which is common among rendering and game development software. [1]

Due to their simplicity, Visual Programming tools were being used for a long time in other environments besides education, such as multimedia creation, data analysis and other fields (BRESSON; AGON; ASSAYAG, 2011; LAURSON; KUUSKANKARE; NORILO, 2009; YOUNG; ARGIRO; KUBICA, 1995; TAKATSUKA; GAHEGAN, 2002).

Visual Program Simulation

In Visual Program Simulation (VPS), the learner takes the role of the computer, he or she is responsible for the program execution. This is possible in VPS tools because the visualization components are interactive. These components are controlled by the learner to declare variables, to evaluate expressions, to make modifications in the program state, and to control the program flow. By using these graphical controls, VPS tool users can guide the program runtime execution step by step (SORVA et al., 2012). Therefore, VPS tools only provide visual support as the user reads and advances through the source code.

The strategy adopted by VPS tools is beneficial for novices. Solving problems using VPS tools challenges novices by making them think like a computer. Such activities help them to understand the dynamics of program runtimes, one of the main difficulties students have in Introductory Programming courses (SORVA; LÖNNBERG; MALMI, 2013).

However, VPS tools are not well suited for more advanced students. From the moment the student has a better consolidated knowledge about program runtime dynamics, having to control program execution step by step becomes bothersome and tedious, reducing students cognitive engagement.

[1] `https://www.blender.org` `https://unity.com` `https://www.unrealengine.com`

Program Animation

Similar to Algorithm Visualization, tools in the Programming Animation (PA) category display visualizations of program runtime elements.

Programming Animation tools can create visualizations from any source code provided by users. However, the generated visualizations usually have lower levels of abstractions, these visualization often contain all declared variables and their values, allocated objects and information of their attributes, and so on. Navigation through the program runtime is often manually controlled by the user and with step (a source code line) resolution (SORVA; KARAVIRTA; MALMI, 2013).

The low level of abstraction makes visualizations harder to understand by early novices. On the other hand, PA tools have debugging capabilities similar to typical tree-view debuggers, and a study reported that PA tools can be used even by professional developers for some activities such as data structure development (OKA; MASUHARA; AOTANI, 2018).

Table 1 – Differences between categories of Software Visualization tools.

Tool type	Source code	Program inputs	State visualization	Code execution
Algorithm Visualization	Provided by the tool. May contain support for only one or multiple algorithms.	May be provided by the tool or the user.	High level visualization, with specific detail for individual algorithms.	Controlled by the tool, but some tools allow the user to control execution.
Visual Programming	Provided by the user, but assisted by the tool. The user programs with visual structures rather than plain text.	Provided by the user.	No direct state visualization.	The execution is uninterrupted as a normal program.

Visual program Simulation	Provided by the user.	Provided by the user.	Low level visualization, showing all the data allocated in the stack and memory.	The user acts as the computer, executing every step of the program. The user has to interact with the state visualization interface to inform what computation is happening in the program.
Program Animation	Provided by the user.	Provided by the user.	Low level visualization, showing all the data allocated in the stack and memory.	Controlled by the user.

3 RELATED WORK

This chapter presents a review of visualization tools intended to use in Introductory Programming or Data Structures and Algorithms courses, and small descriptions on how these tools were evaluated. All tools described in this section were prominent tools created within the last decade, nevertheless, some of them are already discontinued.

3.1 VISUALGO

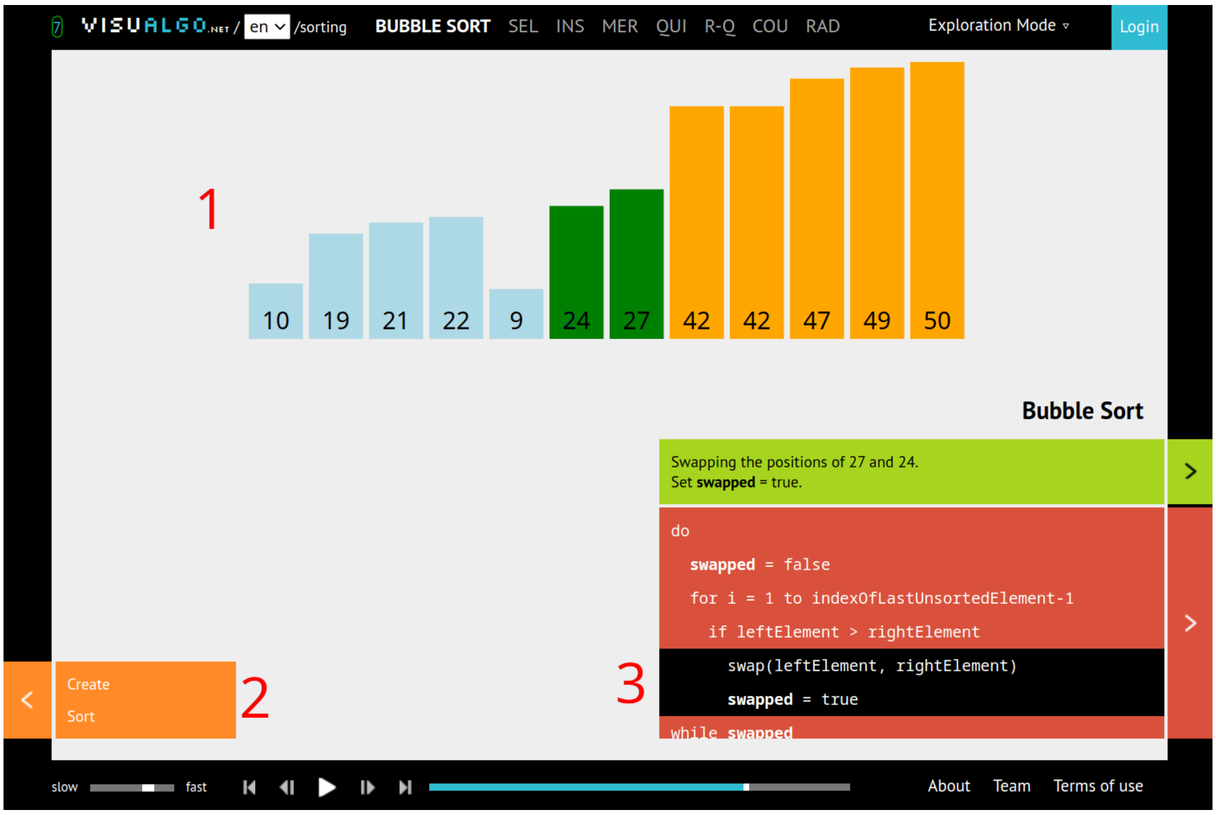
VisuAlgo is a web based tool that provides visualizations for dozens of algorithms commonly studied in Data Structures and Algorithms courses (DIXIT; YALAGI, 2017). [1]

Since VisuAlgo is an Algorithm Visualization tool, it provides predefined visualizations, not allowing users to create their own programs. Although users cannot create their own programs, it is possible to change algorithm inputs in some of the visualizations, allowing users to analyse the behaviour of the algorithms in different cases. The tool provides a large collection of visualizations, for sorting algorithms, lists, trees, graphs, and so on. Figure 2 shows the execution of a sorting algorithm and some of the visualization details.

The authors conducted an experiment with 78 students of Design and Analysis of Algorithms. The participants were divided into 4 groups and had to answer exercises about the quicksort algorithm, two of the four groups were allowed to access Visualgo through its website (VISUALGO,. . . , n.d.). The study found that the best students had good results no matter in which of the groups they were. For the other participants, the ones from the group that was allowed to access Visual exhibited better scores.

Besides VisuAlgo, there are many others Algorithm Visualization tools such as OpenDSA [2], AlgoVIZ (ROMANOWSKA et al., 2018), IScketchMate (ORSEGA; VANDER ZANDEN; SKINNER, 2012), DAVE (VRACHNOS; JIMOYIANNIS, 2014), and more. All of these tools provide very similar visualizations for the available algorithms and data structures. On the other hand, Program Visualization tools offer more diverse visualizations and ways of interaction.

[1] https://visualgo.net
[2] https://opendsa-server.cs.vt.edu



Swapping the positions of 27 and 24.
Set **swapped** = true.

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

Figure 2 – Simulation of a sorting algorithm in VisuAlgo.

The image shows an array (1) being ordered, the operations that the user can run are listed in the orange box (2). On the right size (3), a pseudo-code that accompany the program execution is provided, also describing the operations being executed.

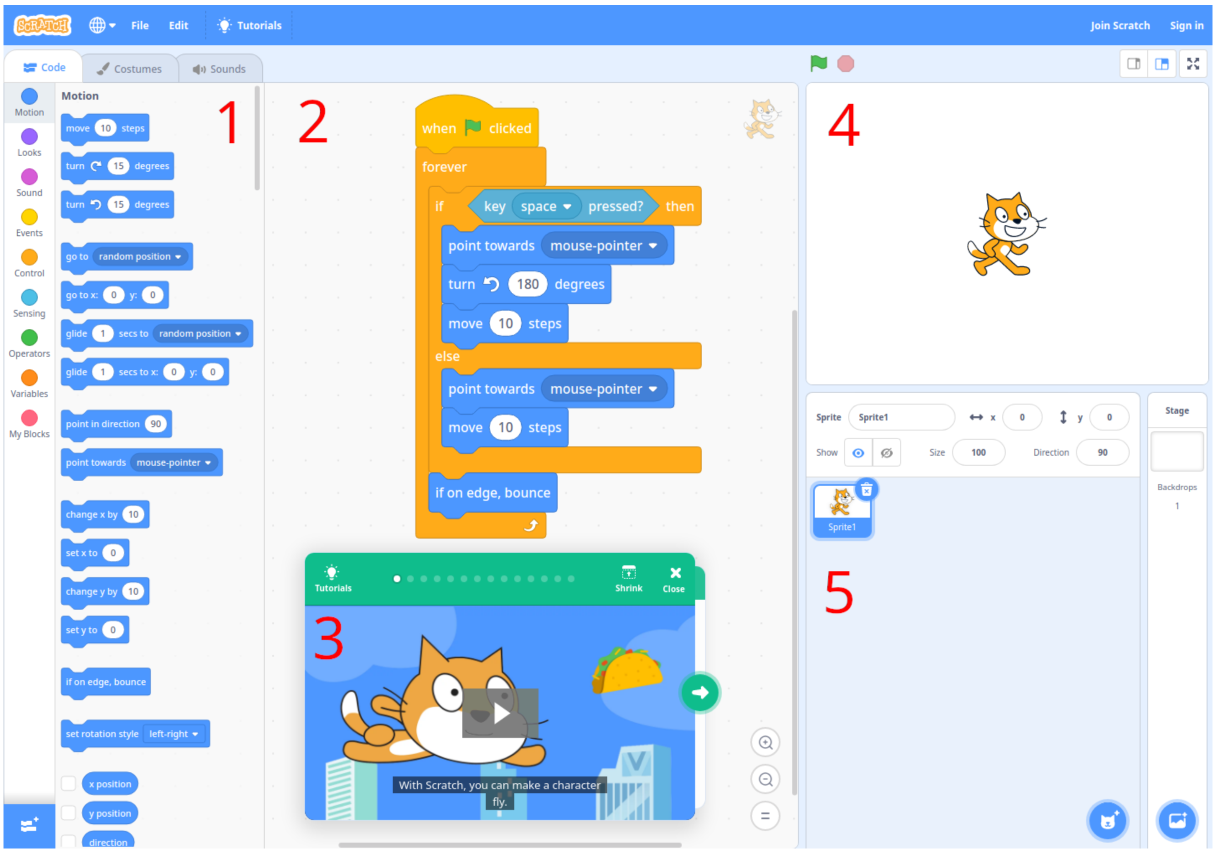
3.2 SCRATCH

Scratch[3] is a web based programming environment for novices (MALONEY et al., 2010). Scratch works by providing a visual interface for program construction. Users can create programs by using blocks of several types, simple blocks that can be used to declare variables, control the program flow, use operators, and more complex blocks that read user input, play sounds, interact with sprites and so forth (BAU et al., 2017). Therefore, Scratch can be categorized as a Visual Programming tool.

Figure 3 shows Scratch interface and the blocks used to build a program. Users create programs by combining several types of logic blocks, which can be used to read inputs, play sounds and control sprites in a canvas.

Visual Programming tools fulfill the role of creating abstractions of programming languages syntax, reducing novices cognitive load. A study with 90 students reported that participants found it was easier to create programs using block-based environments rather than text-based (WEINTROP; WILENSKY, 2015). On the other hand, the number of syntax constructions mapped by the provided blocks are limited, not allowing the use of

[3] `https://scratch.mit.edu`



```
when ⚑ clicked
forever
    if key space pressed? then
        point towards mouse-pointer
        turn ↺ 180 degrees
        move 10 steps
    else
        point towards mouse-pointer
        move 10 steps
    if on edge, bounce
```

Figure 3 – Scratch editor.

Example of a program built using scratch, programs are built using a (1) list of blocks for flow control, variable declarations, operators, read user input and interacting with sprites. Users can drag blocks to the program pane (2) and create any logic they want. Scratch offer many video tutorials (3) in how to create several programs. After build the program, users can execute the program, and sprites (5) will act in the output pane (4) according to the program logic.

more complex programming constructions such as objects, pointers, arrays, classes and more. These concepts are fundamental for Introductory Programming students, as they are needed to express more complex programs.

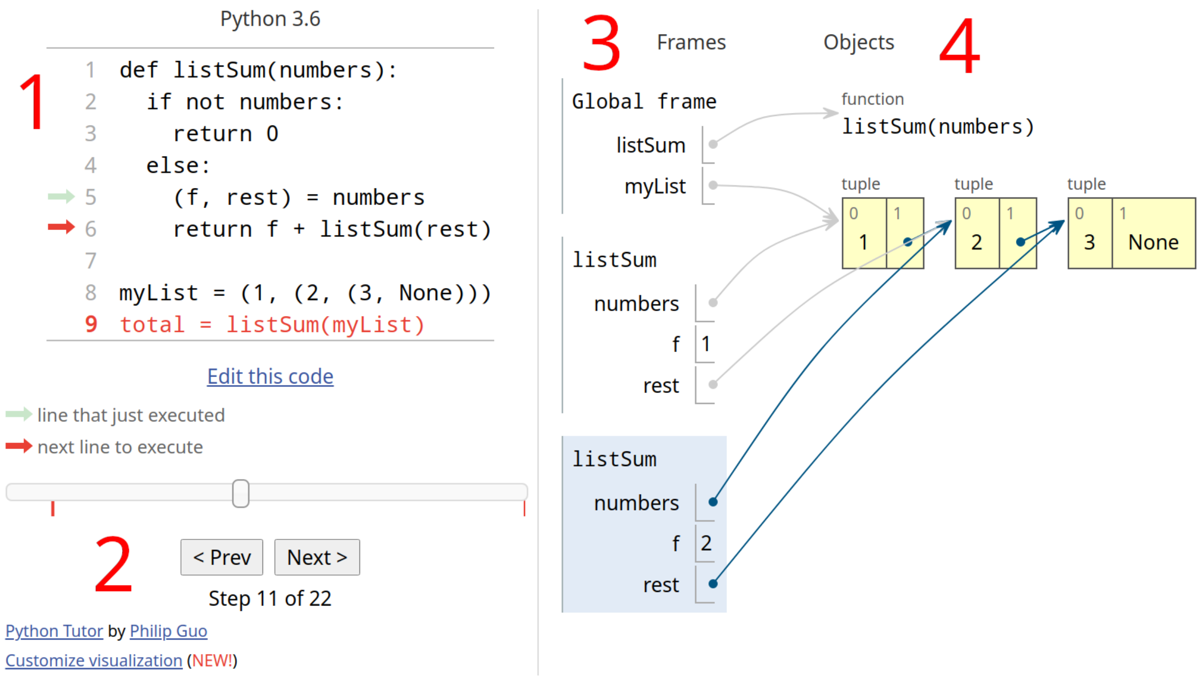
3.3 PYTHON TUTOR

Python Tutor[4] is currently one of the most well-known visualization tools in the Program Animation category. (GUO, 2013). Since its release, Python Tutor has been adopted by some universities in their Introductory Programming courses, such as UC Berkeley[5], MIT, University of Washington and University of Waterloo (GUO, 2013). Some of the major elements that contributed to the high popularity of this tool were the fact that it was developed for the web environment, being easily accessible by students and professors,

[4] http://pythontutor.com
[5] https://cs61a.org/

and the growth in popularity of the Python language among Introductory Programming courses. Before Python Tutor, most Program Visualization tools only supported languages such as Java and C/C++.

Python Tutor was first released in January 2010, motivated by the experience of its creator with the Python language to novices, by drawing messy diagrams on the board. Currently, the tool still receives updates (PYTHON. . . , n.d.). The main goal was to create a tool that professors and students prefer to use in addition to traditional strategies such as sketches and lecture slides (GUO, 2013).



```
def listSum(numbers):
  if not numbers:
    return 0
  else:
    (f, rest) = numbers
    return f + listSum(rest)

myList = (1, (2, (3, None)))
total = listSum(myList)
```

Figure 4 – Python Tutor interface.

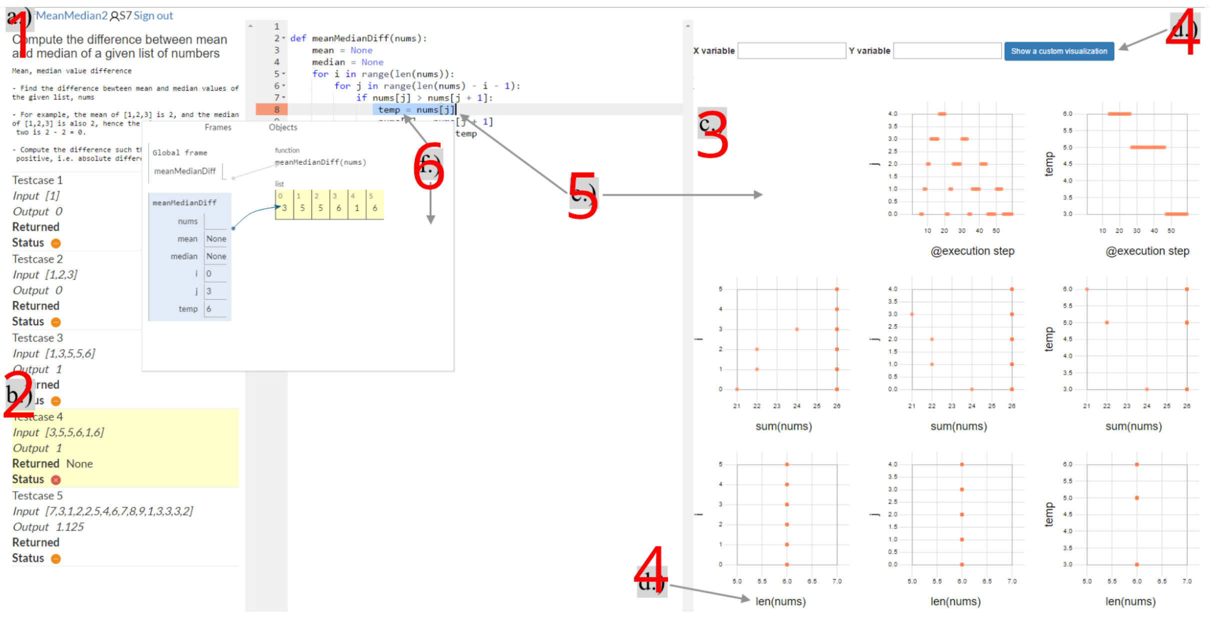
The screenshot shows the visualization of a linked list. The components showed in the visualization are (1) the source code provided to the tool with highlights on the lines being executed, (2) slider and buttons for code step navigation, (3) view of the program stack frames, scopes and variables and (5) view of the program heap, showing allocated objects and references.

Most types of objects can be represented in Python Tutor, with dedicated representations for some built-in data structures such as sets and dictionaries, Figure 4 shows the representation of a linked list made of Python's native tuples. However, Python Tutor can only understand simple linear data structures, such as linked lists, stacks and queues. More complex data structures, such as trees, are still displayed, but the objects positioning does not represent the data structure, and modifications in the structure completely change the object layout. The behavior of Python Tutor visualizations in these situations makes harder to understand data structures, which discourages its use in a Data Structures and Algorithms course.

Due to Python Tutor success and the fact that it is open source, many new tools where created using its source code as base. Some of these tools are Omnicode (KANG; GUO, 2017) and OPT+GRAPH (DIEN; ASNAR, 2018), that provide new elements to the visualization and others tools such as Codeopticon (GUO, 2015) and Codechella (GUO; WHITE; ZANELATTO, 2015), which allow multiple online users to interact in real time through the tool.

3.4 OMNICODE

The objective of Omnicode is to provide a live programming environment (KRAMER et al., 2014; TANIMOTO, 2013; BURNETT; ATWOOD; WELCH, 1998) that shows the entire history of the entire program execution at once. Like Python Tutor, Omnicode can show the abstract representations of program stack and heap, and in addition, represent the entire history of a program. Omnicode uses scatter plots to show variable values throughout the program execution. These plots are used to represent numeric variables and also some properties derived from other data types (KANG; GUO, 2017), Figure 5 shows an example of a program visualization and scatter plots used to represent the entire program state.



Compute the difference between mean and median of a given list of numbers

```
Mean, median value difference

- Find the difference bewteen mean and median values of
the given list, nums

- For example, the mean of [1,2,3] is 2, and the median
of [1,2,3] is also 2, hence the
 two is 2 - 2 = 0.

- Compute the difference such t
 positive, i.e. absolute differe
```

Testcase 1
Input [1]
Output 0
Returned
Status ●

Testcase 2
Input [1,2,3]
Output 0
Returned
Status ●

Testcase 3
Input [1,3,5,5,6]
Output 1
Returned
Status ●

Testcase 4
Input [3,5,5,6,1,6]
Output 1
Returned None
Status ✗

Testcase 5
Input [7,3,1,2,2,5,4,6,7,8,9,1,3,3,3,2]
Output 1.125
Returned
Status ●

```python

def meanMedianDiff(nums):
    mean = None
    median = None
    for i in range(len(nums)):
        for j in range(len(nums) - i - 1):
            if nums[j] > nums[j + 1]:
                temp = nums[j]
```

meanMedianDiff(nums)

0	1	2	3	4	5
3	5	5	6	1	6

meanMedianDiff	
nums	
mean	None
median	None
i	0
j	3
temp	6

Figure 5 – Omnicode IDE.

Omnicode is a tool based on Python Tutor where users can (1) load programming problems from a library and (2) test cases, (3) see visualizations of the history os variables values throughout the program execution in a matrix of scatter plots, (4) visualize values derived from native data structures and evaluate their own expressions, (5) filter generated visualizations by selecting variables in code and (6) view the stack frames and allocated objects. This image was taken and modified from the original article (KANG; GUO, 2017).

The authors ran a small exploratory study with 10 novice programmers. The study objective was to evaluate if Omnicode could contribute to novices ability to write and understand code, form proper mental models and their ability in explain the program behavior to others. They found that Omnicode can contribute to novice students in both program debugging and as a tool to facilitate communication.

3.5 OPT+GRAPH

Just like Omnicode, OPT+GRAPH is another tool derived from Python Tutor, which is in its name, OPT means Online Python Tutor). OPT+GRAPH is one of few programming visualization tools that offer support to graph data structures (DIEN; ASNAR, 2018).

Graphs are detected based on matching of the three most common ways to represent these data structures, which are adjacency matrices, adjacency lists and edge lists. After detecting a graph data structure, the OPT+GRAPH will render the graph in a dedicated pane as shown in Figure 6.

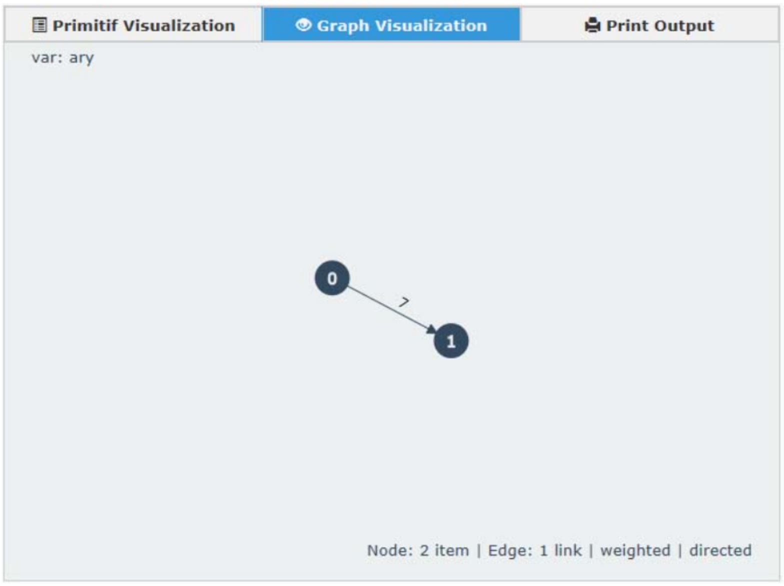


Figure 6 – OPT+GRAPH data structure visualization pane.

Small graph visualization obtained from the OPT+GRAPH tool, other components which are not shown in the picture are similar to Python Tutor's. This image was taken and modified from the original article (DIEN; ASNAR, 2018).

To evaluate OPT+GRAPH, the authors performed an online experiment. The objective was to evaluate the effectiveness of the visualizations provided by the tool based on the correctness and time participants take to answer questions. The presented results showed that visualization tools influenced positively in the time students took to answer questions and the correctness.

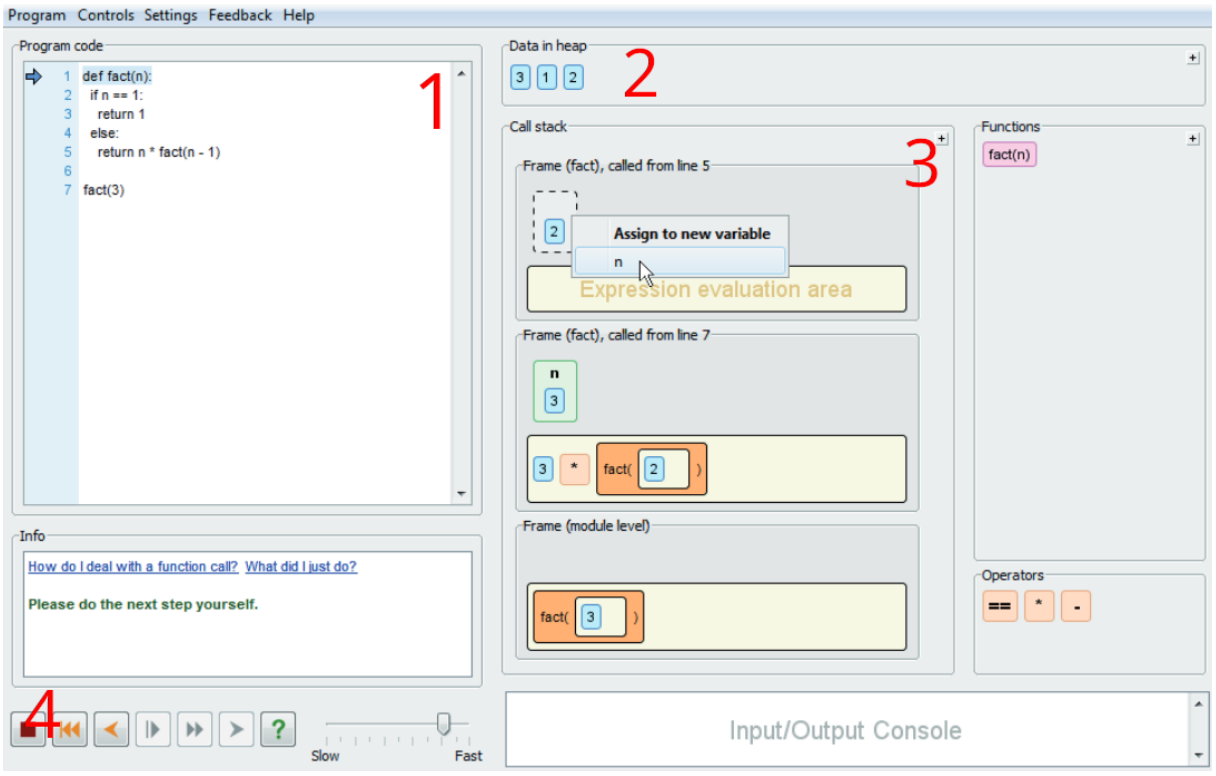
3.6 UUHISLE

UUhisle is another program visualization tool for Introductory Programming and other similar courses, meant for simple and small programs (SORVA et al., 2012; SORVA; LÖNNBERG; MALMI, 2013; UUHISLE,. . . , n.d.). Although only working with simple programs, UUhisle is a flexible tool that can be used for many activities.

Users can use UUhisle to view their program execution, UUhisle acts as a debugger, providing detailed information of the variable values and animations. Professors can use UUhisle to create examples and animations to their students, UUhisle allows professors adjust how the created examples are displayed. Students can also use UUhisle as an Visual Program Simulation tool, where they take the role of the computer and are responsible for carrying the execution of a program. This is possible by allowing users to interact the created visualizations, being able to create variables, call functions and execute many kinds of operations (SORVA et al., 2012). Figure 7 shows the user interacting with the tool interface in the Visual Program Simulation mode.

The authors evaluated UUhisle through a qualitative research with 11 students of Introductory Programming. The authors conducted semi-structured interviews with participants, they had to answer to a Visual Programming Simulation exercise while thinking aloud (SORVA; LÖNNBERG; MALMI, 2013). Supplementary data was also collected from previous studies with UUhisle (SORVA et al., 2012).

The study results were a mix of good and bad results. They found that it is possible for novices to use Visual Program Simulation tool effectively and learn from then in a rich way. However, novices can exhibit different ways of understanding the visualizations, for some participants, learning through VPS is nothing more than learning to perform graphical manipulations. These difficulties must be first addressed to make the tool more useful.



```
1  def fact(n):
2    if n == 1:
3      return 1
4    else:
5      return n * fact(n - 1)
6
7  fact(3)
```

Please do the next step yourself.

Figure 7 – UUhisle, a visualization tool for Introductory Programming Education.

Interface of UUhisle in Visual Program Simulation mode, the user is manually executing a program containing a recursive factorial function. The interface shows the (1) code and line being executed, a (2) basic visualization of objects allocated in the heap, (3) information of variables and values of the program stack, (4) buttons for code execution to step forward and backward.

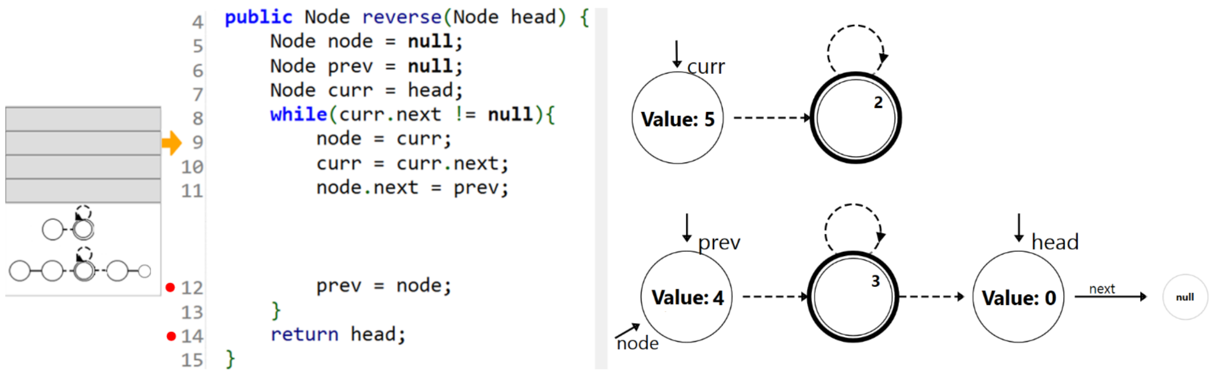
3.7 FLUIDEDIT

Different from previous tools that focused in novices from Introductory Programming courses, FluidEdit tries to help students from Data Structures and Algorithms by providing a method for automatic heap representation that can be used to focus on essential parts of data structures (OU; VECHEV; HILLIGES, 2015).

FluidEdit is a tool to help development, analysis and debugging of data structures. The tool tries to generate visualizations that automatically capture only the essential elements at any given point of the program being analyzed while abstracting the rest. This feature allows users to focus only in the local elements of their program.

FluidEdit also allows users to stop and continue the execution of a program at any point, and interact with the elements of data structure being analyzed by showing, hiding and modifying program objects.

The authors evaluated FluidEdit with 27 participants, ranging from undergraduates to post-doctoral researchers. The experiment main objective was to verify if visualizing the heap of a program could improve the code understanding and help participants to



```
public Node reverse(Node head) {
    Node node = null;
    Node prev = null;
    Node curr = head;
    while(curr.next != null){
        node = curr;
        curr = curr.next;
        node.next = prev;


        prev = node;
    }
    return head;
}
```

Figure 8 – FluidEdit interface.

FluidEdit showing a linked list data structure being reversed. This image was taken and modified from the original article (KANG; GUO, 2017).

find errors in the source code if compared to a traditional debugger. The study reported that FluidEdit helps users to detect errors in data structures algorithms faster than using debuggers provided by IDEs.

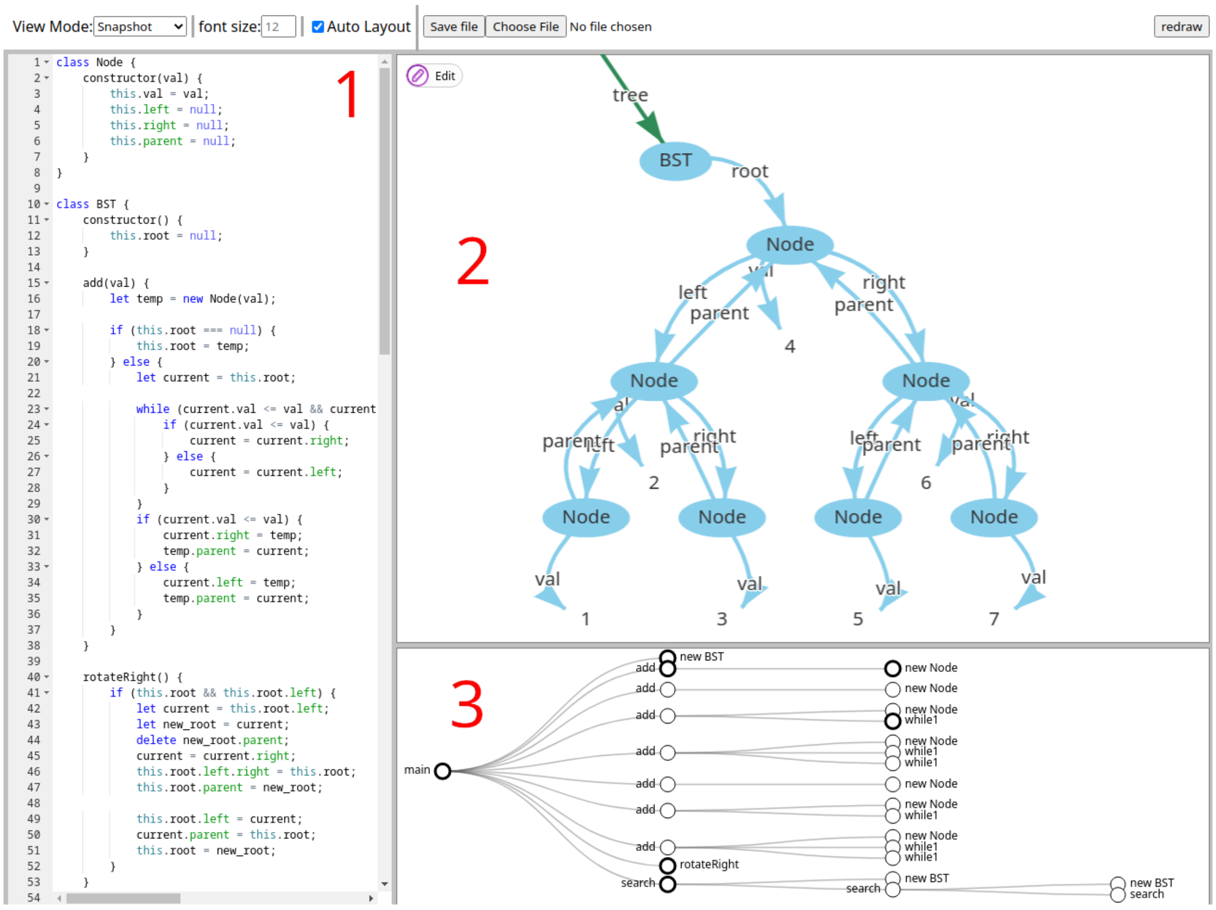
3.8 KANON

Kanon[6] is a development environment focused in the visualization of data structures (OKA; MASUHARA; AOTANI, 2018; OKA; MASUHARA; IMAI, et al., 2017). The objective of this tool however, is not to help novices or Data Structures and Algorithms students, but for any programmer in general that is developing data structures. Although not geared towards students, Kanon provides similar visualizations of program heaps similar to many other educational tools, showing objects, their properties and references.

One of the Kanon main features is the ability to automatically compute data structure layouts, preserve and update them as the user navigates through the code. In Figure 9, the layout was automatically computed by Kanon. This feature is called mental map preservation, and help users preserve the representation of the data structure in their minds by not abruptly changing the layout of objects in the tool. Although this tool is interesting for data structures based on nodes, such as linked lists and trees, the same representation is used for all objects, which makes impossible to visualize data structures based on arrays. Another interesting feature is the support for live programming (KRAMER et al., 2014; TANIMOTO, 2013; BURNETT; ATWOOD; WELCH, 1998), which updates the visualization immediately and renders the program visualization to the point the user is editing.

The authors of Kanon ran an small study with 13 participants, all of then were pro-

[6] `https://prg-titech.github.io/Kanon/`



```
class Node {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}

class BST {
    constructor() {
        this.root = null;
    }

    add(val) {
        let temp = new Node(val);

        if (this.root === null) {
            this.root = temp;
        } else {
            let current = this.root;

            while (current.val <= val && current
                if (current.val <= val) {
                    current = current.right;
                } else {
                    current = current.left;
                }
            }
            if (current.val <= val) {
                current.right = temp;
                temp.parent = current;
            } else {
                current.left = temp;
                temp.parent = current;
            }
        }
    }

    rotateRight() {
        if (this.root && this.root.left) {
            let current = this.root.left;
            let new_root = current;
            delete new_root.parent;
            current = current.right;
            this.root.left.right = this.root;
            this.root.parent = new_root;

            this.root.left = current;
            current.parent = this.root;
            this.root = new_root;
        }
    }
```

Figure 9 – Kanon interface.

Kanon showing the visualization of a binary search tree. On the left (1) the user can provide any code to be executed, by simply changing the code, the visualization is immediately recomputed and presented in the heap. The heap view (2) shows all allocated objects, their properties and references, green arrows represent variables from the stack. Kanon also provide a basic overview of the program stack trace (3), showing functions called and their order.

fessional developers. The study was a small qualitative evaluation of the participants' impressions about the tool. They asked participants to try solving programming problems using Kanon and common textual environments (IDEs). No difference was found between Kanon and textual environments, although participants were positive about using Kanon, the authors reported that participants took longer to resolve errors and used inappropriate strategies to solve problems when using Kanon.

3.9 SURVEYS

All tools presented in this Chapter were developed in the last decade. Still, there are hundreds of older Algorithm Visualization and Software Visualization tools and studies made before that. Surveys of older systems can be found in the following references (SORVA; KARAVIRTA; MALMI, 2013; FOUH; AKBAR; SHAFFER, 2012; SHAF-

FER et al., 2010; URQUIZA-FUENTES; VELÁZQUEZ-ITURBIDE, 2009).

Section 4.3 compares the visualizations of some of the tools reviewed in this Chapter with our proposed tool.

4 WILLOW

This chapter presents WILLOW, an educational tool for generating program visualizations. In the following sections we give a basic introduction of the tool, then we present design details and a brief overview of the tool architecture.

WILLOW is a Program Animation tool (Section 2.4.2), therefore, it can create visualization directly from the source code of a program. WILLOW focuses in providing features that allow users to manipulate the generated visual elements, allowing the creation of more expressive visualizations, which can be useful for representing data structures and algorithms.

Because WILLOW requires extra interactions to create visualizations, the tool is mainly targeted at instructors, they can use the tool to create lecture material by creating visualizations for the programs they want to teach, and testing the program with different input data. Although instructors are the main target, students can also use the tool without any hassle for a variety of activities such as developing programs with visual support, analysis and debugging of data structures and algorithms, understanding basic programming concepts ans so forth.

WILLOW's source code, website and other resources are available through the following URL: `https://github.com/pedro00dk/willow`.

Videos with examples of WILLOW's visualizations are available online on YouTube and can be accessed through the following playlist:

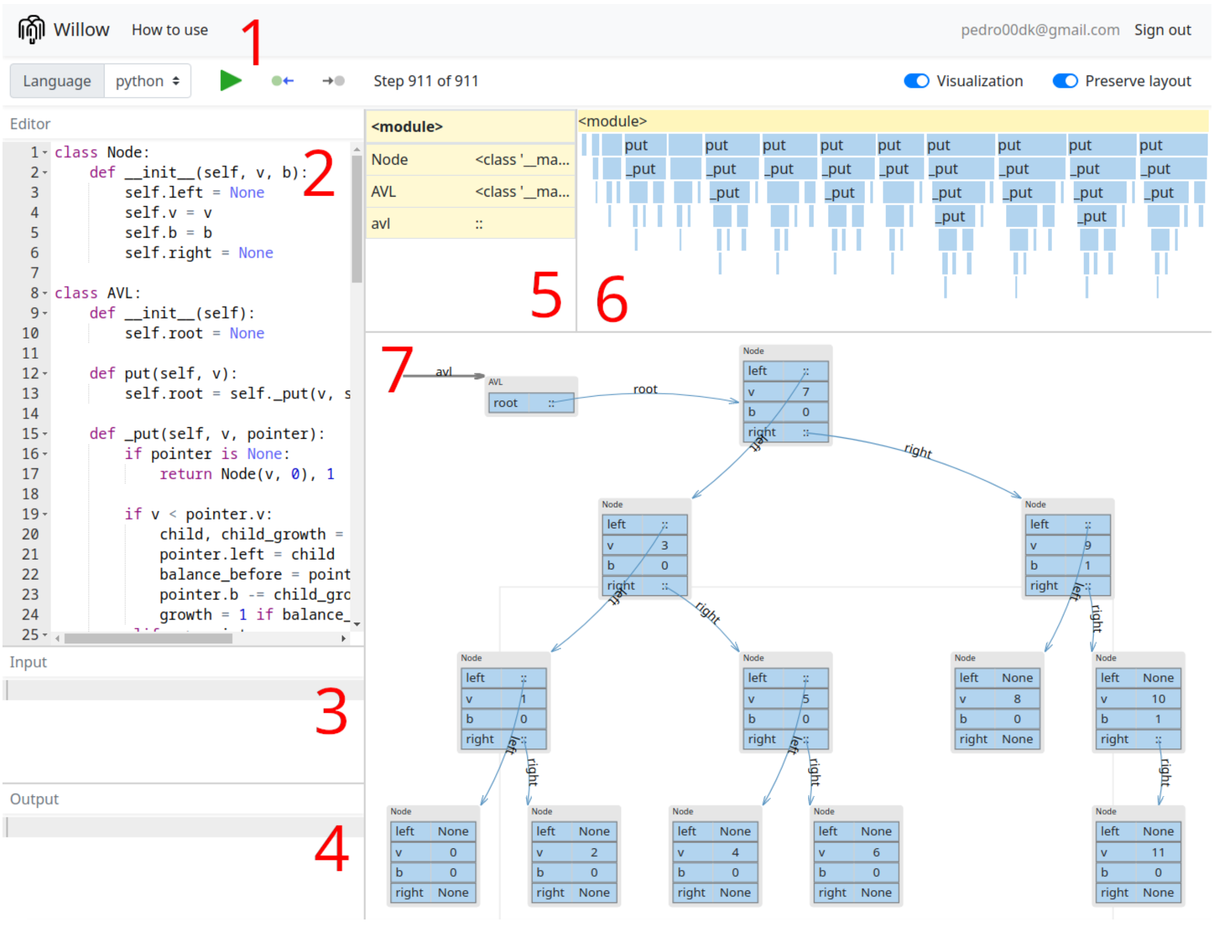
`https://www.youtube.com/playlist?list=PLpNZKTBEk73m5-DcKbpc45PZIe8WbEsj2`

4.1 DESIGN

Figure 10 shows a screenshot of WILLOW. The tool is divided into two main groups of components. On the left side we have the source code and input editors, and the output pane. On the right side there are the visualization components, that display the program stack and heap representation.

WILLOW visualizations are based on *Stack* and *Heap* memory abstractions, which are common in several programming languages. Many elements of WILLOW's visualizations are inspired by *Python Tutor* (Figure 4), but with extra features to support the representation of more complex algorithms and data structures.

By interacting with WILLOW's visualizations, users can modify them to represent more complex concepts. Data structures and algorithms that rely on uni-dimensional or bi-dimensional arrays such as sorting algorithms, binary search, heaps, dynamic programming and more, can be represented using special data representations that help the visualization of interactions on these structures. Node based data structures such as linked



```
class Node:
    def __init__(self, v, b):
        self.left = None
        self.v = v
        self.b = b
        self.right = None

class AVL:
    def __init__(self):
        self.root = None

    def put(self, v):
        self.root = self._put(v, s

    def _put(self, v, pointer):
        if pointer is None:
            return Node(v, 0), 1

        if v < pointer.v:
            child, child_growth =
            pointer.left = child
            balance_before = point
            pointer.b -= child_grc
            growth = 1 if balance_
```

<module>	
Node	<class '_ma...
AVL	<class '_ma...
avl	::

Figure 10 – Willow interface.

Screenshot of Willow shows a visualization of a balanced tree data structure. At the top (1) there is the utility bar, which contains language, navigation and some visualization controls, as well as the information of the current program execution step. On the left side (2, 3, 4) there are respectively the source code editor, the input editor and the output pane. The list of stack frames, their variables ans values is displayed in the center (5), on the right side (6) there is the program call tree, which shows all created scopes. The heap visualization (7) shows the allocated objects, their properties names and values, and references among objects and from variables of the stack.

lists, queues, stacks, deques and all kinds of binary search trees can also be represented and even animated by Willow.

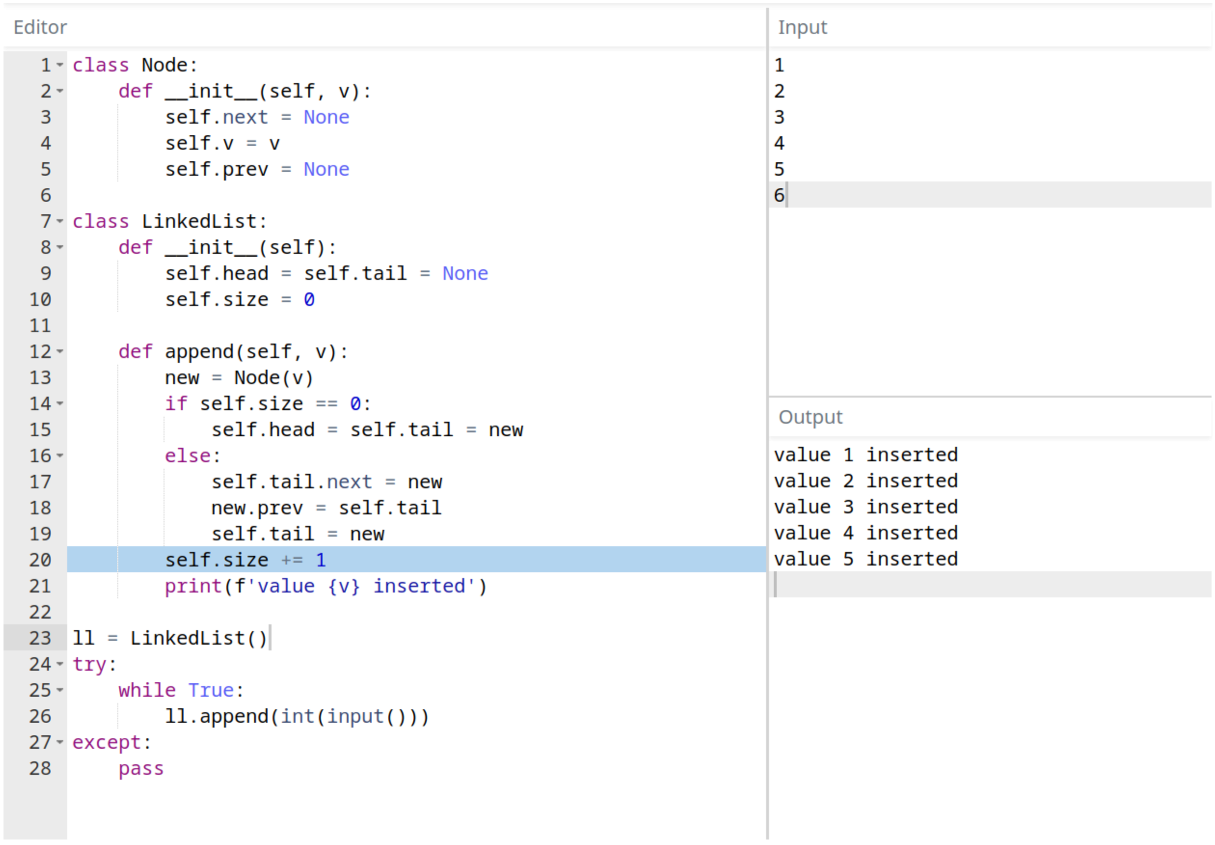
Willow is implemented as a web based tool, it can be accessed without installing any software. This feature makes the tool easily available for instructors and students, which can access it from any computer with an internet connection.

4.1.1 Editors

The source editor allows users to provide any code they want to execute. Basic syntax highlight and snippets are provided automatically according to the selected language. During the code execution phase, the source editor also provides information on which line is going to be executed and the type of operation the code executed. These operations are distinguished by the the line highlight color, they are: function calls (green), function returns (yellow) and raised exceptions (red).

The input editor allows users to provide input to the program to be executed. This editor acts as the program standard input stream, meaning that users can use simple ways to read input data as if the code would have been executing locally. Because of that, it is easier to parameterize the code and test different cases just by changing the input.

Besides the created program visualizations, WILLOW also outputs any text information the program generates. Any data written to the standard output and standard error streams is displayed in the output pane. This includes printed messages, uncaught exceptions and compilation errors.



```python
class Node:
    def __init__(self, v):
        self.next = None
        self.v = v
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = None
        self.size = 0

    def append(self, v):
        new = Node(v)
        if self.size == 0:
            self.head = self.tail = new
        else:
            self.tail.next = new
            new.prev = self.tail
            self.tail = new
        self.size += 1
        print(f'value {v} inserted')

ll = LinkedList()
try:
    while True:
        ll.append(int(input()))
except:
    pass
```

```
1
2
3
4
5
6
```

```
value 1 inserted
value 2 inserted
value 3 inserted
value 4 inserted
value 5 inserted
```

Figure 11 – WILLOW's editors.

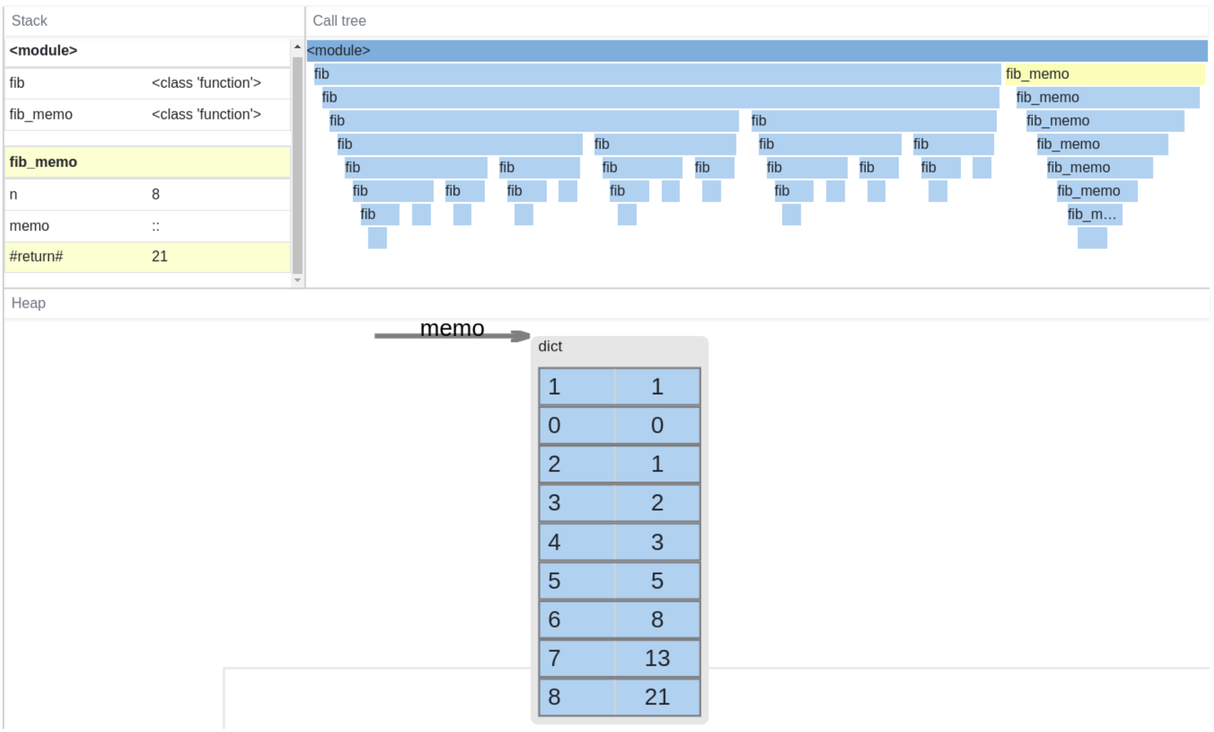
WILLOW's editors, they are shown in a different layout from Figure 10 because the visualization is disabled.

4.1.2 Stack and Call Tree

The stack component shows all program scopes (function calls) and their declared variables at the current program execution point. When the user navigates through the program, the stack also highlights variables that changed value, allowing easy identification of what has been modified. Stack scopes also show function return values and raised exceptions, even though they are not captured by any variable.

The call tree component displays all function calls performed during program execution. It shows their names, the caller and which other functions they call. It also highlights the scope of the current execution point. The call tree is specially useful to navigate through the program by clicking on the function scopes, which makes the current execution point jump to the beginning selected scope, or by double clicking to go to the end of the scope instead.

Figure 12 shows the stack and call tree of two Fibonacci sequence algorithms, the first implementation (fib) uses the naive recursive strategy, while the second implementation (fib_memo) uses memoization. Similarly to the source code editor, both stack and call tree show color codes to indicate function calls, return and exceptions.



<module>	
fib	<class 'function'>
fib_memo	<class 'function'>
fib_memo	
n	8
memo	::
#return#	21

1	1
0	0
2	1
3	2
4	3
5	5
6	8
7	13
8	21

Figure 12 – WILLOW's stack and call tree of a Fibonacci sequence algorithm.

Screenshot of the execution of two Fibonacci sequence algorithms. The *Call Tree* component on the top right shows the recursive behaviour of both implementations. The first algorithm (left to right) is the naive implementation, while the second algorithm is a memoized version. The *Stack* on the top left shows the last scope of the Fibonacci algorithm, the yellow color and the "`#return#`" variable indicates the scope is returning. On the bottom, the *Heap* component shows the dictionary used by the memoized algorithm.

4.1.3 Heap

The *heap* shows objects created by the program at some execution point. This component is where users can interact with the visualizations, the interactions allow users to modify the visual representation of objects, change their layout and create animations.

The *heap* component does not display all objects of a program, many objects are omitted, such as builtin objects and strings, since these objects are allocated in the heap, visualizations would get too polluted. Each program object is associated to a node type, which is how WILLOW visually represents the objects in the view. However, objects are not tightly coupled to their node types. This means that they can be changed if needed, changing the way the object is displayed.

4.1.3.1 Node Types

Figure 13 shows WILLOW's nodes. Four types of nodes are currently supported, they are:

- **Array**: The array node shows all fields of the underlying object as a list of its values. Array is the default node for integer indexable types, such as implementations of arrays and lists in different languages.
- **Columns**: This is an alternative to visualize numeric arrays as a column chart, which can help noticing when changes happen in the underlying object. This node can be used to highlight swap operations in sorting algorithms. The arrangement helps the user to understand how elements are sorted. The visualization is based in common patterns found in examples of sorting algorithms.
- **Map**: This node displays object fields as a pair of columns of keys and values. Map is usually used for dictionary-like objects, Figures 10 and 12 use this node to represent objects.
- **Field**: Field is used to show a single property of an array or object. It is useful for representing user-created data structures, where objects may contain many properties, making it cleaner and easier for understanding.

Each node type also comes with a set of extra parameters, which can be modified by the user through context menus, affecting the way the node is rendered.

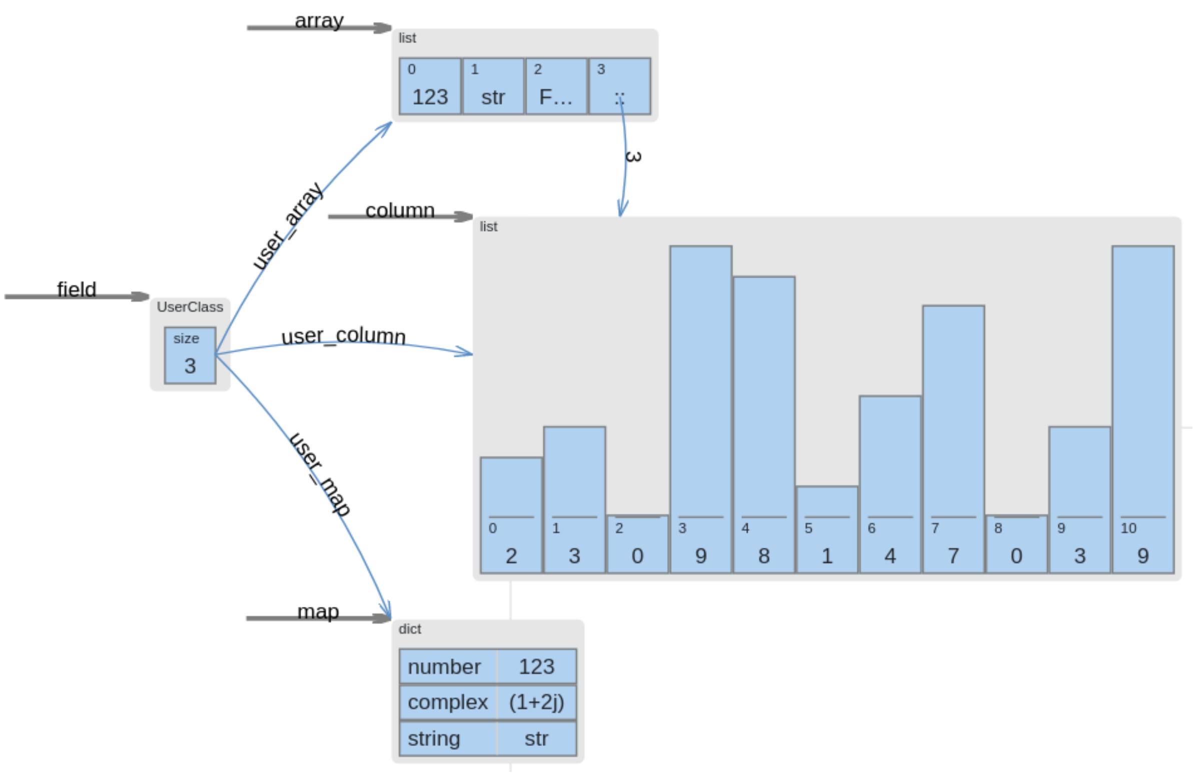


Figure 13 – Willow's node representations.

All four node types currently supported by Willow.

4.1.3.2 Positioning and Animation

All nodes rendered by Willow can be moved by the user. Their positions are remembered by Willow and, when the user goes through the same section of code again, Willow replays all previous object positions. This feature can be used to create animations for many kinds of algorithms, e.g. linked list insertions or balanced tree rotations.

Another feature is the detection and automatic layout of groups of objects that belong to common data structures such as lists or trees. The automatic layout feature is triggered by the user by double-clicking any object that belongs to a data structure, which applies the layout to the inner elements that compose the data structure. Nevertheless, there are some restrictions in the data structure detection, which are as follows:

- The inner data structure elements must be made of objects of the same type.
- Objects of a data structure can contain references to other objects which store values, but these values must have a different type from the data structure objects, otherwise they will be detected as part of the structure.

By combining detection of data structures, automatic layout and re-positioning, users can quickly create animations of entire data structures without having to move objects one by one.

4.1.3.3 *Program Navigation*

WILLOW provides three ways to navigate though the program being visualized. The user can use the keyboard left and right arrows, or click on the step forward and step backward buttons in the utility bar (Figure 10). WILLOW allows navigation in both directions of the program, the user can go forward or backward, even when an exception is reached during the program execution, this feature is called *time travel navigation*.

It is also possible to click on the scope of a function displayed in the call tree (Figure 12), allowing users to jump to any point in the program they desire, skipping parts of the program that would not be useful to the visualization. By jumping from an scope to another, all the accumulated differences in the objects and variables of a program are highlighted, providing an overview of all changes.

4.1.4 Language Support

WILLOW mainly supports the Python programming language, which is a popular language among introductory courses. In the recent years, Python has been adopted as the Introductory Programming language across many Universities. MIT[1] and UC Berkeley, some of the largest departments of Computer Science use Python. Several online courses also use Python in their introductory courses (ATEEQ et al., 2014; GUO, 2013).

Despite Python's growing popularity, many Introductory Programming courses still use Java or even C/C++ as their programming language. Based on that, Willow also provides support for Java. The current language can be switched easily through the utility bar as shown in Figure 10.

4.1.5 Limitations

Although being a Program Animation tool that can execute user provided source code, there are some limitations on what can be executed and for how long. Some of these limitations come from Willow design itself, where others are dependent on the selected language back-end.

The main limitations are:

- Willow expects that all code of the program is provided in a single file. Since our main goal is to support Introductory Programming and Data Structures and

[1] `https://ocw.mit.edu/courses/intro-programming`

Algorithms classes, we believe that requiring everything to be in a single file is reasonable. Still, this limitation can confuse novices in some situations, such as declaring many classes in a single file.

- Programs have restricted time to execute. Time restriction comes in two forms: the duration of the program execution in seconds and the number of source code steps executed. Source code steps are approximately linked to the source code lines. A source code line usually represents a single program step, but in some cases more steps are executed per line.

Programming language limitations are related to the access of languages libraries and features. These limitations include blocked access to the file system, network and multi-threading libraries. These limitations were intentionally added to prevent abuse against the tool.

There are also limitations related to the visualization. Most simple data structures and algorithms have abstract representations similar to machine representations, e.g. each linked list node is an object or struct. These data structures are easy to represent using Willow. More complex data structure are harder to, or can not be represent using Willow. Graphs are an example of structures of which Willow can not create good abstract representations. This limitation is due to different ways of representing graphs, such as adjacency matrix, adjacency list and edges list. These machine representations of graphs are not similar to its abstract representation. In this case, the generated visualizations would run over the machine representation of the graph, not an abstract representation. Willow can however be used to highlight the differences of the machine representations, showing their advantages and disadvantages

4.2 ARCHITECTURE

Figure 14 shows WILLOW's architecture, which is composed of three elements: the user client, a database to log user actions, and tracers, which are responsible for executing and collecting program runtime information.

4.2.1 Client

WILLOW's client is a web application implemented using REACT[2], a popular library for creating user interfaces. The client is composed of several modules, mainly split into user, tracer and visualization tasks.

[2] `https://reactjs.org`

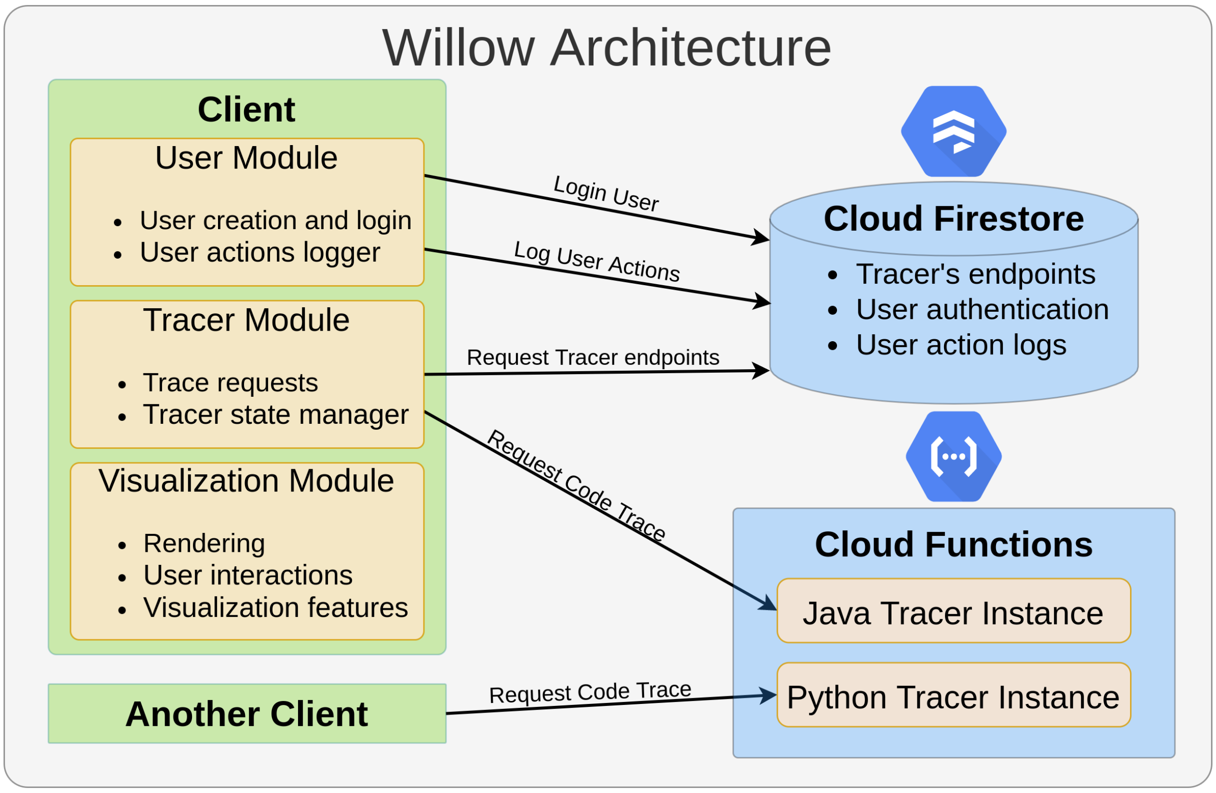


Figure 14 – WILLOW's architecture.

User task modules are in charge of signing in and out the user and collecting activity logs yielded by other modules. These activities are temporarily stored in a cache and then uploaded to the database every few seconds.

Tracer tasks manage tracer languages and URLs collected from the database, and issue trace requests to tracers. Trace requests contain the code to be executed as well as text input. Once the request is answered, the tracer module creates an index to access any point of the program in execution order, by line of code or by name of any function called during code execution.

After a successful code trace, the user will have access to the interface elements that allow navigating through the code as described in Section 4.1.3.3. When the user selects a program point to be visualized, the visualization modules update the stack and heap views, checking if there are any visualization modifications applied by the user, analysing the heap graph looking for data structures, and applying automatic positioning, if requested by the user. When the user jumps to another point of the program, the visualization modules verify if there is any new positioning recorded for objects at the new program point and smoothly animate these objects from the old to the new position.

4.2.2 Database

Willow's database is implemented with Google's Firestore database services, which already provides user authentication services. Since this is mainly used for logging purposes, there is no need for a back-end to manage authentication or other complex tasks. The database stores actions users execute in the client, including the source code; input and language used in tracing requests; a basic summary of the request result, indicating if the code executed successfully or if there was any compilation or runtime error; and user interactions with the visualization, such as steps using the keyboard, arrows or call tree scopes, changes in object representation and the use of automatic layout detection and positioning features. The database also stores currently available tracers in a table with tracer languages and URLs.

4.2.3 Tracers

Tracers are the components which conduct all necessary code compilation, execution and analysis. Tracers work similarly to traditional debuggers. When a tracer receives a code trace request from a client, it executes the code and creates records of the program state at each step. The record contains information about the line of code that was executed. For instance, if the current line is a function call, a function return or if an exception is being raised, the variables declared in each scope of the program and their values at that specific execution point, as well as objects allocated in the heap and their properties. At the end of the program analysis, the record is serialized into JSON and sent back to the client.

Because Willow only interacts with tracers through HTTP requests, they are completely independent from other Willow's modules and can be implemented using any technology. Previous versions of Willow had their tracers implemented using IAAS technologies (Infrastructure as a service) and Docker, which allowed high control of the resources that tracers had access, and fast initial response times. However, since the tool was not frequently used, the cost of maintaining the virtual machine provided by the IAAS was unnecessarily high. The current Willow version implements tracers using FAAS technologies (Function as a service, also known as serverless) provided by Google Cloud Functions[3]. Since tracers are stateless, the migration from IAAS was fairly simple and yielded a large reduction in costs, while increasing system scalability.

Tracers can be implemented in any language. Both Java and Python tracers are implemented in their respective languages to benefit from features that each programming language provides to support the implementation of debuggers.

[3] `https://cloud.google.com/functions`

The Java Tracer uses the Java Platform Debugger Architecture (JPDA), which is used to spawn a new JVM process in debug mode (also called debuggee JVM) to execute the requested code, and then connect the current JVM using the Java Debug Interface (JDI). Interruptions are sent to the debuggee JVM and its current state is recorded through the JDI.

The Python Tracer is simpler than the Java Tracer, since Python code does not require compilation, and also, for performance reasons, the code to be traced is executed using Python's builtin `exec` function. The `exec` function also allows the specification of an initial global scope, which is used to create a new empty scope and prevent variable contamination between the code to be traced and the tracer itself. The specification of a global scope is also used to restrict access to some builtin function and modules. Python offers its debugging capabilities through the `settrace` function of the `sys` module. After setting a trace function, it is called after each executed line of code. The trace function has access to the program scope, making easy to directly collect all record data.

4.3 COMPARISON WITH OTHER TOOLS

Program Animation tools generally provide support for only simple algorithms and structures common to introductory coding courses. Even most recent tools have limitations when handling some data types or data structures. WILLOW's extends on these tools by providing features that support the visualization of a larger set of algorithms and data structures.

Python Tutor, for instance, cannot represent data structures more complex than linked lists, even doubly linked lists diagrams are drawn in messy layouts, which makes harder to visualize the data structure. This is due to the static visualizations, making impossible change how objects are represented or change their positions. Figure 15 shows the Python Tutor execution of same balanced tree algorithm presented in Figure 10. The final diagram layout is messy and hard to understand.

Another advantage of WILLOW over Python Tutor is the program navigation. Python Tutor visualizations only provide a slider where users can drag to choose a point of of the program to be rendered. WILLOW on the other hand, allows users to precisely navigate to any function call of the program through the Call Tree.

Kanon (OKA; MASUHARA; AOTANI, 2018) is another recent Program Animation tool with similar features to WILLOW. Similarly to WILLOW, Kanon supports the detection of some data structures, such as lists and trees, and it has some other interesting features such as live programming and code navigation based on line of code being edited.

However, there are some disadvantages in using Kanon with educational purposes. Kanon only supports the JavaScript programming language, which is not commonly adopted as an introductory language. Another disadvantage of Kanon is the represen-

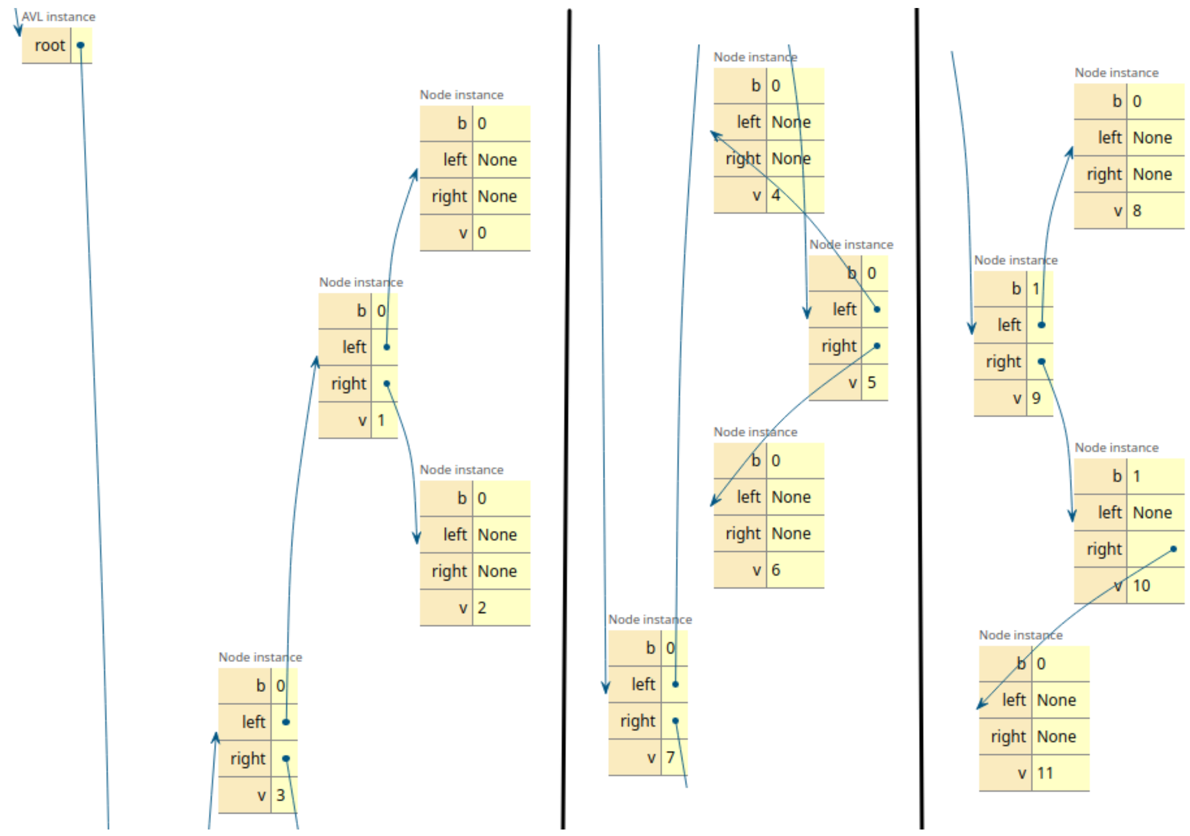


Figure 15 – Python Tutor visualization of a tree structure.

The illustration is cut in three parts because is too long and elements are badly positioned, making it harder to navigate.

tation adopted for arrays, which are extremely poor and cannot be used to illustrate algorithms.

Figure 16 shows how Kanon represents arrays, as opposed to Willow in Figures 13, which has more than one way to represent data of different types. Kanon uses a representation that does not preserve the order of elements, which makes the visualization impractical to represent several array based algorithms and data structures.

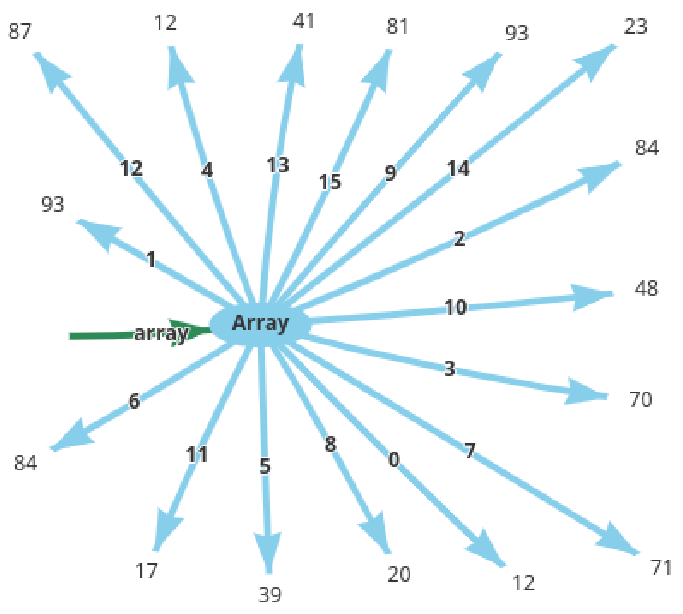


Figure 16 – Kanon's representation of arrays.

This representation of an array has no ordering structure; therefore, it cannot be properly used in array based algorithms.

5 EVALUATION

This chapter discusses in detail the studies we carried to evaluate WILLOW and the obtained results. We ran two studies to evaluate WILLOW. The first is a survey with instructors of Introductory Programming (IP), and Data Structures and Algorithms (DSA) courses, and the second is an experiment with programmers.

The main goal of the survey was to evaluate the potential impact of WILLOW as a teaching aid. The specific goals of the survey are 1) to understand the practices that are commonly adopted by instructors when teaching (e.g., how often they use the board as opposed to slide presentations and other strategies) and 2) to understand their opinions on the usefulness of WILLOW, relative to the quality of the produced examples, resistance and difficulties on the adoption of the tool in class, and opinions on the impact of the tool on their students.

Based on encouraging results obtained in the survey, we conducted a second study where we recruited participants from the freelancing platform *upwork*[1]. In the experiment we switched the focus from the instructor (analysed in the survey) to the student. During the experiment, participants had to solve programming problems with and without the support of WILLOW. The goal of the experiment was to evaluate how can WILLOW assist participants in solving programming problems and to what extent. More precisely, we assessed if participants that use WILLOW can produce more correct solutions, if the tool helped them to solve problems faster, and also if there were any difference in the their confidence when solving the problems with or without WILLOW.

5.1 SURVEY

This section elaborates on the study design of the survey we conducted to assess the potential impact of WILLOW as an aid to instructors of IP and DSA courses.

5.1.1 Objective

The survey objective is **to analyse** the perceptions and opinions of participants about WILLOW, **with the purpose of** comparing the tool with traditional teaching strategies adopted by the participants, **with respect to** the benefits WILLOW may provide in the creation of programming concepts and algorithms examples, **from the point of view of** instructors of IT courses, **in the context of** Introductory Programming, and Algorithms and Data Structures courses.

[1] https://www.upwork.com/

5.1.2 Research Questions

We translated our objective in two research questions:

- RQ1: What are the most common practices used by instructors when teaching coding-related courses?
- RQ2: What are the perceptions of these instructors towards WILLOW?

The first question helps us to understand the behavior of instructors when teaching code-related courses. For example, a "hands-on" instructor that engages with students in writing code during lectures is more likely to react positively to WILLOW and other SV tools. This question therefore attempts to characterize the teacher. The second question evaluates the reactions of teachers based on observations of the key features of WILLOW.

5.1.3 Research Method

To answer our research questions, we adopted a mixed methods research method (LAMBERT; LAMBERT, 2012). Since we collect participants perceptions and opinions about WILLOW, this survey can be classified as a descriptive study (LAMBERT; LAMBERT, 2012), where outcomes are unlikely to change by carrying the study multiple times. This allowed us to carry out the study in a cross-sectional manner (WOHLIN et al., 2012; LEVIN, 2006) (run the study only once), because there are no factors in the study that require a longitudinal application.

5.1.4 Population

The population of this study is composed of instructors that teach or have taught in IT-related courses, such as Computer Science, Computer Engineering, Software Engineering and so forth. Since the study is carried out only once (Subsection 5.1.3) and we use an online questionnaire to collect participants' data (Subsection 5.1.5), we did not present a participation agreement form to the participants. Because of that, the consent term and ethical concerns where included in the questionnaire.

5.1.5 Questionnaire

We used online questionnaires, which is an indirect strategy for data collection (where there is no physical interaction between researchers and participants).

The questionnaire was designed with two main goals:

1. Identify the practices that are most commonly adopted by instructors in their classrooms (to answer RQ1).

2. Identify their perceptions about Willow (to answer RQ2).

The questionnaire presents the study objectives, researchers' names and contact information, a description of Willow, the expected duration to answer the questions, concerns about privacy and a consent term. The questionnaire contains mainly closed questions, containing single choice, multiple choices and Likert scale questions. All Likert scale questions in the questionnaire are followed by optional open questions, where participants could provide rationale for their answers if they wanted.

In the first part, we asked whether or not instructors of IP and DSA courses adopt certain practices in the classroom, for example, sketch diagrams on the board, use programming tools during class, etc. In the second part, we prepared various short videos of approximately 2 minutes to demonstrate the tool features using small examples of data structures and algorithms which are related to the courses we focused. We also provided a link to Willow so that participants could give a go on the tool, if they wanted.

In total, 6 videos were presented in the questionnaire, 3 videos contained basic concepts commonly taught in Introductory Programming courses, the remaining 3 videos contained examples commonly taught in Data Structures and Algorithms course. All videos used the *Python* language. In what follows we detail each video.

- Introductory Programming:

  - *Tuples, Lists and Mutation*: This example shows the creation of these data structures, followed by some operations that can be executed with them, such as indexing elements, slicing, update, insertion and removal in lists, and sorting. The goal of this video is to demonstrate which operations are possible with each data structure, what happens when one attempts to change the contents of an immutable data structure, and consequently demonstrate the difference between mutable and immutable data structures.

  - *Recursion*: This video presents the visualization of a factorial function. The main goal of the visualization is to shows the behavior of recursive functions with respect to the program stack. In the video, it is possible to see how the function scopes are stacked one after the other in the stack, and the value of their variables preserved during the execution until the function reaches the base case, when they start to be popped.

  - *Objects and Special Functions*: This video show how Willow displays objects, the parameter binding in method calls and how Python can overload operations through the use of functions with special names.

- Data Structures and Algorithms:
  - *Quicksort*: This video shows an example of the classic Quicksort algorithm. To better represent the operations the quicksort algorithm executes in the list, a column based representation is used. The column representation in combination with the default color hints of modifications in the data structure is used to highlight the swap operations executed by the sorting algorithm.
  - *N-Queens*: The second video about Data Structures and Algorithms shows two examples of the N-Queens backtracking algorithm for N=4 and N=5. Similar to the Quicksort example, a different visualization of the list data structure is adopted. This time a matrix is used, and combined again with color hints of modifications, the visualization displays the check areas of the board every time a new queen is placed or removed.
  - *Self-Balancing Tree*: The last example example show the visualization of consecutive insertions in an AVL-Tree. this visualization uses more advanced features, such as data structure detection and automatic positioning (Section 4.1.3.2), and time traveling (Section 4.1.3.3). These features are combined to show animations of the insertion of new nodes in the tree.

Different features of Willow are presented in the videos, such as multiple object representations, delta highlighting, time-traveling, automatic data structure detection and positioning, and animations. The videos are available online on YouTube and can be accessed through the following playlist:

`https://www.youtube.com/playlist?list=PLpNZKTBEk73m5-DcKbpc45PZIe8WbEsj2`

The questionnaire was revised several times in pilot trials. Two instructors not directly involved with Willow offered their assistance to informally assess whether the survey balanced completeness (as to answer the questions we posed) and time-consumption (as to reduce the effort of participants). Their answers were discarded and not used in the analysis. Finally, the questionnaire included background questions (e.g., name, experience, etc.) and questions about practices, questions about the perception of participants on Willow in the context of IP and DSA disciplines, and questions to collect feedback and criticism.

Considering background information, we asked each participant's name, institution, courses taught, years of experience, and demographics. Regarding the teaching practices adopted by instructors, the survey covered different practices that we observed as commonly used and had an open text field for participants to indicate practices that we did not cover. Finally, considering the part related to the perception of participants about Willow, we prepared questions related to the videos in the playlist above, asking participants whether such a tool could help the teaching activities. For example, after showing a video containing examples of tree rotations in Willow, we asked "*Do you agree that the animations shown in the video helps to illustrate tree rotation?*". We collected answers about such perceptions on a 5-point Likert scale. The scale indicates how strongly the participant felt about a question associated with a given video. Participants had the option to provide a rationale for their scores.

Since participants may have experience in only one of the courses of interest, we split videos and questions into two sections contained in the same form, one for each course. We also made all questions optional, and informed that participants may skip a question if that question does not apply. At the end, we collected suggestions, comments, and critics. The form content is available in the Appendix 1.[2]

5.1.6 Results

This section discusses the data we collected from the survey. We sent invitations through email to 1771 instructors that teach or taught IP or DSA and collected a total of 111 responses from them. Not all of the 111 responses where considered in the analysis, 6 participants did not have experience with the courses of interest, and 1 participant answered the form twice, 104 remained. From the 104 participants that answered the survey, 97 of them taught IP, 81 taught DSA, and 74 participants taught both courses.

To send such number of invitations, we established a criteria to gather candidates. Participants should be instructors in one of Brazil's Federal Universities. This criteria was established because it provided us a limit of candidates to send emails. If we chose to collect candidates of any Brazilian institution, it could not be possible to gather candidates from all institutions. Federal Universities are also recognized in Brazil by their above average quality, and are spread across the country, providing us with high diversity data. Figure 17 shows the number of participants per institution.

Figure 18 shows two histograms—one for each course of interest—to summarize characteristics. Each histogram shows the distribution of years of teaching experience of participants on a given course. Although the number of experienced instructors in IP was higher compared to DSA, overall, there was a balanced number of experienced and inexperienced instructors participating in the survey. We considered that the answer of both kinds of participants are important.

5.1.6.1 Participants Teaching Practices

To understand teaching practices adopted by the participants, we presented questions that relate to common strategies used in classes. Figure 19 shows the prevalence of each

[2] The questionnaire and videos were translated to English. The original language is Portuguese.

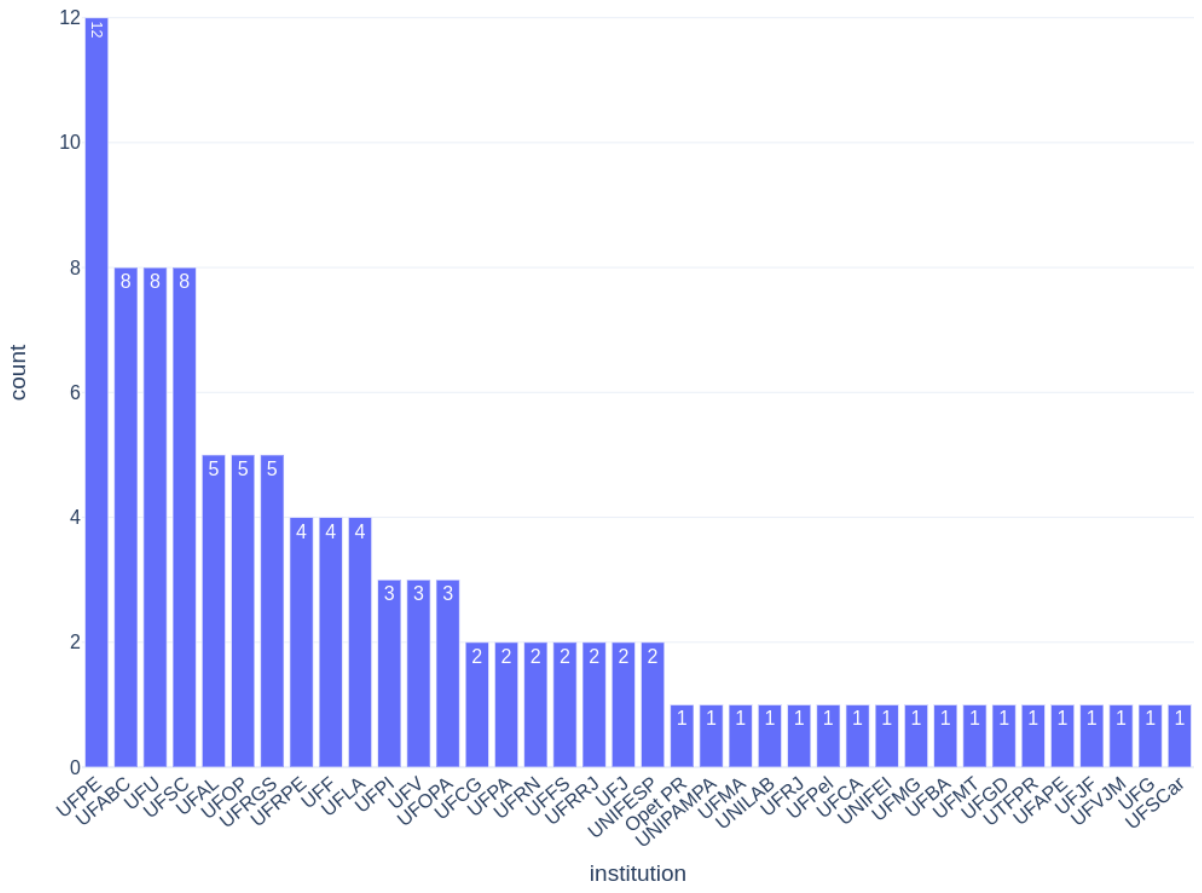


Figure 17 – Participants' teaching institutions.

practice. The size of each bar shows the number of participants who adopted a given practice, as indicated in the survey.

The most commonly-adopted practices in the classroom appears at the top of the figure. The three most common were writing sketches of code and data on the board, writing program code during class, and using slides to illustrate diagrams. Overall, we found that the adoption of interactive visual aids (e.g., visualizations of state transitions, or that allow changes in source code or input) in teaching IP and DSA was relatively low, as indicated in the size of the second to last listed practice on Figure 19.

We also asked questions about instructors' experiences with SV tools, we asked "*Have you ever used any tool that automatically create program visualizations?*", and collected the names of tools that participants have used. Since participants may not be familiar with the definition of Software Visualization (SV), we used a somewhat generic question, and filtered results based on the list of tools that participants provided. For instance, some participants mentioned that they use PowerPoint, which does not qualify as a SV tool. After filtering only those with answers that interest us, we identified that some participants have not necessarily used SV tools to prepare lectures or during the lectures, but they know that such tools exist and what can they can do. In total, 29 participants

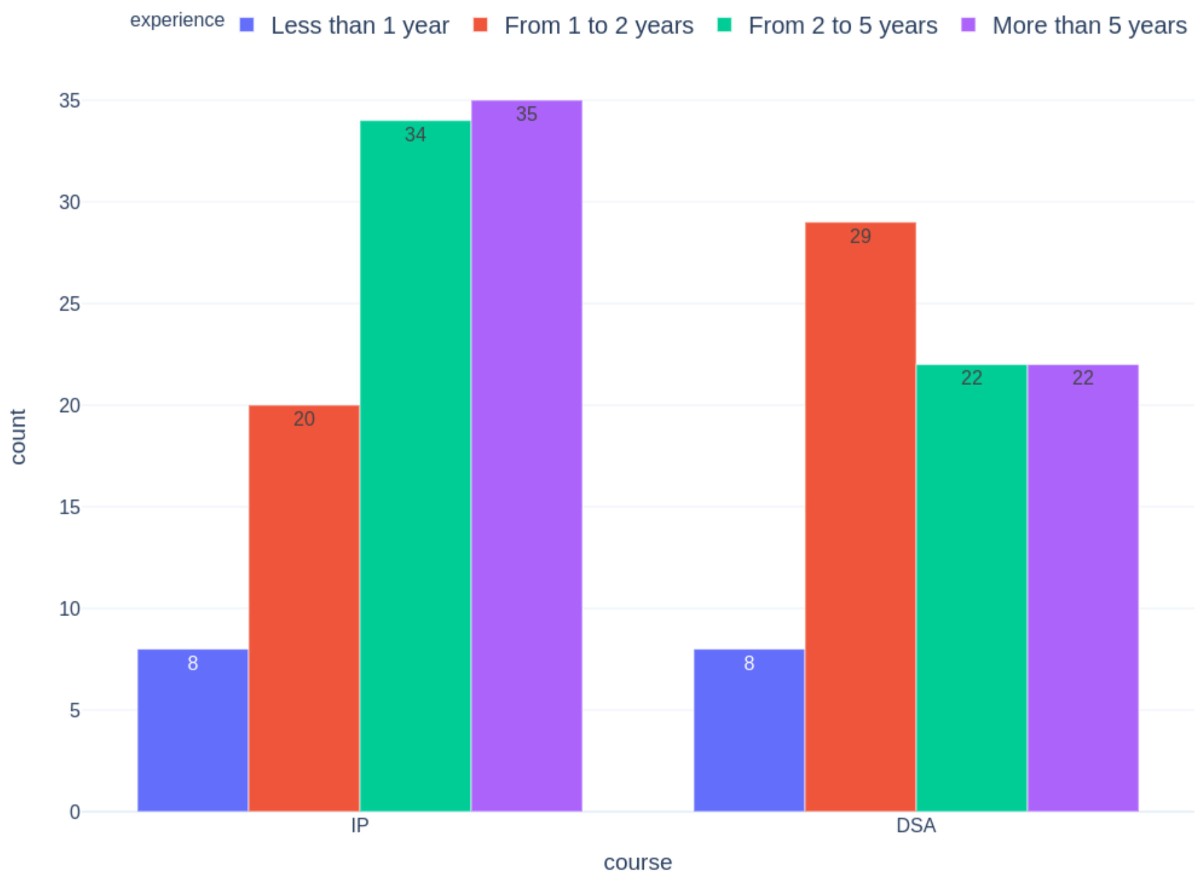


Figure 18 – Participants years of experience with IP and DSA.

have tried SV tools.

Although the use of simple illustrations is common among participants (79.8%), only 29 participants (27.9%) have tried SV tools to create visualizations. Among these, the most commonly used tools were Python Tutor (GUO, 2013) and VisuAlgo (DIXIT; YALAGI, 2017), which had been tried by 16 and 13 instructors respectively, older tools such as Jeliot (3) and BlueJ (3) where also reported, a few remaining tools were reported by one participant each.

Participants also gave their opinions on results obtained from SV tools they have used. Most of them had good experiences, only one participant reported results as unsatisfying. Although we could not verify if instructors indeed used SV tools during class or to create lecture materials, when correlating the use of SV tools with adopted practices, we found that 59% of instructors that use tools to create or that contain interactive visualizations have used SV tools, whereas for all other practices, 25% have used SV tools. This suggests that some instructors adopted SV in their classes.

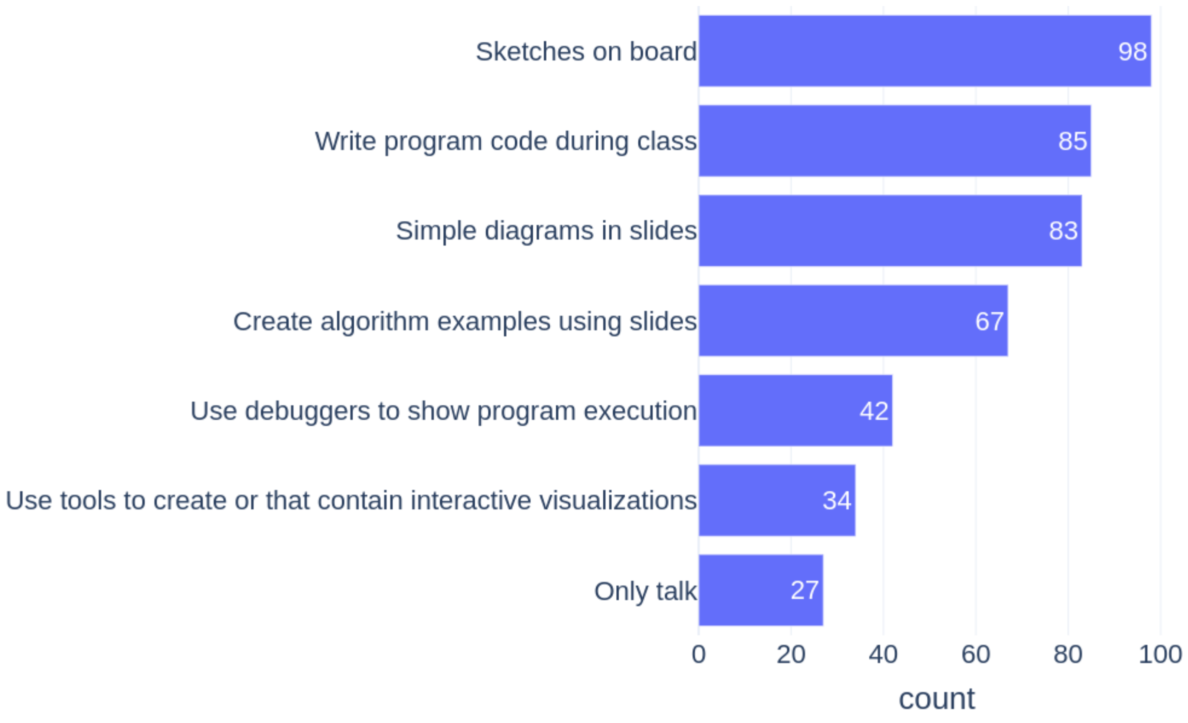


Figure 19 – Common teaching practices adopted by instructors.

Answering RQ1: According to the collected data, board sketches is the most common practice used by instructors, closely followed by writing code during class and using lecture slides to show diagrams and algorithm illustrations. Only 27.9% of instructors reported experimenting with SV tools, and the adoption of such tools in classes is even smaller.

5.1.6.2 Participants Perceptions Towards Willow

In the following sections of the form, the goal was to collect perceptions of the participants towards Willow, in the context of IP and DSA disciplines.

Figure 20 shows participants' perceptions toward videos containing examples of basic programming concepts, data structures and algorithms, created using Willow. These videos are described in Section 5.1.5. Figure 20 displays a diverging stacked bar chart quantifying and summarizing participants' perceptions over different Willow features illustrated by the videos. The size of each bar represents the number of participants who have chosen a particular value in a 5-point Likert scale, ranging from "totally disagree" to "totally agree" or "Very unsatisfactory" to "Very satisfactory" for a given question (e.g., "Regarding recursion, do you think your students can benefit from the features illustrated in the video?"). The three first rows in the chart refer to the perception of basic programming concepts common to IP [3]. The fourth and fifth rows refer to data structures and algorithms common to DSA, and the last row refers to participants opinions regarding if WILLOW could contribute to lecture material participants proposed previously for explaining tree rotations. The scatter traces are the average and median of participants' answers.

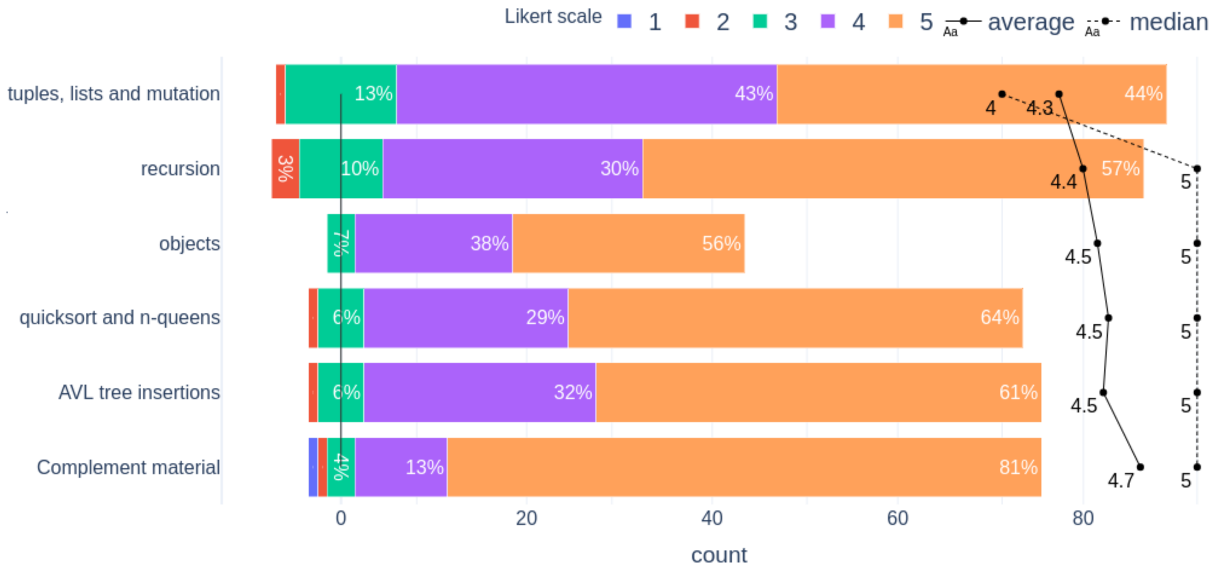


Figure 20 – Participants perception about WILLOW's visualizations.

Diverging stacked bar chart showing the perception of participants about WILLOW in a scale from 1 (negative perception) to 5 (positive perception). The average and median are represented by the scatter traces on the right side.

Figure 20 shows that participants had an overall good acceptance on all of WILLOW's visualizations, and agreed with the possible applicability of WILLOW in the teaching process.

In general participants mostly agreed with the following benefits of using WILLOW:

- *Good representation of programming concepts*: Participants liked WILLOW's visualizations and agreed that the visualization elements can provide good visual abstractions to explain programming concepts and inner workings of algorithms to students.

- *Fast way to create examples*: Many participants reported that WILLOW may help by making the creation of examples faster. They would not have to create examples in lecture slides or draw sketches, which would save class time. Some participants also said it would be very useful for algorithms that require drawing of too many steps, such as sorting algorithms. Other participants also cited the possibility of

[3] The number of answers in the "objects" entry is smaller due to a problem in the execution phase, part of the responses had to be discarded.

making small modifications in the algorithms without having to redraw illustrations again.

- *Benefits to students*: Another common received feedback was the benefits for students, such as being able to explore the content in an interactive manner, and that Willow could be used as an "extra class" resource, where students could use it to study alone by following visualizations of algorithm executions, and go beyond the explanations given by their instructions.

A few participants did not appreciate some of Willow's visualizations. The most common complaints were related to:

- *Complicated visualizations*: Participants reported that some visualizations were not intuitive due to several causes. For example, they complained that sometimes too much information could be displayed at once, and that having multiple focus points may confuse beginners. Moreover, they also mention pointer "pollution", when multiple variables point to the same object. Willow already applies filtering of references, this feature can be made customizable to filter more references and reduce this problem.

- *Indistinguishable objects*: Some objects rendered in the heap are very similar, such as tuples and lists, which are distinguishable only by the type name over the object representation. Different representations of objects already exist in Willow, these representations could be used to differentiate objects, new representations can also be implemented if necessary.

- *Adherence of visualizations to a single programming language*: Instructors complained that visualizations may be tied to a single programming language, which could compromise students understanding of programming concepts. Some participants also reported that they would not use Willow because it does not support C/C++. Because the visualizations used the Python language, some participants thought it was the only supported language. It is true that Willow currently does not support C/C++, but visualizations are not restrained to a single language. Both Python and Java can be used, and since Willow has an extensible language support, C/C++ and other languages might be added. Although further investigation on how to map language elements in visualization elements is necessary.

Answering RQ2: Results indicate that the majority of participants (91.3%) had positive impressions about Willow. In particular, 5 participants manifested strong interest in applying Willow in their next classes, even though Willow is still not mature.

5.1.7 Threats to Validity

Some of the common validity criteria used in quantitative studies, such as internal validity, reliability, and objectivity are not suitable for qualitative studies. The main threats to validity in qualitative studies are related to trustworthiness, which can be translated in the question "Can the findings of the study be trusted?". Based on that, we adopted the following validity criteria: credibility, transferability, and confirmability (KORSTJENS; MOSER, 2018)

The following describes each criteria and our actions to accomplish them:

- *Credibility*: The confidence that research findings are correctly inferred from the collected data. To ensure credibility, the development of the questionnaire and other research materials was conducted by two researchers, the questionnaire completeness was corroborated with pilot trials with two instructors, and although this is a qualitative study, a large part of the data collected from questions was already quantified and categorized through the use Likert scales and predefined categories.
- *Transferability*: The degree to which the results of our survey can be transferred to other contexts or settings with other respondents. All materials necessary to carry the study —questionnaire content, videos and WILLOW itself— are publicly available. We collected responses from a large sample of our population, which are IP and DSA instructors, totalizing 104 participants of several universities.
- *Confirmability*: The degree to which the findings of the study could be confirmed by other researchers. All the findings we obtained were derived directly from the questionnaire answers, a large part of our conclusions were derived from the quantitative data we collected, while using participants provided rationale to further understand their decisions. Although, we did not had authorization to publish participants responses. Finally, we also described in detail our research design, processes and taken decisions throughout this document.

5.2 EXPERIMENT

This section reports the results of an experiment comparing the performance of developers who used WILLOW and traditional textual tools to solve simple data structure and algorithms problems.

5.2.1 Objective

According to the Goal/Question/Metric (GQM) template, our research goal (BASILI; ROMBACH, 1988) is as follows:

Analyze developers as experimental subjects, **for the purpose of** comparing the performance when using WILLOW against textual tools (IDEs or text editors of preference), **with respect to** time to solve the problem, confidence of the developer in their answer, and the correctness of the answer, **from the point of view of** amateur software developers, **in the context of** an experiment involving developers hired from an online outsourcing platform platform and involving online tools.

5.2.2 Research Question and Hypothesis

We translated our goal in the following research question, and its associated hypothesis:

- RQ3: What are the benefits WILLOW can offer to programmers when solving data structures and algorithms problems, with regard to time to solve, confidence and correctness?

In the definition of the objective (Section 5.2.1), we manifested that we want to compare participants in two situations, which are the use of WILLOW and the use of traditional textual tools. The participant performance can be translated in the metrics described in the objective. Based on that, and since we are looking for any benefits WILLOW can provide to users, the hypothesis for the RQ3 is stated as:

H_0: There is no benefit in using WILLOW *as an assisting tool with regard to time to solve, confidence and correctness.*

$$H_0 : \mu(TIME)_{\text{textual}} \leq \mu(TIME)_{\text{willow}} \wedge$$
$$\mu(CONFIDENCE)_{\text{textual}} \geq \mu(CONFIDENCE)_{\text{willow}} \wedge$$
$$\mu(CORRECTNESS)_{\text{textual}} \geq \mu(CORRECTNESS)_{\text{willow}}$$

H_a: There is benefit in using WILLOW *as an assisting tool with regard to time to solve, confidence and correctness.*

$$H_{\text{a}} : \mu(TIME)_{\text{textual}} > \mu(TIME)_{\text{willow}} \vee$$
$$\mu(CONFIDENCE)_{\text{textual}} < \mu(CONFIDENCE)_{\text{willow}} \vee$$
$$\mu(CORRECTNESS)_{\text{textual}} < \mu(CORRECTNESS)_{\text{willow}}$$

- $\mu(TIME)_{\text{textual}}$: The time taken to answer the problem using textual tools.

- $\mu(TIME)_{\text{willow}}$: The time taken to answer the problem using Willow.
- $\mu(CONFIDENCE)_{\text{textual}}$: The confidence of the participant in their solution using textual tools.
- $\mu(CONFIDENCE)_{\text{willow}}$: The confidence of the participant in their solution using Willow.
- $\mu(CORRECTNESS)_{\text{textual}}$: The correctness of the participant's solution using textual tools.
- $\mu(CORRECTNESS)_{\text{willow}}$: The correctness of the participant's solution using Willow.

5.2.3 Variables

5.2.3.1 Independent Variables

The only independent variable (factor) of this experiment is the tool to be used. This independent variable contains two levels:

1. participant must use a textual environment (text editor, IDE, etc.) to solve problems.
2. participant must use Willow to solve problems.

This variable is manipulated by asking participants to use Willow in the middle of the experiment. More about level switching is explained in the Sections 5.2.7 and 5.2.8. Since the tool to be used is the only factor of the experiment, the levels of this variable can be interpreted directly as the only treatments of the experiment.

5.2.3.2 Dependent Variables

The dependent variables are the time to understand and solve problems *(TIME)*, participants confidence in their solutions *(CONFIDENCE)* and the correctness of the solution *(CORRECTNESS)*.

TIME is collected when the participant submits a solution, the elapsed time is automatically by the tool used to conduct the experiment. *CONFIDENCE* is provided by the participant after solving each problem, participants choose the confidence on their solution through a 5-point Likert scale asking how confident they have in the provided solution. *CORRECTNESS* is extracted from participants' source code, by executing it with test inputs. The correctness of each answer was evaluated against 6 test cases which are available in Appendix Section 6.2.

5.2.4 The Choice of Dependent Variables

Since the purpose of visualization tools is to provide visual aid through the creation of illustrations, we expected that users would understand the behavior of programs' abstract constructs more easily and clearly with the help of illustrations. For this reason, we decided to collect the time the participant takes to solve a problem and the correctness of the solution. The confidence was collected to complement the other variables, the purpose of this metric was to verify if Willow had any impact in the participants' own understanding of their solutions.

5.2.5 Experimental Subjects

The population is composed of developers with "basic knowledge" of the Python programming language and data structures and algorithms, regardless of the their professional experience or formation. Due to the COVID-19 outbreak, universities were closed and this made hard to recruit students to participate in the experiment. This way, we used the Upwork platform (UPWORK, 2020). We published our experiment as a job offer in the platform and received proposals from freelancers all over the world. Each proposal was filtered according to the population constraints, and we then hired each freelancer and sent all information necessary to run the experiment. We received 34 proposals from freelancers, and 18 participants were selected and participated in our experiment.

The participant selection was complex due to our population constraints. In our conception, "basic knowledge" means that participants should have the intuition on how to solve problems, but they should not have much experience or be experts in the topics. Participants also could not be too amateurs, to the point of not even understanding the problems, or having no idea how to solve them.

Another limitation on participant selection was the financial, due to the limited budget, we could only hire 18 freelancers as participants.

5.2.5.1 Ethical Concerns

Due to the nature of the experiment, we did not notice any possible ethical concerns that could lead to problems, except for privacy concerns. We decided that all participant information would be private, and clearly stated this in the experiment introduction. Although, since all participants were paid to participate in the experiment, we did not allow participants to request withdrawal from the experiment and consequently discard their responses.

5.2.6 Objects and Tasks

Four objects (data structures and algorithms problems) are used during the experiment. All problems are fairly simple to solve, but also allow different solutions with varying difficulty degrees. The problems are:

1. *Cycle Detection*: Participants have to implement a solution to find if a linked list contains a cycle;
2. *Jessie Cookies*: Participants have to consecutively combine the smallest elements in an array and create a new one until the smallest element surpasses a threshold;
3. *Lowest Common Ancestor*: Participants need to find the lowest common ancestor of two nodes in a binary search tree;
4. *Reverse List*: Participants have to reverse a singly linked list.

Each problem contains a basic description of the objective, a program template, which reads inputs and prints outputs, and some examples. Participants are supposed to implement only one function, which is described in the objective and with the signature already in the template. Problems (1) and (2) compose the first task and problems (3) and (4) compose the second task.

Appendix Section 6.2 contains the form used to present the questions during the experiment. This form contains the description of all objects of the experiment.

5.2.7 Experimental Design

Our experiment is based on programming exercises, where participants have to solve algorithms and data structures problems. We chose this kind of experiment because our goal is to objectively evaluate and classify participants' solutions using test cases. This helped us to avoid the subjective part of the participants' opinions on the tool, which has already been evaluated already in the survey. The only inherently subjective element in this study is the participant's confidence in the provided answers.

We adopted a single-factor *latin-square* (WOHLIN et al., 2012) experimental design with with two treatments. The adopted design is illustrated by Table 2. The *latin-square* design was chosen to avoid *within-subjects* design drawbacks in this specific experiment, i.e., to reduce impact on the performance of participants when solving problems they already solved using another treatment. In this design, participants are divided into groups, and each group solves a set of tasks using different treatments (Section 5.2.6). Treatments correspond to the tool used by the participant to solve a task (Section 5.2.3).

Table 2 – Latin-square design, a complete 2x2 layout.

	Treatment 1	Treatment 2
Participant group 1	Task 1	Task 2
Participant group 2	Task 2	Task 1

An important property to ensure validity of *latin-square* designs is randomizing the participants' groups. In controlled experiments where all participants are present at the same time, this can be done *a priori*, before the experiment starts. To conduct this experiment, however, we used the Upwork freelancing platform (UPWORK, 2020) for selecting participants. Therefore, we did not had access to all participants promptly so they could not be randomly distributed into groups prior to the experiment. To mitigate and solve this problem, we created a form engine, dubbed JSON-FORM[4], designed to apply simple factorial designs in an online setting. With JSON-FORM, when new participants access the form, they are randomly assigned to a group, automatically, and the specific order for tasks and treatments of the chosen group is then applied. JSON-FORM also solved other issues that we had experienced when using common form engines (e.g., Google Forms), such as collecting the time that participants take to solve problems. Most importantly, it enables us to present formatted text, which is a prominent need in an experiment such as ours, where we need to show source code when describing the input and output of the presented problems. With the help of JSON-FORM, participants could join the experiment at any moment, without researcher supervision.

5.2.8 Protocol

The process started with hiring amateur freelancers. After posting a job calling for participants, we filtered participants questions about data structures and algorithms, removing candidates with too much or too little experience (the former would not benefit from the tool due to very solid mental models, the latter would not be capable of answering questions due to lack of knowledge). After being accepted, the participant received instructions and access to the form created with JSON-FORM (the form is available in the Appendix Section 6.2). Participants could answer the form at any moment until their due date.

When accessing the form, participants are presented with information about the study, such as researchers contact, study objective, privacy concerns, procedures and expected time to answer the form (which is approximately 1 hour based on 5 pilots). The form presents tasks in different orders based on the group which was randomly chosen for the

[4] `https://github.com/pedro00dk/json-form`

participant. After the introduction section of the form, the following sections present two problems to be solved using any textual tool (e.g., IDEs or text editors) the participant feel most comfortable.

After the first two questions, Willow is presented to the participants and a small tutorial section must be completed before proceeding to the next problems. The tutorial is composed of four steps, which teach participants how to execute code on Willow and have basic interactions with the visualization. After the tutorial, two more problems are presented, which have to be solved using Willow.

Participants were prohibited of accessing search engines and look for solutions, although, search for documentation was allowed. Participants also had to record their screen during the entire process, the recording was used to verify if participants respected the restrictions (Section 5.2.10) and followed the tutorial and other procedures.

5.2.9 Results

This section discusses the data we collected from the experiment. From the 18 participants, 6 were graduated developers, 3 were developers without a major degree, and 9 were undergraduate students. Since our hypothesis is composed of sub-components, we first present the data and hypothesis tests, and then discuss about them.

Figure 21 shows participants confidence in the solutions they submitted when using textual environments and Willow, each bar represents the number of participants that manifested their confidence in a 5-point Likert scale.

As shown in Figure 21, the confidence of participants participants clearly does not follow the Poisson distribution. We applied the Brunner-Munzel test (BRUNNER; MUNZEL, 2000), which is a non-parametric test similar to the Wilcoxon-Mann-Whitney U test (also known as generalized Wilcoxon test), but does not assume similar variance of the samples being tested. Our tests yielded a p-value of 0.49 in a one-sided test, which does not reject the confidence term in from the null hypothesis:

$$\mu(CONFIDENCE)_{\text{textual}} \geq \mu(CONFIDENCE)_{\text{willow}}$$

Figure 22 shows the time in minutes participants took to solve the programming problems using their preferred textual environment and Willow. Each point represents one of the samples of a participant, and the violin plot on the side displays the distribution of the data.

The distribution of the time participants took to solve problems does not follow the normal distribution. We applied Brunner-Munzel test to verify if there were differences between the samples. We obtained a p-value of 0.59 in an one-sided test, which does not reject the time term from the null hypothesis:

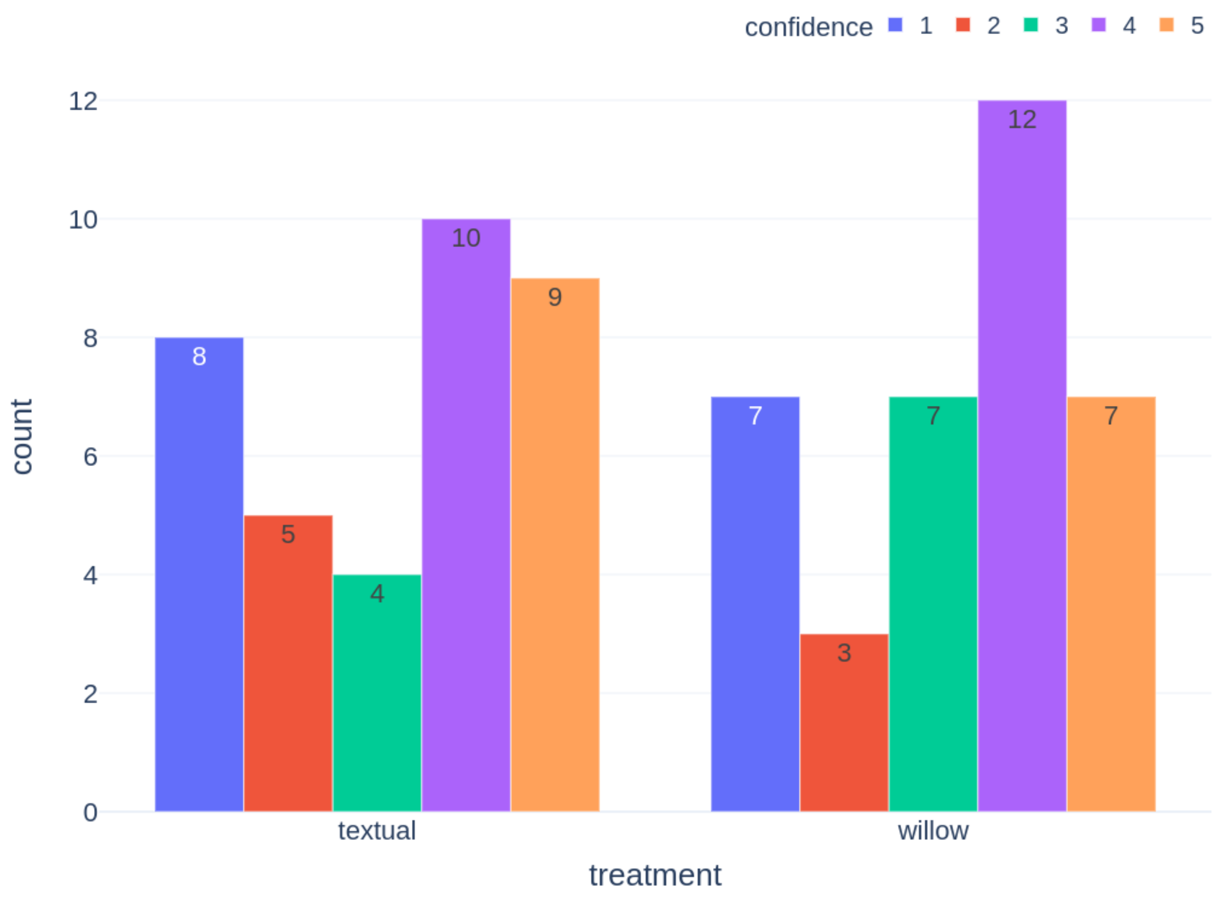


Figure 21 – Participants confidence in their solutions.

$$\mu(TIME)_{\text{textual}} \leq \mu(TIME)_{\text{willow}}$$

Figure 23 shows the statistics of the solutions correctness when using textual tools and Willow. Each trace represents the average (colored traces) or median (gray traces) for each question. Section 5.2.6 describes each question. The purple trace and dotted trace with squares show the average and median of solutions written with textual tools. The green trace and dotted trace with circles shows the average and median of solutions written with Willow.

The distribution of the solutions correctness does not follow the normal distribution since the average and median are far apart. We applied Brunner-Munzel test again to verify if there were differences between the samples. We obtained a p-value of 0.38 in an one-sided test, which does not reject the correctness term from the null hypothesis:

$$\mu(CORRECTNESS)_{\text{textual}} \leq \mu(CORRECTNESS)_{\text{willow}}$$

We also tested our hypothesis with all subsets of questions. For the time and confidence variables, we did not find any marginally better result. However, some subsets of questions presented much better results. Especially, *q1* and *q2* individually yielded a p-value of

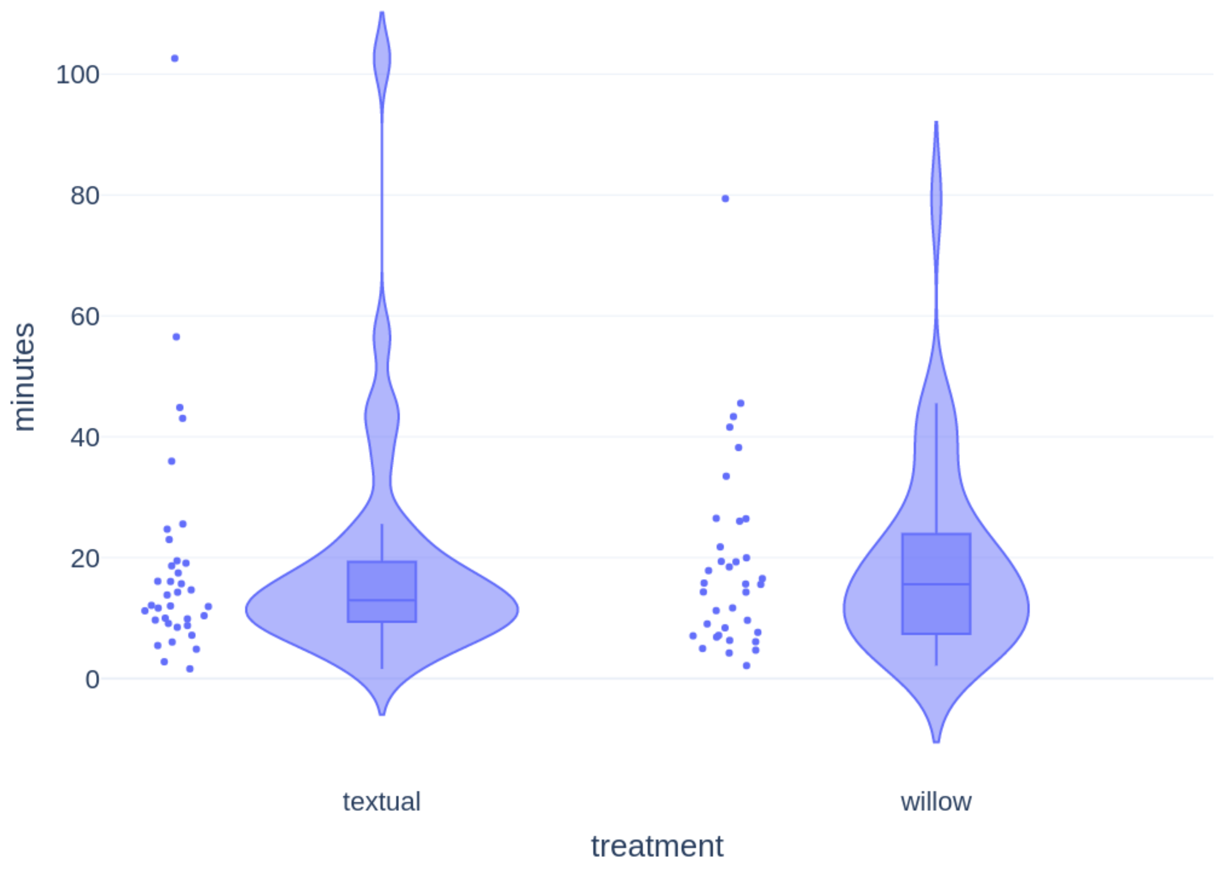


Figure 22 – Time participants take to solve problems.

0.14 and 0.11 individually. When combining the solutions for *q1* and *q2*, we obtained a p-value of 0.054. Still, the hypothesis could not be rejected at a significance level of 5%. Discarding the answers of *q3* and *q4* and increasing the significance level might increase type II error, which would be favorable to our conclusion. Thus we prefer to stick with a significance level of 5% in this experiment.

Answering RQ3: Since we could not reject the null hypothesis, the evidence collected in this study does not allow us to claim that WILLOW brings the benefits we analysed to programmers with regard to time to solve, confidence and correctness.

Since the survey with instructors presented very positive results, we expected good results in this follow-up experiment, even though the context and populations are different. We suspect that our main problems were the participant selection and the WILLOW's tutorial.

Due to the population constraints related to participant's knowledge and experience, and the online setting in which the experiment occurred, it was hard to classify and select

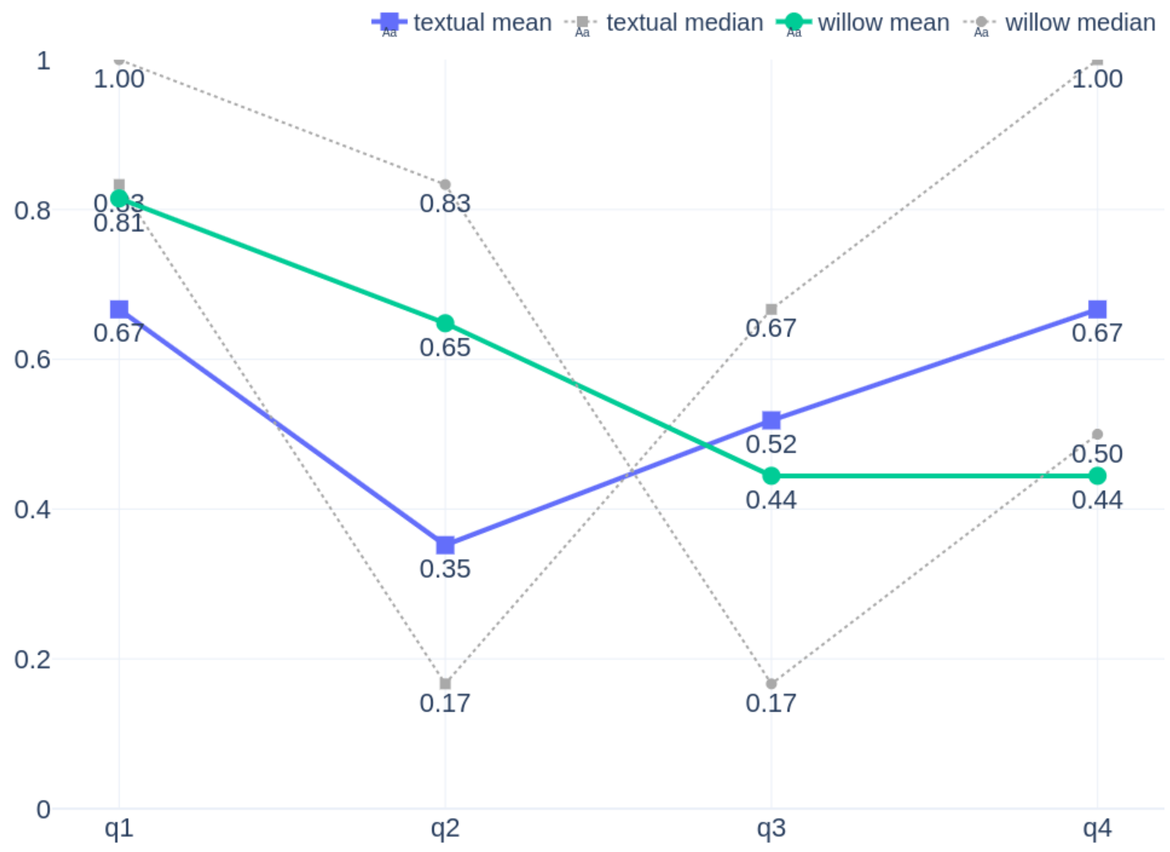


Figure 23 – Participants' scores average and medians for each question and treatment.

participants to the experiment. Some participants were much more experienced than the population allows and others were too immature. For instance, 3 of the 18 participants were not able to solve a single problem.

Willow's tutorial was too short for the experiment, but because of financial restrictions we could not make the experiment any longer. Willow's tutorial can be viewed in Appendix Section 6.2.

5.2.10 Threats to Validity and Control Actions

During the planning phase of this experiment, we focused mainly in how to handle threats that could occur in an online setting. Since the experiment happened in an asynchronous manner (experimenter did not watch participants resolution), we decided to focus in preventive and fix-up control actions. Table 3 shows a list with threats to validity and their respective control actions.

Table 3 – Threats to validity and control actions.

Threats to Validity	Threat Description	Control Actions
Incomplete data collection (internal)	Participants may submit incomplete answers.	⋆ Clearly state in the data collection tool how to fill the data. ⋆ Verify if the failure is caused by the submission method.
Interaction between treatments (construct)	When a treatment influences positively or negatively the results of the next treatments.	⋆ The *latin-square* design is used to prevent participants from solving the same questions with different treatments.
Insufficient Training (internal)	When the training process does not provide all necessary information to the participants, or does not give enough time for practicing.	
Familiarity with the experimental material (internal)	When the participant has very little or no knowledge of the experiment material.	⋆ Use simple problems that allow multiple different solutions.
Heterogeneous Sample (conclusion)	When there are differences among the participants related to expertise and experience.	⋆ Use an experimental design that minimizes the impact of heterogeneous samples (*latin-square*). ⋆ Assign participants to groups randomly.
Fatigue can affect the participant's performance (internal)	If the experiment is too long, participants may feel tired or bored.	⋆ Use problems that can be resolved more quickly. ⋆ Add breaks between the objects and tasks.

Application of inappropriate statistical test (conclusion)	There are factors that must be met in order for a statistical test to be used, such as sample size and data normality.	⋆ Verify is the sample data fits all constraints of the statistical test. ⋆ Verify the adoption of the statistical test in other experiments.
Precision on data collection (internal)	The data may not be precise if the participants are responsible for collecting it.	⋆ Use a tool to do all data collection.
Abandonment (internal)	Participants may abandon the experiment (internal)	⋆ Discard participant's incomplete data.
Interruption (internal)	Participant may have to stop executing the experiment by any reason.	⋆ Add breaks between the objects and tasks.
Ignore treatments (construct)	Participant may ignore the treatments when solving the tasks.	⋆ Disqualify participant and discard their data.
Search for solutions (construct)	Participants may search for solutions of the problems online.	⋆ Disqualify participant and discard their data.

6 FINAL CONSIDERATIONS

This chapter presents the conclusions we drawn from the development and evaluation of Willow. We then describe our expectations of future work involving the tool.

6.1 CONCLUSION

This work presents Willow, a tool to support Data Structure and Algorithms lessons through the creation of interactive program visualizations that can be used by instructors and students in the teaching-learning process. Notable features are the ability to change the representation of the program data, create animations and navigate to any point of a program, specialized nodes to show array data or select specific fields of an object allow the user to better control which and how objects are shown in order to make programs easier to understand. This allows the creation of visualizations for Introductory Programming concepts, and Data Structures and Algorithms.

The first conducted study with instructors revealed that Willow can be promising if used as an aid tool for the creation of algorithm visualization in both Introductory Programming and Data Structures and Algorithms courses. Instructors also agreed that Willow could be beneficial to students if used directly by them. In our follow-up experiment, however, we were not able to verify such statement, even though we had such positive results in the first study.

6.2 FUTURE WORK

Future work includes execution of another experiment similar to the one executed with freelancers, this time with students from IT courses. We also plan to continue the development of Willow, bringing new features, such as better data structure detection, representations for complex data structures like graphs, new ways to visualize objects, general visualization improvements and support for new programming languages. Finally, we want to publish our results to present Willow and our studies to the scientific community.

REFERENCES

ALTADMRI, Amjad; BROWN, Neil CC. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In: PROCEEDINGS of the 46th ACM Technical Symposium on Computer Science Education. [S.l.: s.n.], 2015. P. 522–527.

ANDERSON, John R; JEFFRIES, Robin. Novice LISP errors: Undetected losses of information from working memory. **Human–Computer Interaction**, Taylor & Francis, v. 1, n. 2, p. 107–131, 1985.

ATEEQ, Muhammad et al. C++ or Python? Which One to Begin with: A Learner's Perspective. In: IEEE. 2014 International Conference on Teaching and Learning in Computing and Engineering. [S.l.: s.n.], 2014. P. 64–69.

BARBOSA, Marcelo RG et al. Implementação de compilador e ambiente de programação icônica para a linguagem logo em um ambiente de robótica pedagógica de baixo custo. In: 1. BRAZILIAN Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE). [S.l.: s.n.], 2009. v. 1.

BASILI, Victor R; ROMBACH, H Dieter. The TAME project: Towards improvement-oriented software environments. **IEEE Transactions on software engineering**, IEEE, v. 14, n. 6, p. 758–773, 1988.

BAU, David et al. Learnable programming: blocks and beyond. **Communications of the ACM**, ACM New York, NY, USA, v. 60, n. 6, p. 72–80, 2017.

BAYMAN, Piraye; MAYER, Richard E. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. **Communications of the ACM**, ACM New York, NY, USA, v. 26, n. 9, p. 677–679, 1983.

BECKER, William E; WATTS, Michael. Teaching economics at the start of the 21st century: Still chalk-and-talk. **American Economic Review**, v. 91, n. 2, p. 446–451, 2001.

BLENDER, the free and open source 3D creation suite. [S.l.: s.n.]. `https://www.blender.org/`. Accessed: 2020-06-06.

BERKELEY University CS 61A: Structure and Interpretation of Computer Programs. [S.l.: s.n.]. `https://cs61a.org/`. Accessed: 2020-06-10.

BRESSON, Jean; AGON, Carlos; ASSAYAG, Gérard. OpenMusic: visual programming environment for music composition, analysis and research. In: PROCEEDINGS of the 19th ACM international conference on Multimedia. [S.l.: s.n.], 2011. P. 743–746.

BRUNNER, Edgar; MUNZEL, Ullrich. The nonparametric Behrens-Fisher problem: asymptotic theory and a small-sample approximation. **Biometrical Journal: Journal of Mathematical Methods in Biosciences**, Wiley Online Library, v. 42, n. 1, p. 17–25, 2000.

BURNETT, Margaret M; ATWOOD, John Wesley; WELCH, Zachary T. Implementing level 4 liveness in declarative visual programming languages. In: IEEE. PROCEEDINGS. 1998 IEEE Symposium on Visual Languages (Cat. No. 98TB100254). [S.l.: s.n.], 1998. P. 126–133.

CHRISTIAWAN, Lucky; KARNALIM, Oscar. AP-ASD1: An Indonesian Desktop-based Educational Tool for Basic Data Structure Course. **Jurnal Teknik Informatika dan Sistem Informasi**, v. 2, n. 1, 2016.

CLANCY, Michael. Misconceptions and attitudes that interfere with learning to program. **Computer science education research**, Taylor and Francis Group, London, p. 85–100, 2004.

DANIELSIEK, Holger; PAUL, Wolfgang; VAHRENHOLD, Jan. Detecting and understanding students' misconceptions related to algorithms and data structures. In: PROCEEDINGS of the 43rd ACM technical symposium on Computer Science Education. [S.l.: s.n.], 2012. P. 21–26.

DIEN, Habibie Ed; ASNAR, Yudistira Dwi Wardhana. OPT+ Graph: Detection of Graph Data Structure on Program Visualization Tool to Support Learning. In: IEEE. 2018 5th International Conference on Data and Software Engineering (ICoDSE). [S.l.: s.n.], 2018. P. 1–6.

DIXIT, Rashmi K; YALAGI, Pratibha S. Visualization based intelligent tutor system to improve study of Computer Algorithms. **Computer**, v. 1, n. 2, p. 1, 2017.

DOUKAKIS, Dimitrios; GRIGORIADOU, Maria; TSAGANOU, Grammatiki. Understanding the programming variable concept with animated interactive analogies. In: PROCEEDINGS of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA'07). [S.l.: s.n.], 2007.

DU BOULAY, Benedict. Some difficulties of learning to program. **Journal of Educational Computing Research**, SAGE Publications Sage CA: Los Angeles, CA, v. 2, n. 1, p. 57–73, 1986.

EBRAHIMI, Alireza. Novice programmer errors: Language constructs and plan composition. **International Journal of Human-Computer Studies**, Elsevier, v. 41, n. 4, p. 457–480, 1994.

EGAN, Matthew Heinsen; MCDONALD, Chris. Program visualization and explanation for novice C programmers. In: ACE. [S.l.: s.n.], 2014. P. 51–57.

FITZGERALD, Sue et al. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. **Computer Science Education**, Taylor & Francis, v. 18, n. 2, p. 93–116, 2008.

FOUH, Eric; AKBAR, Monika; SHAFFER, Clifford A. The role of visualization in computer science education. **Computers in the Schools**, Taylor & Francis, v. 29, n. 1-2, p. 95–117, 2012.

GARNER, Sandy; HADEN, Patricia; ROBINS, Anthony. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In: PROCEEDINGS of the 7th Australasian conference on Computing education-Volume 42. [S.l.: s.n.], 2005. P. 173–180.

GÖTSCHI, Tina; SANDERS, Ian; GALPIN, Vashti. Mental models of recursion. In: PROCEEDINGS of the 34th SIGCSE technical symposium on Computer science education. [S.l.: s.n.], 2003. P. 346–350.

GUO, Philip J. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In: PROCEEDINGS of the 28th Annual ACM Symposium on User Interface Software & Technology. [S.l.: s.n.], 2015. P. 599–608.

________. Online python tutor: embeddable web-based program visualization for cs education. In: ACM. PROCEEDING of the 44th ACM technical symposium on Computer science education. [S.l.: s.n.], 2013. P. 579–584.

GUO, Philip J; WHITE, Jeffery; ZANELATTO, Renan. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In: IEEE. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). [S.l.: s.n.], 2015. P. 79–87.

GUZDIAL, Mark. Centralized mindset: A student problem with object-oriented programming. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 27, n. 1, p. 182–185, 1995.

HERTZ, Matthew; JUMP, Maria. Trace-based teaching in early programming courses. In: PROCEEDING of the 44th ACM technical symposium on Computer science education. [S.l.: s.n.], 2013. P. 561–566.

HOLLIDAY, Mark A; LUGINBUHL, David. CS1 assessment using memory diagrams. In: PROCEEDINGS of the 35th SIGCSE technical symposium on Computer science education. [S.l.: s.n.], 2004. P. 200–204.

HUNDHAUSEN, Christopher D; DOUGLAS, Sarah A; STASKO, John T. A meta-study of algorithm visualization effectiveness. **Journal of Visual Languages & Computing**, Elsevier, v. 13, n. 3, p. 259–290, 2002.

JACKSON, James; COBB, Michael; CARVER, Curtis. Identifying top Java errors for novice programmers. In: IEEE. PROCEEDINGS frontiers in education 35th annual conference. [S.l.: s.n.], 2005. t4c–t4c.

KACZMARCZYK, Lisa C et al. Identifying student misconceptions of programming. In: PROCEEDINGS of the 41st ACM technical symposium on Computer science education. [S.l.: s.n.], 2010. P. 107–111.

KANON, a live programming environment specialized for data structure programming. [S.l.: s.n.]. `https://prg-titech.github.io/Kanon/`. Accessed: 2020-06-15.

KANG, Hyeonsu; GUO, Philip J. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In: PROCEEDINGS of the 30th Annual ACM Symposium on User Interface Software and Technology. [S.l.: s.n.], 2017. P. 737–745.

KARPIERZ, Kuba; WOLFMAN, Steven A. Misconceptions and concept inventory questions for binary search trees and hash tables. In: PROCEEDINGS of the 45th ACM technical symposium on Computer science education. [S.l.: s.n.], 2014. P. 109–114.

KELLEHER, Caitlin; PAUSCH, Randy. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 37, n. 2, p. 83–137, 2005.

KORSTJENS, Irene; MOSER, Albine. Series: Practical guidance to qualitative research. Part 4: Trustworthiness and publishing. **European Journal of General Practice**, Taylor & Francis, v. 24, n. 1, p. 120–124, 2018.

KRAMER, Jan-Peter et al. How live coding affects developers' coding behavior. In: IEEE. 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). [S.l.: s.n.], 2014. P. 5–8.

LAMBERT, Vickie A; LAMBERT, Clinton E. Qualitative descriptive research: An acceptable design. **Pacific Rim International Journal of Nursing Research**, v. 16, n. 4, p. 255–256, 2012.

LAURSON, Mikael; KUUSKANKARE, Mika; NORILO, Vesa. An overview of PWGL, a visual programming environment for music. **Computer Music Journal**, MIT Press, v. 33, n. 1, p. 19–31, 2009.

LEVIN, Kate Ann. Study design III: Cross-sectional studies. **Evidence-based dentistry**, Nature Publishing Group, v. 7, n. 1, p. 24–25, 2006.

LOPEZ, Mike et al. Relationships between reading, tracing and writing skills in introductory programming. In: PROCEEDINGS OF THE FOURTH INTERNATIONAL WORKSHOP ON COMPUTING EDUCATION RESEARCH. [S.l.: s.n.], 2008. P. 101–112.

MALETIC, Jonathan I; MARCUS, Andrian; COLLARD, Michael L. A task oriented view of software visualization. In: IEEE. PROCEEDINGS First International Workshop on Visualizing Software for Understanding and Analysis. [S.l.: s.n.], 2002. P. 32–40.

MALONEY, John et al. The scratch programming language and environment. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 10, n. 4, p. 1–15, 2010.

MCCAULEY, Renee et al. Debugging: a review of the literature from an educational perspective. **Computer Science Education**, Taylor & Francis, v. 18, n. 2, p. 67–92, 2008.

MIT Open Courseware, introductory programming courses. [S.l.: s.n.]. `https://ocw.mit.edu/courses/intro-programming/`. Accessed: 2020-06-22.

MORENO, Andrés et al. Visualizing programs with Jeliot 3. In: PROCEEDINGS of the working conference on Advanced visual interfaces. [S.l.: s.n.], 2004. P. 373–376.

MULLER, Orna. Pattern oriented instruction and the enhancement of analogical reasoning. In: PROCEEDINGS of the first international workshop on Computing education research. [S.l.: s.n.], 2005. P. 57–67.

NAPS, Thomas L; RÖSSLING, Guido, et al. Exploring the role of visualization and engagement in computer science education. In: WORKING group reports from ITiCSE on Innovation and technology in computer science education. [S.l.: s.n.], 2002. P. 131–152.

NAPS, Thomas; COOPER, Stephen, et al. Evaluating the educational impact of visualization. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 35, n. 4, p. 124–136, 2003.

OPENDSA, build your knowledge of Data Structures through visualizations and practice! [S.l.: s.n.]. `https://opendsa-server.cs.vt.edu/`. Accessed: 2020-06-06.

OKA, Akio; MASUHARA, Hidehiko; AOTANI, Tomoyuki. Live, synchronized, and mental map preserving visualization for data structure programming. In: PROCEEDINGS of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. [S.l.: s.n.], 2018. P. 72–87.

OKA, Akio; MASUHARA, Hidehiko; IMAI, Tomoki, et al. Live Data Structure Programming. In: COMPANION to the first International Conference on the Art, Science and Engineering of Programming. [S.l.: s.n.], 2017. P. 1–7.

ORSEGA, Michael C; VANDER ZANDEN, Bradley T; SKINNER, Christopher H. Experiments with algorithm visualization tool development. In: PROCEEDINGS of the 43rd ACM technical symposium on Computer Science Education. [S.l.: s.n.], 2012. P. 559–564.

OU, Jibin; VECHEV, Martin; HILLIGES, Otmar. An interactive system for data structure development. In: PROCEEDINGS of the 33rd Annual ACM Conference on Human Factors in Computing Systems. [S.l.: s.n.], 2015. P. 3053–3062.

PAUL, Wolfgang; VAHRENHOLD, Jan. Hunting high and low: instruments to detect misconceptions related to algorithms and data structures. In: PROCEEDING of the 44th ACM technical symposium on Computer science education. [S.l.: s.n.], 2013. P. 29–34.

PRICE, Blaine A; BAECKER, Ronald M; SMALL, Ian S. A principled taxonomy of software visualization. **Journal of Visual Languages & Computing**, Elsevier, v. 4, n. 3, p. 211–266, 1993.

PYTHON Tutor development history. [S.l.: s.n.]. `https://github.com/pgbovine/OnlinePythonTutor/blob/master/history.txt`. Accessed: 2020-06-10.

QIAN, Yizhou; HAMBRUSCH, Susanne, et al. Teachers' Perceptions of Student Misconceptions in Introductory Programming. **Journal of Educational Computing Research**, SAGE Publications Sage CA: Los Angeles, CA, v. 58, n. 2, p. 364–397, 2020.

QIAN, Yizhou; LEHMAN, James. Students' misconceptions and other difficulties in introductory programming: A literature review. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 18, n. 1, p. 1–24, 2017.

QIAN, Yizhou; LEHMAN, James D. Correlates of Success in Introductory Programming: A Study with Middle School Students. **Journal of Education and Learning**, ERIC, v. 5, n. 2, p. 73–83, 2016.

RAGONIS, Noa; BEN-ARI, Mordechai. A long-term investigation of the comprehension of OOP concepts by novices. **Computer Science Education**, Taylor & Francis, 2005.

RAMALINGAM, Vennila; LABELLE, Deborah; WIEDENBECK, Susan. Self-efficacy and mental models in learning to program. In: PROCEEDINGS of the 9th annual SIGCSE conference on Innovation and technology in computer science education. [S.l.: s.n.], 2004. P. 171–175.

ROBINS, Anthony; ROUNTREE, Janet; ROUNTREE, Nathan. Learning and teaching programming: A review and discussion. **Computer science education**, Taylor & Francis, v. 13, n. 2, p. 137–172, 2003.

ROMANOWSKA, Katarzyna et al. Towards Developing an Effective Algorithm Visualization Tool for Online Learning. In: IEEE. 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI). [S.l.: s.n.], 2018. P. 2011–2016.

ROQUE, Ricarose Vallarta. **OpenBlocks: an extendable framework for graphical block programming systems**. 2007. PhD thesis – Massachusetts Institute of Technology.

SAJANIEMI, Jorma; KUITTINEN, Marja. An experiment on using roles of variables in teaching introductory programming. **Computer Science Education**, Taylor & Francis, v. 15, n. 1, p. 59–82, 2005.

SAJANIEMI, Jorma; KUITTINEN, Marja; TIKANSALO, Taina. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. **Journal on Educational Resources in Computing (JERIC)**, ACM New York, NY, USA, v. 7, n. 4, p. 1–31, 2008.

SCRATCH, create stories, games and animations, share with others around the world. [S.l.: s.n.]. `https://scratch.mit.edu/`. Accessed: 2020-06-06.

SHAFFER, Clifford A et al. Algorithm visualization: The state of the field. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 10, n. 3, p. 1–22, 2010.

SILVA RIBEIRO, Romenig da; BRANDÃO, Leônidas de O; BRANDÃO, Anarosa AF. Uma visão do cenário Nacional do Ensino de Algoritmos e Programação: uma proposta baseada no Paradigma de Programção Visual. In: 1. BRAZILIAN Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE). [S.l.: s.n.], 2012. v. 23.

SORVA, Juha et al. **Visual program simulation in introductory programming education**. 2012. PhD thesis.

SORVA, Juha; KARAVIRTA, Ville; MALMI, Lauri. A review of generic program visualization systems for introductory programming education. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 13, n. 4, p. 1–64, 2013.

SORVA, Juha; LÖNNBERG, Jan; MALMI, Lauri. Students' ways of experiencing visual program simulation. **Computer Science Education**, Taylor & Francis, v. 23, n. 3, p. 207–238, 2013.

STOREY, M-AD; FRACCHIA, F David; MÜLLER, Hausi A. Cognitive design elements to support the construction of a mental model during software exploration. **Journal of Systems and Software**, Elsevier, v. 44, n. 3, p. 171–185, 1999.

SYKES, Edward R. Determining the effectiveness of the 3D Alice programming environment at the computer science I level. **Journal of Educational Computing Research**, SAGE Publications Sage CA: Los Angeles, CA, v. 36, n. 2, p. 223–244, 2007.

TAKATSUKA, Masahiro; GAHEGAN, Mark. GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization. **Computers & Geosciences**, Elsevier, v. 28, n. 10, p. 1131–1144, 2002.

TANIMOTO, Steven L. A perspective on the evolution of live programming. In: IEEE. 2013 1st International Workshop on Live Programming (LIVE). [S.l.: s.n.], 2013. P. 31–34.

UNITY Engine, a cross-platform game engine developed by Unity Technologies. [S.l.: s.n.]. `https://unity.com/`. Accessed: 2020-06-06.

UNREAL Engine, the world's most open and advanced real-time 3D creation tool. [S.l.: s.n.]. `https://www.unrealengine.com/en-US/`. Accessed: 2020-06-06.

UPWORK. [S.l.: s.n.], 2020. Available from: <`https://www.upwork.com/`>. Visited on: 1 Aug. 2020.

URQUIZA-FUENTES, Jaime; VELÁZQUEZ-ITURBIDE, J Ángel. A survey of successful evaluations of program visualization and algorithm animation systems. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 9, n. 2, p. 1–21, 2009.

UUHISLE, a Program Visualization Tool for Introductory Programming Education. [S.l.: s.n.]. `http://uuhistle.org/index.php`. Accessed: 2020-06-06.

VAINIO, Vesa; SAJANIEMI, Jorma. Factors in novice programmers' poor tracing skills. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 39, n. 3, p. 236–240, 2007.

VISUALGO, visualising data structures and algorithms through animation. [S.l.: s.n.]. `https://visualgo.net/`. Accessed: 2020-06-06.

VELÁZQUEZ-ITURBIDE, J Ángel. Students' Misconceptions of Optimization Algorithms. In: PROCEEDINGS of the 2019 ACM Conference on Innovation and Technology in Computer Science Education. [S.l.: s.n.], 2019. P. 464–470.

VRACHNOS, Euripides; JIMOYIANNIS, Athanassios. Design and evaluation of a web-based dynamic algorithm visualization environment for novices. **Procedia Computer Science**, Elsevier, v. 27, p. 229–239, 2014.

WEINTROP, David; WILENSKY, Uri. To block or not to block, that is the question: students' perceptions of blocks-based programming. In: PROCEEDINGS of the 14th international conference on interaction design and children. [S.l.: s.n.], 2015. P. 199–208.

WOHLIN, Claes et al. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.

YOUNG, Mark; ARGIRO, Danielle; KUBICA, Steven. Cantata: Visual programming environment for the Khoros system. **ACM SIGGRAPH Computer Graphics**, ACM New York, NY, USA, v. 29, n. 2, p. 22–24, 1995.

ZEHRA, Shamama et al. Student misconceptions of dynamic programming. In: PROCEEDINGS of the 49th ACM technical symposium on Computer Science Education. [S.l.: s.n.], 2018. P. 556–561.

ZINGARO, Daniel et al. Identifying student difficulties with basic data structures. In: PROCEEDINGS of the 2018 ACM Conference on International Computing Education Research. [S.l.: s.n.], 2018. P. 169–177.

APPENDIX A - SURVEY QUESTIONNAIRE

Opinions and interest in the Willow tool

# Researchers
Pedro Moraes -- phsm@cin.ufpe.br
Leopoldo Teixeira -- lmt@cin.ufpe.br
Marcelo d'Amorim -- damorim@cin.ufpe.br
Waldemar Neto -- waldemar.neto@gmail.com

# Participants
The participant must have experience as a professor teaching a CS-related course.

# Willow
Willow is a tool for creating program visualizations from source code, focusing on representing basic programming concepts, algorithms, and data structures.
It is available at https://willow-beta.web.app/
It is not required to use Willow before answering this form, short videos will be presented showing features while the participant responds to the survey.

# Objective
Analyze participants' interests and opinions about the Willow tool, for the purpose of comparing Willow with traditional teaching strategies used by participants, with respect to the ease of creating illustrations of basic programming concepts, algorithms, and data structures created to assist in the teaching, from the point of view of professors of CS-related courses.

# Duration
The estimated duration is 15 minutes.
All questions besides the demographic questionnaire below are optional.

# Privacy
The information obtained from your participation in this study will be kept strictly confidential. Any material will be referenced only by an identifier. For registration of the work, you must provide your name. Any results presented in future scientific publications will be anonymized
* Required

1. By proceeding, I declare that I have had sufficient time to read and understand the information contained in this form. The objectives have been explained, as well as what will be required of me as a participant. I am aware that I can give up participating in the survey at any time, and in doing so, request that my data should not be used for analysis and discarded. *

Check all that apply.

☐ I agree to participate

2. Your name *

3. Your teaching institution *

4. When was the last time you taught any of the courses below? *

IP: Introduction to Programming, Programming 101, or equivalent; DSA: Data structures and Algorithms or equivalent.

Mark only one oval per row.

	I am currently teaching it	Less than 1 year	From 1 to 2 years	From 2 to 5 years	More than 5 years	Never
IP	◯	◯	◯	◯	◯	◯
DSA	◯	◯	◯	◯	◯	◯

5. For how long have you taught these courses? *

IP: Introduction to Programming, Programming 101, or equivalent; DSA: Data structures and Algorithms or equivalent.

Mark only one oval per row.

	Never	Less than 1 year	Between 1 and 2 years	Between 2 and 5 years	More than 5 years
IP	◯	◯	◯	◯	◯
DSA	◯	◯	◯	◯	◯

6. Do you prepare examples of programs, algorithms, or data structures beforehand to be used as lesson material? *

Mark only one oval.

◯ Yes

◯ No

7. Do you think this task takes too long or it is tedious?

Mark only one oval.

	1	2	3	4	5	
Totally agree	◯	◯	◯	◯	◯	Totally disagree

8. Do you use visual resources, such as illustrations or animations to help with students' comprehension of algorithms and code snippets? *

IP: Introduction to Programming, Programming 101, or equivalent; DSA: Data structures and Algorithms or equivalent.

Mark only one oval per row.

	Sim	Não
IP	◯	◯
DSA	◯	◯
Other disciplines	◯	◯

9. What tools or strategies have you used to teach in your classes? *

Check all that apply.

- ☐ I only talk and do not use visual resources
- ☐ Draw sketches on a blackboard
- ☐ Use simple diagrams in lecture slides
- ☐ Build entire algorithm examples using slide presentations
- ☐ Write program code during class (python notebooks, live coding)
- ☐ Use programs or websites that build or contain interactive visualizations
- ☐ Use traditional debuggers to show program execution step by step

Other: ☐ ________________

10. Have you ever used any tool to automatically create program visualizations? *

Mark only one oval.

◯ Yes

◯ No

11. Which visualization tools have you used?

Only answer this question if you have already used some visualization tool

Check all that apply.

☐ Python Tutor
☐ VisuAlgo
☐ OpenDSA
☐ UUhisle

Other: ☐ ____________________

12. What did you think of the results?

Only answer this question if you have already used some visualization tool

Mark only one oval.

	1	2	3	4	5	
Very unsatisfactory	◯	◯	◯	◯	◯	Very satisfactory

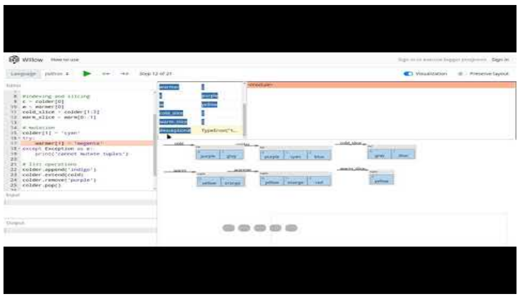
Introductory Programming courses

This section contains questions about using Willow in the context of an introductory programming discipline or similar.

Note: The visualizations shown are created directly from the program source code.
We recommend you to open the videos in a new tab by clicking on the "youtube" button (after starting it).

13. Which topics in introductory programming courses do you see as most challenging for students? Please, provide some rationale, if possible.

This video shows basic interactions with tuples and lists using Willow. The goal is to show the behavior of the operations over mutating structures.


http://youtube.com/watch?v=ntxWBNUuvFU

14. Do you think the effect of operations and functions over lists and tuples in the video above is clear and that the features of Willow can help students to comprehend mutability?

Mark only one oval.

	1	2	3	4	5	
Totally disagree	⬭	⬭	⬭	⬭	⬭	Totally agree

15. If you want, you can provide the rationale for your answer from the previous question.

____
____
____
____

16. Are difficulties or misconceptions in understanding the concept of recursion common among your students?

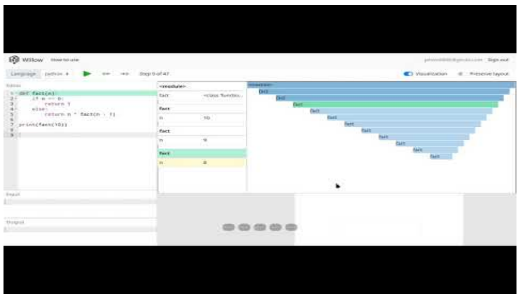
Mark only one oval.

- ( ) Yes
- ( ) No

17. What strategies or metaphors do you use to explain recursion?

____
____
____
____

This video shows the runtime behavior of a simple recursive algorithm using Willow.



```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)

print(fact(10))
```

http://youtube.com/watch?v=fiOc3QjpDPo

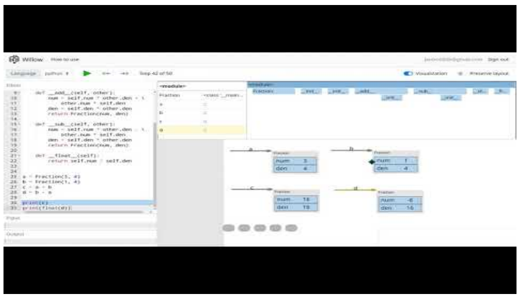
18. Regarding recursion, do you think your students can benefit from the features illustrated in the video?

Mark only one oval.

	1	2	3	4	5	
Totally disagree	◯	◯	◯	◯	◯	Totally agree

19. If you want, you can provide the rationale for your answer from the previous question.

This video shows an example of class and objects, also showing how special methods are called by operators.



```
def __add__(self, other):
    num = self.num * other.den + \
        other.num * self.den
    den = self.den * other.den
    return Fraction(num, den)

def __sub__(self, other):
    num = self.num * other.den - \
        other.num * self.den
    den = self.den * other.den
    return Fraction(num, den)

def __float__(self):
    return self.num / self.den

a = Fraction(3, 4)
b = Fraction(1, 4)
c = a + b
d = b - a

print(d)
print(float(d))
```

http://youtube.com/watch?v=nBcGBUSRBdk

20. The object-oriented paradigm is a complex subject for novices. Many professors recur to illustrations to demonstrate the relationship between classes and objects. Do you think Willow can help to create visualizations about this subject to be presented to students?

Mark only one oval.

	1	2	3	4	5	
Totally disagree	⬭	⬭	⬭	⬭	⬭	Totally agree

21. If you want, you can provide the rationale for your answer from the previous question.

____

____

____

____

____

Data Structures and Algorithms

This section contains questions about using Willow in the context of a data structures and algorithms course.

Note: The visualizations shown are created directly from the program source code.
We recommend you to open the videos in a new tab by clicking on the "youtube" button (after starting it).

22. How much do you agree with this statement: "Visualizing animated versions of algorithms and data structures helps students on understanding how they work".

Mark only one oval.

	1	2	3	4	5	
Totally disagree	⬭	⬭	⬭	⬭	⬭	Totally agree

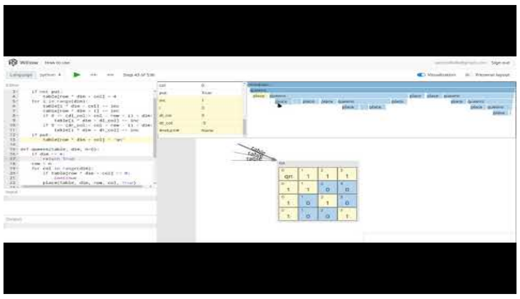
23. If you want, you can provide the rationale for your answer from the previous question.

The video shows an array ordered by the quicksort algorithm. A column-based representation is used to facilitate perceiving operations on the list.

```
def quicksort(array, left, right):
    if left >= right: return
    i, j = left, right
    pivot = array[(left + right) // 2]
    while i <= j:
        while array[i] < pivot:
            i += 1
        while array[j] > pivot:
            j -= 1
        if i <= j:
            array[i], array[j] = array[j], array
            i, j = i + 1, j - 1
    quicksort(array, left, j)
    quicksort(array, i, right)

array = [
    12, 11, 10, 9, 13, 16, 15, 14,
    8, 3, 2, 1, 0, 4, 7, 6, 5
]
quicksort(array, 0, len(array) - 1)
```

http://youtube.com/watch?v=VuoXhjiR3OI

This video is a visualization of the N queens problem. This time, a matrix represents the board. When a new queen is placed, we can see the board highlighting the changes.



http://youtube.com/watch?v=UG22rlOsPoA

24. The videos above show the same data structure (list) represented in different forms. Do you think that flexible visualizations may contribute to students' understanding of an algorithm or data structure behavior if used correctly?

Mark only one oval.

	1	2	3	4	5	
Totally disagree	⬭	⬭	⬭	⬭	⬭	Totally agree

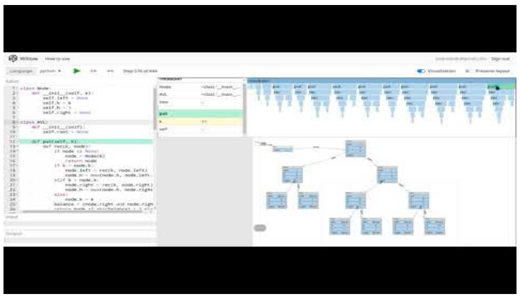
25. If you want, you can provide the rationale for your answer from the previous question.

26. Suppose you will teach a lesson on balanced trees. How would you explain rotation operations to students?

Check all that apply.

☐ Draw on the blackboard the trees before and after the rotation
☐ Create lecture slides showing examples of tree rotations
☐ Use visualization tools to create examples of tree rotations
Other: ☐ ____________________

This video contains a series of insertions of sequential values in an AVL tree, showing, in a general way, how rotation operations modify the tree.


http://youtube.com/watch?v=KsApC71GxNQ

27. What did you think of the animations shown in the video above?

Mark only one oval.

	1	2	3	4	5	
Very unsatisfactory	◯	◯	◯	◯	◯	Very satisfactory

28. If you want, you can provide the rationale for your answer from the previous question.

29. Do you agree that the animations shown in the video could help to complement the material you proposed previously?

Mark only one oval.

	1	2	3	4	5	
Totally disagree	◯	◯	◯	◯	◯	Totally agree

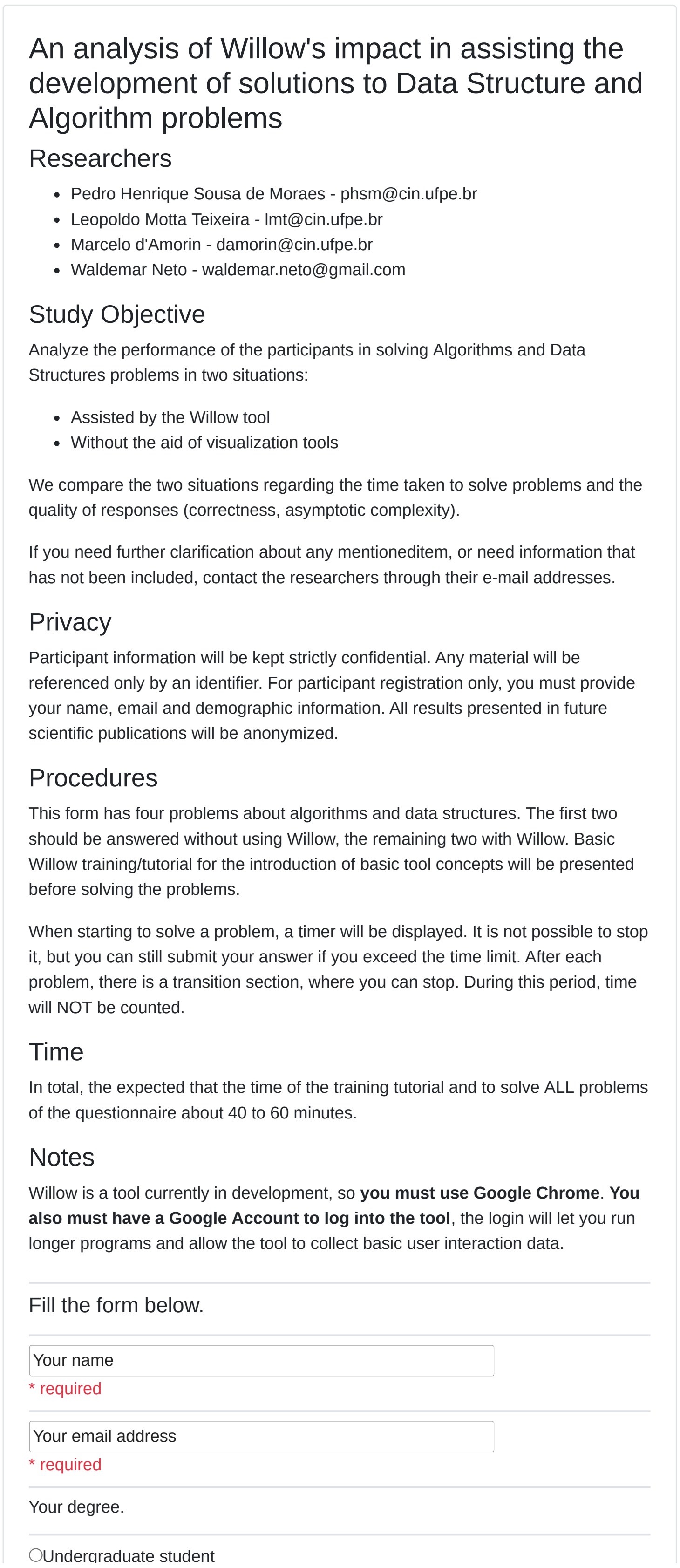
30. If you want, you can provide the rationale for your answer from the previous question.

Conclusion

31. Do you think Willow could contribute in any way to you as a professor and to your students? Why?

32. Are there any suggestions, comments, or criticism you want to make?

APPENDIX B - EXPERIMENT FORM



An analysis of Willow's impact in assisting the development of solutions to Data Structure and Algorithm problems

Researchers

- Pedro Henrique Sousa de Moraes - phsm@cin.ufpe.br
- Leopoldo Motta Teixeira - lmt@cin.ufpe.br
- Marcelo d'Amorin - damorin@cin.ufpe.br
- Waldemar Neto - waldemar.neto@gmail.com

Study Objective

Analyze the performance of the participants in solving Algorithms and Data Structures problems in two situations:

- Assisted by the Willow tool
- Without the aid of visualization tools

We compare the two situations regarding the time taken to solve problems and the quality of responses (correctness, asymptotic complexity).

If you need further clarification about any mentioneditem, or need information that has not been included, contact the researchers through their e-mail addresses.

Privacy

Participant information will be kept strictly confidential. Any material will be referenced only by an identifier. For participant registration only, you must provide your name, email and demographic information. All results presented in future scientific publications will be anonymized.

Procedures

This form has four problems about algorithms and data structures. The first two should be answered without using Willow, the remaining two with Willow. Basic Willow training/tutorial for the introduction of basic tool concepts will be presented before solving the problems.

When starting to solve a problem, a timer will be displayed. It is not possible to stop it, but you can still submit your answer if you exceed the time limit. After each problem, there is a transition section, where you can stop. During this period, time will NOT be counted.

Time

In total, the expected that the time of the training tutorial and to solve ALL problems of the questionnaire about 40 to 60 minutes.

Notes

Willow is a tool currently in development, so **you must use Google Chrome**. **You also must have a Google Account to log into the tool**, the login will let you run longer programs and allow the tool to collect basic user interaction data.

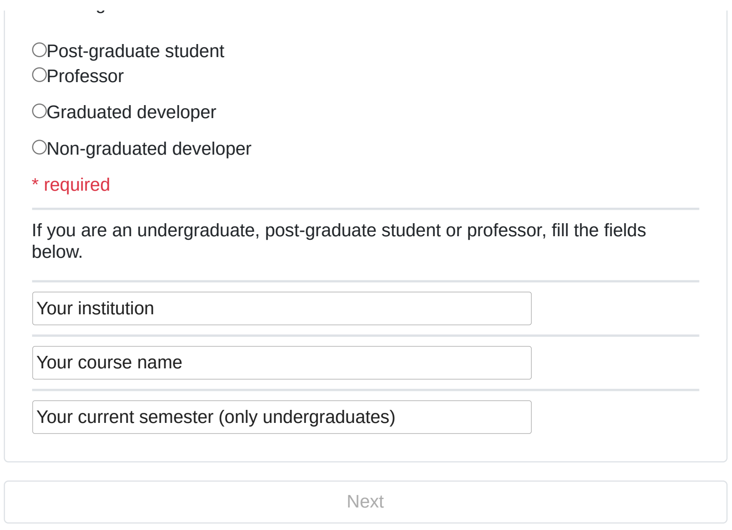
Fill the form below.

Your name
* required

Your email address
* required

Your degree.

○Undergraduate student



○Post-graduate student
○Professor

○Graduated developer

○Non-graduated developer

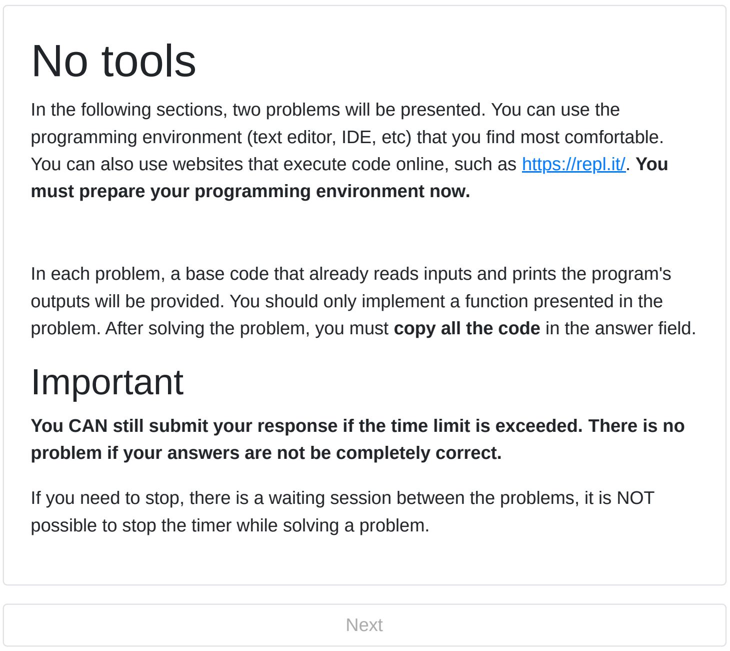
* required

If you are an undergraduate, post-graduate student or professor, fill the fields below.

Your institution

Your course name

Your current semester (only undergraduates)



No tools

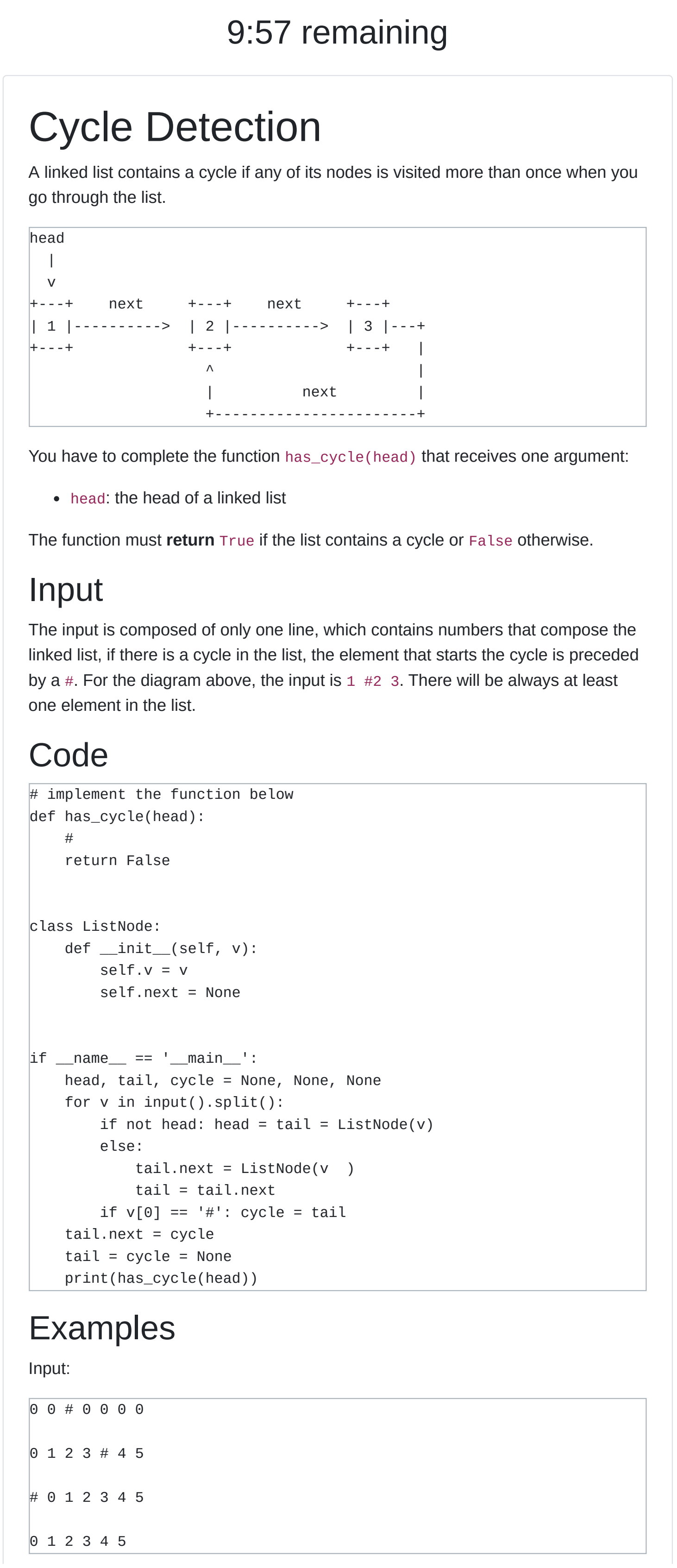
In the following sections, two problems will be presented. You can use the programming environment (text editor, IDE, etc) that you find most comfortable. You can also use websites that execute code online, such as https://repl.it/. **You must prepare your programming environment now.**

In each problem, a base code that already reads inputs and prints the program's outputs will be provided. You should only implement a function presented in the problem. After solving the problem, you must **copy all the code** in the answer field.

Important

You CAN still submit your response if the time limit is exceeded. There is no problem if your answers are not be completely correct.

If you need to stop, there is a waiting session between the problems, it is NOT possible to stop the timer while solving a problem.



9:57 remaining

Cycle Detection

A linked list contains a cycle if any of its nodes is visited more than once when you go through the list.

```
head
  |
  v
+---+    next     +---+    next     +---+
| 1 |---------->  | 2 |---------->  | 3 |---+
+---+             +---+             +---+   |
                    ^                       |
                    |          next         |
                    +-----------------------+
```

You have to complete the function `has_cycle(head)` that receives one argument:

- `head`: the head of a linked list

The function must **return** `True` if the list contains a cycle or `False` otherwise.

Input

The input is composed of only one line, which contains numbers that compose the linked list, if there is a cycle in the list, the element that starts the cycle is preceded by a `#`. For the diagram above, the input is `1 #2 3`. There will be always at least one element in the list.

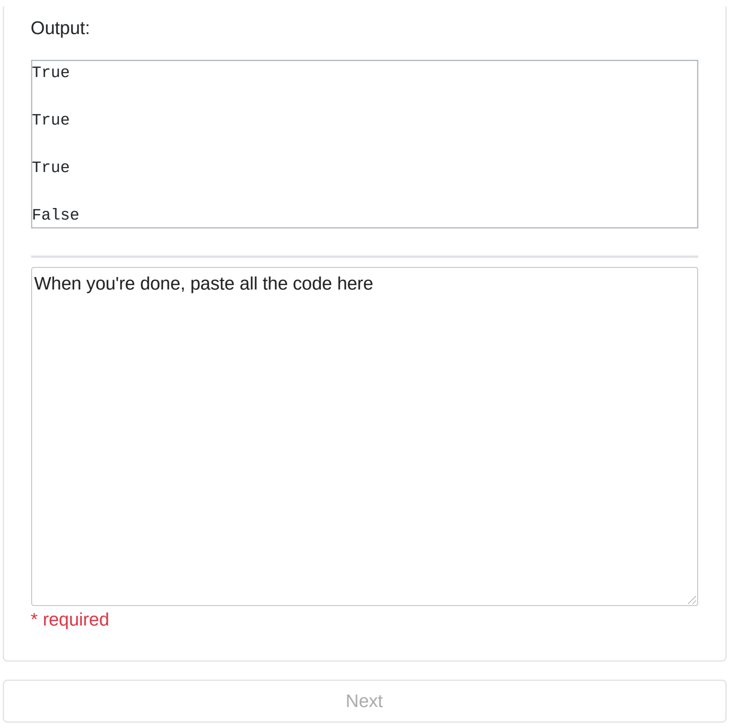
Code

```
# implement the function below
def has_cycle(head):
    #
    return False


class ListNode:
    def __init__(self, v):
        self.v = v
        self.next = None


if __name__ == '__main__':
    head, tail, cycle = None, None, None
    for v in input().split():
        if not head: head = tail = ListNode(v)
        else:
            tail.next = ListNode(v  )
            tail = tail.next
        if v[0] == '#': cycle = tail
    tail.next = cycle
    tail = cycle = None
    print(has_cycle(head))
```

Examples

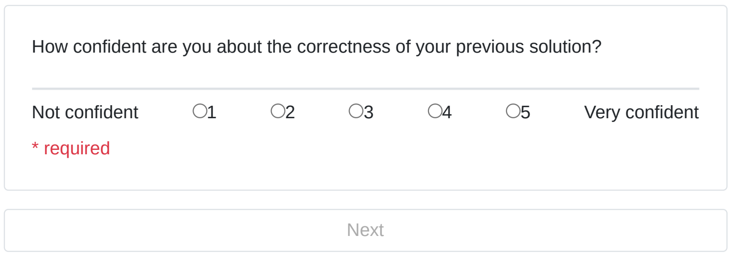
Input:

```
0 0 # 0 0 0 0

0 1 2 3 # 4 5

# 0 1 2 3 4 5

0 1 2 3 4 5
```


Output:

```
True

True

True

False
```

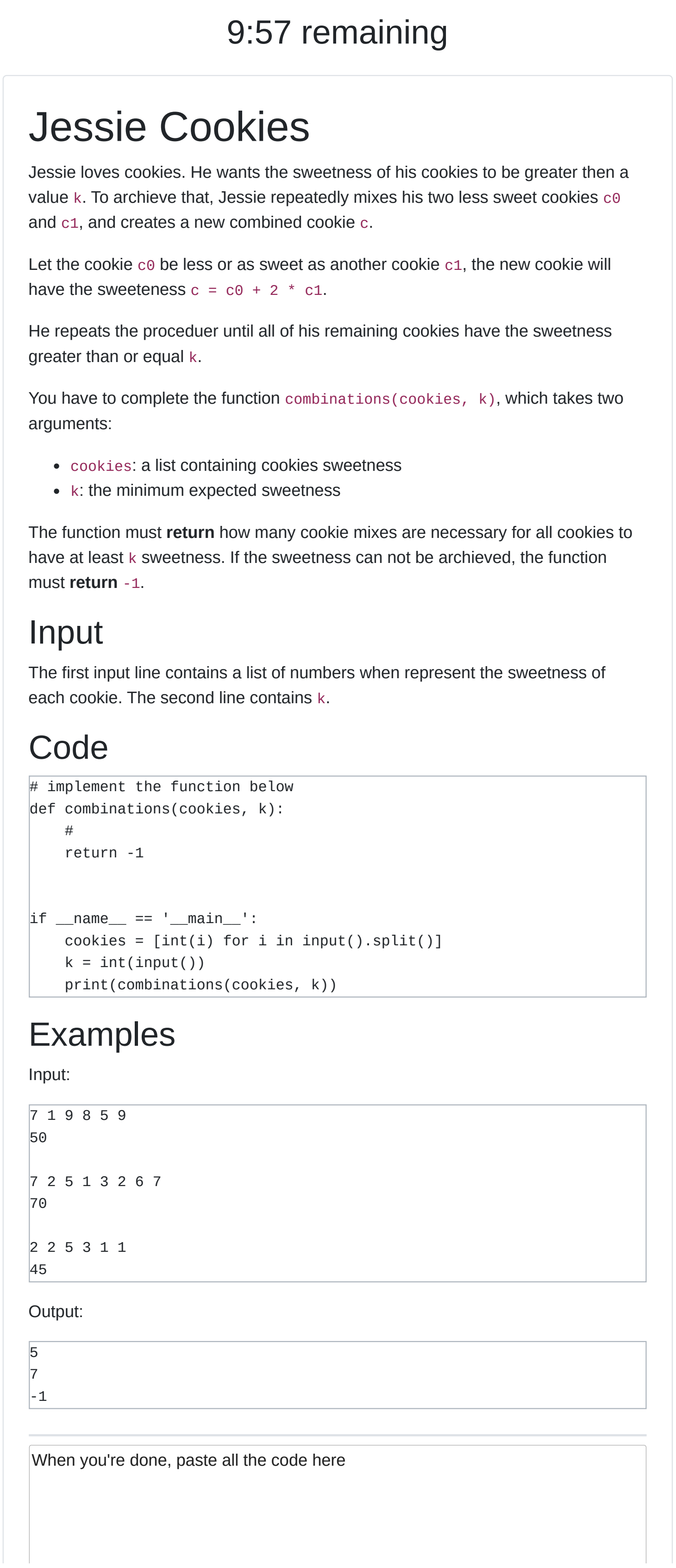
When you're done, paste all the code here



How confident are you about the correctness of your previous solution?

Not confident	○1	○2	○3	○4	○5	Very confident

* required



9:57 remaining

Jessie Cookies

Jessie loves cookies. He wants the sweetness of his cookies to be greater then a value `k`. To archieve that, Jessie repeatedly mixes his two less sweet cookies `c0` and `c1`, and creates a new combined cookie `c`.

Let the cookie `c0` be less or as sweet as another cookie `c1`, the new cookie will have the sweeteness `c = c0 + 2 * c1`.

He repeats the proceduer until all of his remaining cookies have the sweetness greater than or equal `k`.

You have to complete the function `combinations(cookies, k)`, which takes two arguments:

- `cookies`: a list containing cookies sweetness
- `k`: the minimum expected sweetness

The function must **return** how many cookie mixes are necessary for all cookies to have at least `k` sweetness. If the sweetness can not be archieved, the function must **return** `-1`.

Input

The first input line contains a list of numbers when represent the sweetness of each cookie. The second line contains `k`.

Code

```
# implement the function below
def combinations(cookies, k):
    #
    return -1


if __name__ == '__main__':
    cookies = [int(i) for i in input().split()]
    k = int(input())
    print(combinations(cookies, k))
```

Examples

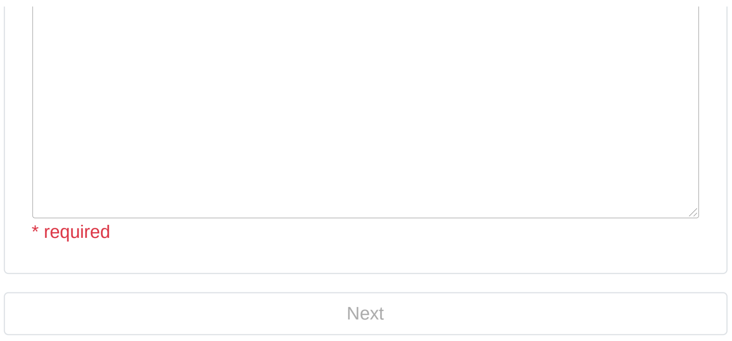
Input:

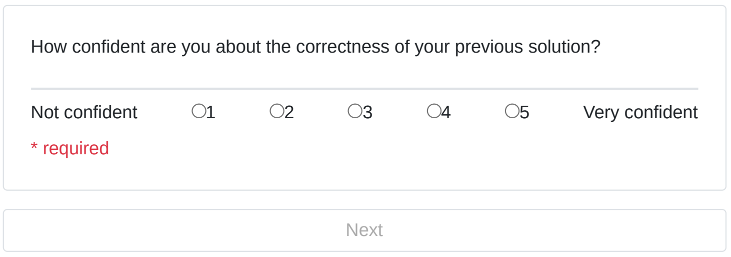
```
7 1 9 8 5 9
50

7 2 5 1 3 2 6 7
70

2 2 5 3 1 1
45
```

Output:

```
5
7
-1
```

When you're done, paste all the code here


* required



How confident are you about the correctness of your previous solution?

Not confident ○1 ○2 ○3 ○4 ○5 Very confident

* required

Willow

In the following sections, two problems will be presented. You must use Willow to help visualizing and solving the problems.

Tutorial

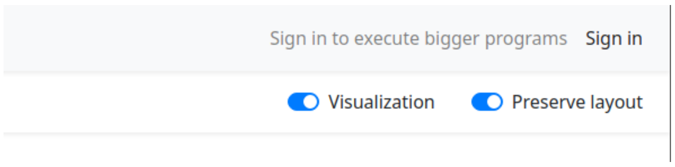
You **must follow this tutorial slowly and paying close attention to the text** before advancing to the next problems. The estimated time to finish this tutorial ranges from **6 to 10 minutes**.

Before proceeding to the problems, you must access Willow and follow the steps below. The steps will provide basic training on how to use Willow. Read the instructions, look at the images and clips, and then, do the same.

First click on the link below to access the Willow website.

https://willow-beta.web.app/

After accessing the site, you must log-in with a google account through the button in the right upper corner. The log-in will allow you to run longer programs.



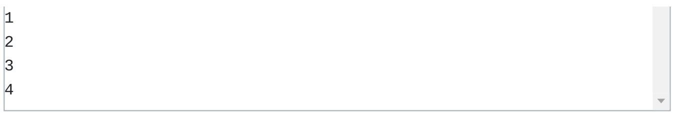
Step 1

Select the **Python** language in the toolbar, and then paste the code below into `Editor`. This code is a linked list implementation, and it will be used to show features of Willow.

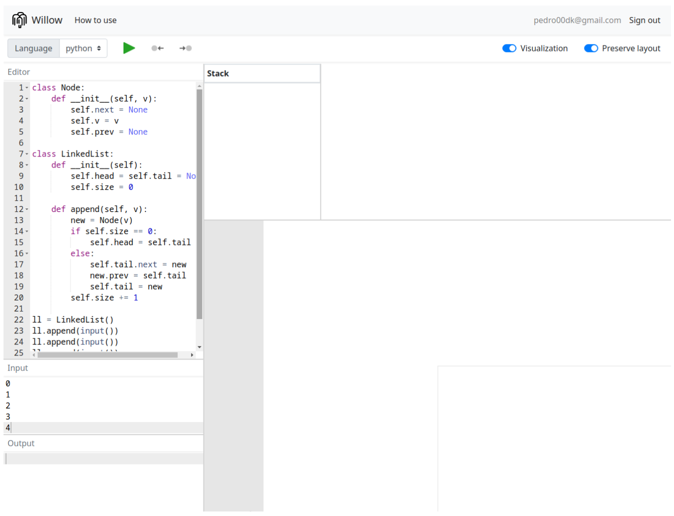
```
class Node:
    def __init__(self, v):
        self.next = None
        self.v = v
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = None
        self.size = 0

    def append(self, v):
        new = Node(v)
        if self.size == 0:
            self.head = self.tail = new
        else:
            self.tail.next = new
            new.prev = self.tail
            self.tail = new
        self.size += 1

ll = LinkedList()
ll.append(input())
ll.append(input())
ll.append(input())
ll.append(input())
ll.append(input())
```

Paste also the program inputs into `Input`, which are the values that will be used to create the list nodes.

```
0
```



```
1
2
3
4
```

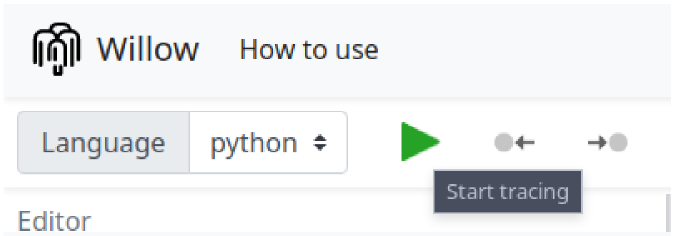


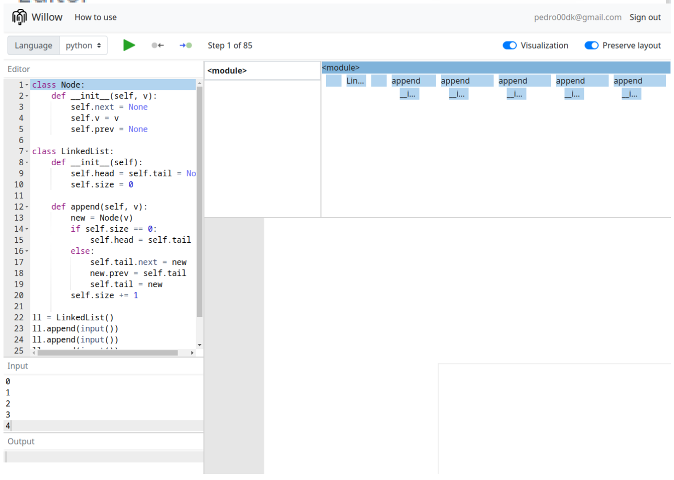
```python
class Node:
    def __init__(self, v):
        self.next = None
        self.v = v
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = No
        self.size = 0

    def append(self, v):
        new = Node(v)
        if self.size == 0:
            self.head = self.tail
        else:
            self.tail.next = new
            new.prev = self.tail
            self.tail = new
        self.size += 1

ll = LinkedList()
ll.append(input())
ll.append(input())
```

```
0
1
2
3
4
```

Step 2

To run the code, click the play button on the toolbar. Wait a few seconds for the visualization is generated.



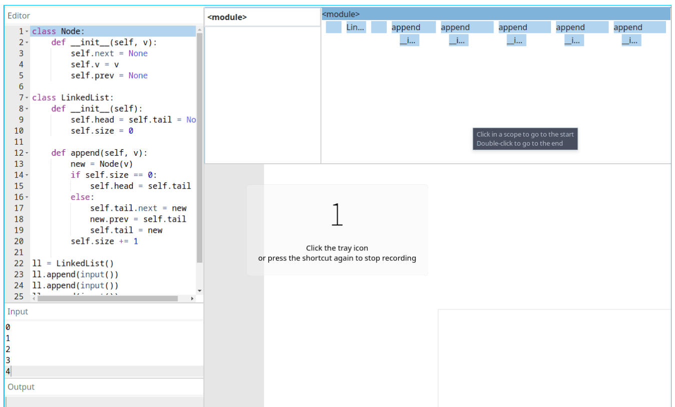


```python
class Node:
    def __init__(self, v):
        self.next = None
        self.v = v
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = No
        self.size = 0

    def append(self, v):
        new = Node(v)
        if self.size == 0:
            self.head = self.tail
        else:
            self.tail.next = new
            new.prev = self.tail
            self.tail = new
        self.size += 1

ll = LinkedList()
ll.append(input())
ll.append(input())
```

```
0
1
2
3
4
```

Step 3

Use the **right and left arrows** to navigate to the end of the program. **You can also click on a call tree scope to jump to this point in the program.** Double-clicking jumps to the end of a scope instead of the beginning.

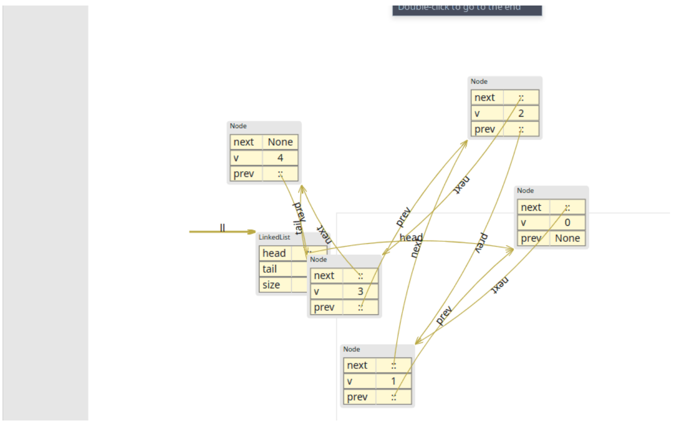

Editor
Input
Output
<module>
Lin...
append
Click in a scope to go to the start
Double-click to go to the end
Click the tray icon
or press the shortcut again to stop recording

```
class Node:
    def __init__(self, v):
        self.next = None
        self.v = v
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = No
        self.size = 0

    def append(self, v):
        new = Node(v)
        if self.size == 0:
            self.head = self.tail
        else:
            self.tail.next = new
            new.prev = self.tail
            self.tail = new
        self.size += 1

ll = LinkedList()
ll.append(input())
ll.append(input())
```

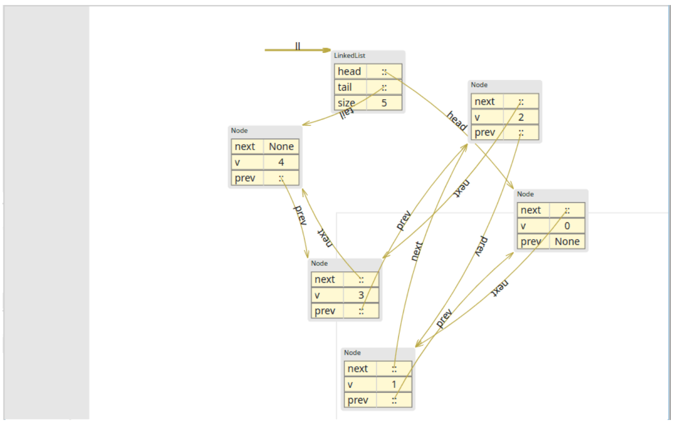
```
0
1
2
3
4
```

Step 4

Now, try to drag the heap objects around.



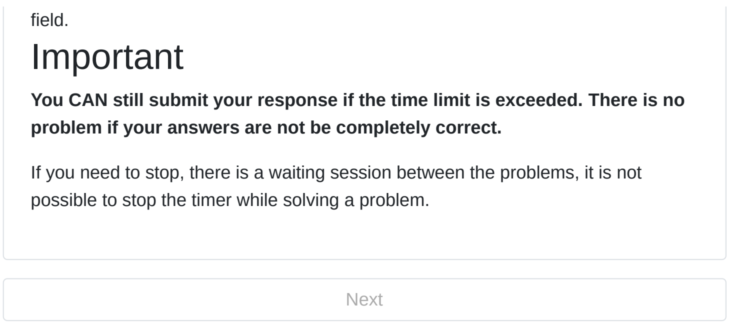
You can also activate automatic layout, just double-click on any internal object in the list. It also works with other linked data structures, such as binary trees.



When activating automatic layout, the outline of the nodes will be slightly darkened. If you move any node, the automatic arrangement will be disabled.

With automatic layout enabled, try to navigate through the program using the arrow keys and clicking on the scopes.

In each problem, a base code that already reads inputs and prints the outputs of the program will be provided. You should only implement a function presented in the problem. After solving the problem, you must **copy all the code** in the answer

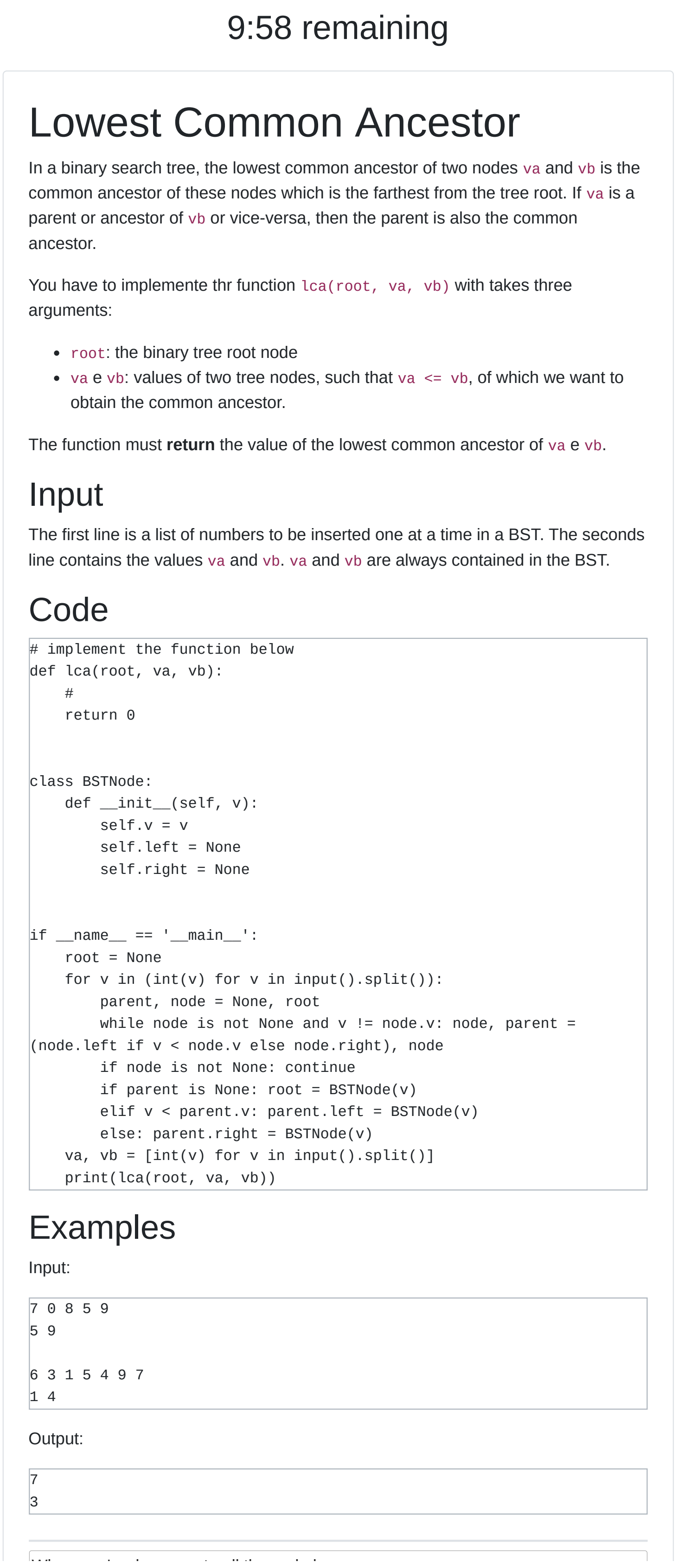


field.

Important

You CAN still submit your response if the time limit is exceeded. There is no problem if your answers are not be completely correct.

If you need to stop, there is a waiting session between the problems, it is not possible to stop the timer while solving a problem.



9:58 remaining

Lowest Common Ancestor

In a binary search tree, the lowest common ancestor of two nodes `va` and `vb` is the common ancestor of these nodes which is the farthest from the tree root. If `va` is a parent or ancestor of `vb` or vice-versa, then the parent is also the common ancestor.

You have to implemente thr function `lca(root, va, vb)` with takes three arguments:

- `root`: the binary tree root node
- `va` e `vb`: values of two tree nodes, such that `va <= vb`, of which we want to obtain the common ancestor.

The function must **return** the value of the lowest common ancestor of `va` e `vb`.

Input

The first line is a list of numbers to be inserted one at a time in a BST. The seconds line contains the values `va` and `vb`. `va` and `vb` are always contained in the BST.

Code

```
# implement the function below
def lca(root, va, vb):
    #
    return 0


class BSTNode:
    def __init__(self, v):
        self.v = v
        self.left = None
        self.right = None


if __name__ == '__main__':
    root = None
    for v in (int(v) for v in input().split()):
        parent, node = None, root
        while node is not None and v != node.v: node, parent =
(node.left if v < node.v else node.right), node
        if node is not None: continue
        if parent is None: root = BSTNode(v)
        elif v < parent.v: parent.left = BSTNode(v)
        else: parent.right = BSTNode(v)
    va, vb = [int(v) for v in input().split()]
    print(lca(root, va, vb))
```

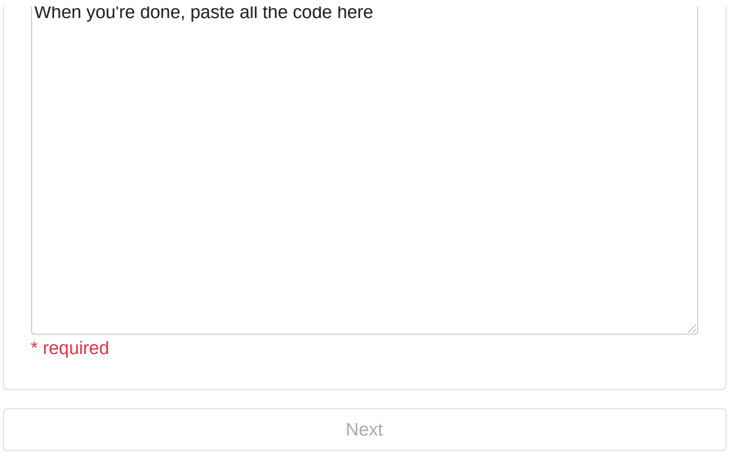
Examples

Input:

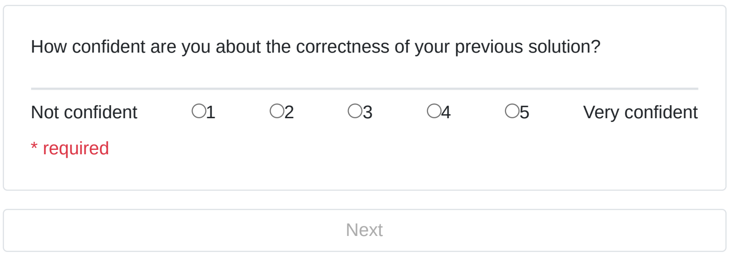
```
7 0 8 5 9
5 9

6 3 1 5 4 9 7
1 4
```

Output:

```
7
3
```


When you're done, paste all the code here

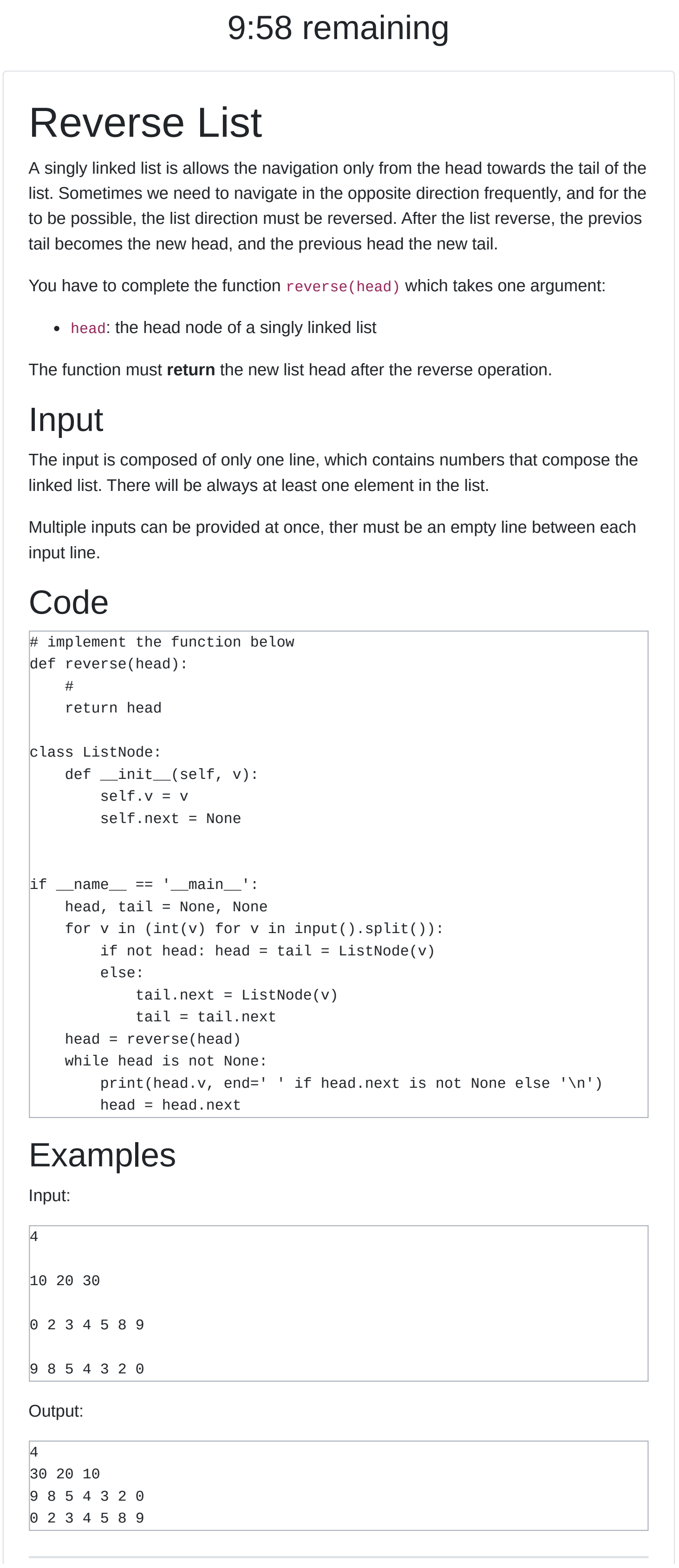
* required



How confident are you about the correctness of your previous solution?

Not confident	○1	○2	○3	○4	○5	Very confident

* required



9:58 remaining

Reverse List

A singly linked list is allows the navigation only from the head towards the tail of the list. Sometimes we need to navigate in the opposite direction frequently, and for the to be possible, the list direction must be reversed. After the list reverse, the previos tail becomes the new head, and the previous head the new tail.

You have to complete the function `reverse(head)` which takes one argument:

- `head`: the head node of a singly linked list

The function must **return** the new list head after the reverse operation.

Input

The input is composed of only one line, which contains numbers that compose the linked list. There will be always at least one element in the list.

Multiple inputs can be provided at once, ther must be an empty line between each input line.

Code

```
# implement the function below
def reverse(head):
    #
    return head

class ListNode:
    def __init__(self, v):
        self.v = v
        self.next = None


if __name__ == '__main__':
    head, tail = None, None
    for v in (int(v) for v in input().split()):
        if not head: head = tail = ListNode(v)
        else:
            tail.next = ListNode(v)
            tail = tail.next
    head = reverse(head)
    while head is not None:
        print(head.v, end=' ' if head.next is not None else '\n')
        head = head.next
```

Examples

Input:

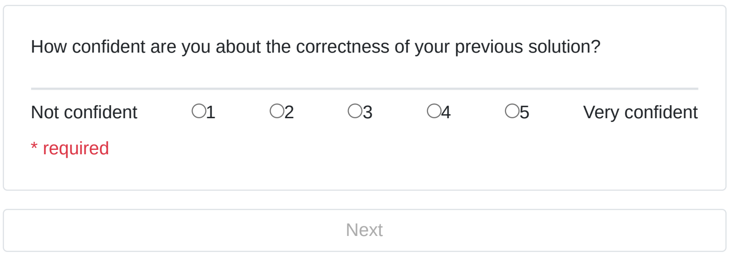
```
4

10 20 30

0 2 3 4 5 8 9

9 8 5 4 3 2 0
```

Output:

```
4
30 20 10
9 8 5 4 3 2 0
0 2 3 4 5 8 9
```

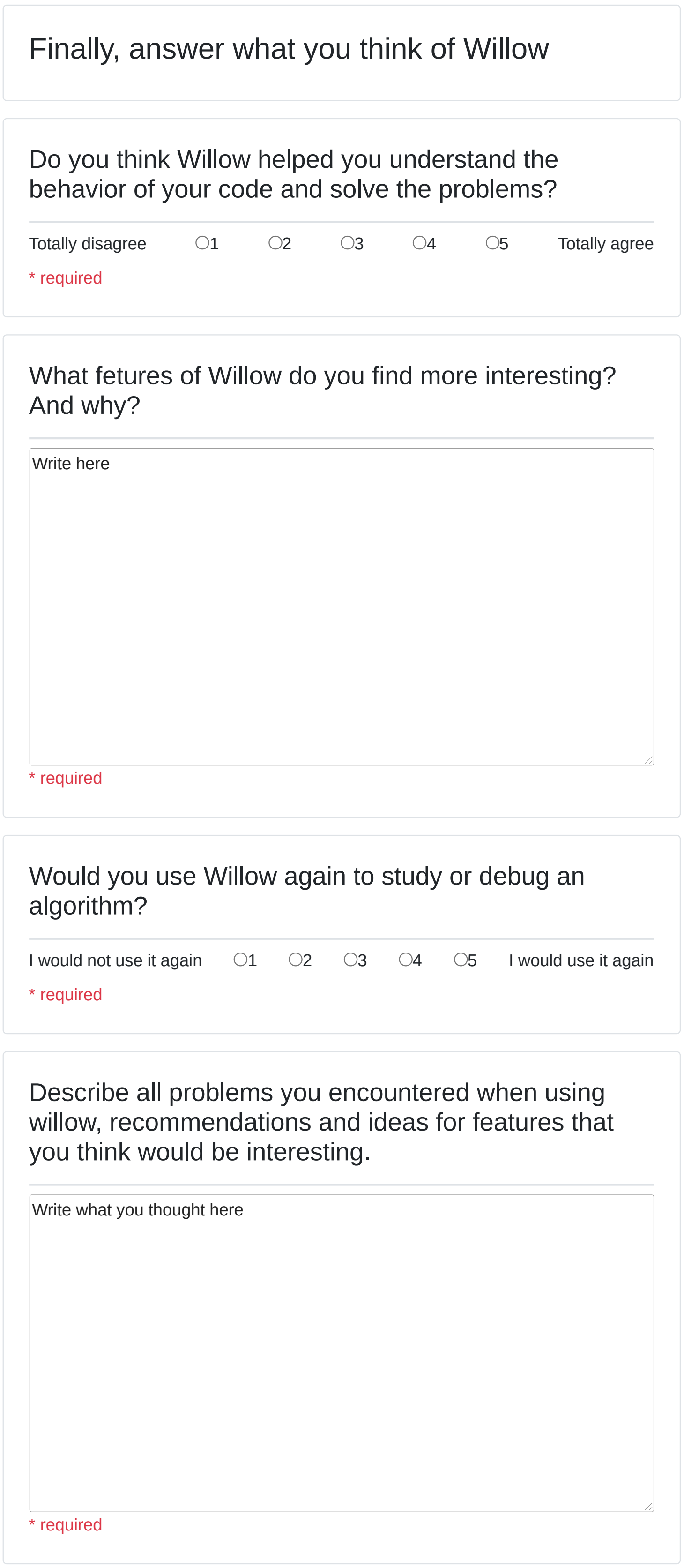
When you're done, paste all the code here

* required


How confident are you about the correctness of your previous solution?

Not confident	◯1	◯2	◯3	◯4	◯5	Very confident

* required



Finally, answer what you think of Willow

Do you think Willow helped you understand the behavior of your code and solve the problems?

Totally disagree	○1	○2	○3	○4	○5	Totally agree

* required

What fetures of Willow do you find more interesting? And why?

Write here

* required

Would you use Willow again to study or debug an algorithm?

I would not use it again	○1	○2	○3	○4	○5	I would use it again

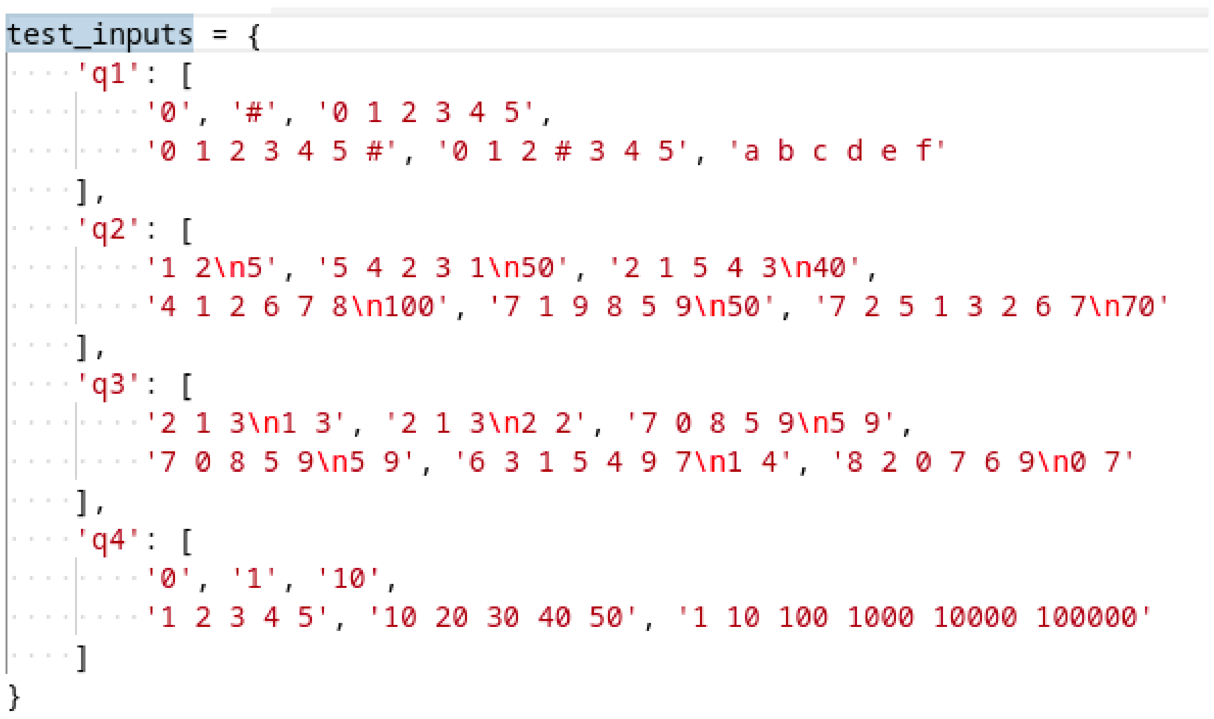
* required

Describe all problems you encountered when using willow, recommendations and ideas for features that you think would be interesting.

Write what you thought here

* required

APPENDIX C - EXPERIMENT TEST INPUTS



```
test_inputs = {
    'q1': [
        '0', '#', '0 1 2 3 4 5',
        '0 1 2 3 4 5 #', '0 1 2 # 3 4 5', 'a b c d e f'
    ],
    'q2': [
        '1 2\n5', '5 4 2 3 1\n50', '2 1 5 4 3\n40',
        '4 1 2 6 7 8\n100', '7 1 9 8 5 9\n50', '7 2 5 1 3 2 6 7\n70'
    ],
    'q3': [
        '2 1 3\n1 3', '2 1 3\n2 2', '7 0 8 5 9\n5 9',
        '7 0 8 5 9\n5 9', '6 3 1 5 4 9 7\n1 4', '8 2 0 7 6 9\n0 7'
    ],
    'q4': [
        '0', '1', '10',
        '1 2 3 4 5', '10 20 30 40 50', '1 10 100 1000 10000 100000'
    ]
}
```