Università degli Studi di Udine

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Dottorato di Ricerca in Informatica, Matematica e Fisica
Ciclo XXXIV

Ph.D. Thesis

Task-related models for teaching and assessing iteration learning in high school

Candidate
Emanuele Scapin

Supervisor
Prof. Alberto Policriti

Co-Supervisor
Dr. Claudio Mirolo

Year 2022

Author's e-mail: emanuele.scapin@uniud.it; emanuele.scapin@escapin.it

Author's address:

Department of Mathematics, Computer Science and Physics
University of Udine
via delle Scienze, 206
33100 Udine
Italy

To Alessandra and to my family

I know nothing and my heart aches

Fernando Pessoa, *The Book of Disquiet*

Abstract

A number of studies report students' difficulties with basic flow-control constructs, and specifically with iteration. Although such issues are less explored in the context of pre-tertiary education, this seems to be especially the case for high-school programming learning, where the difficulties concern both the "mechanical" features of the notional machine as well as the logical aspects connected with the constructs, ranging from the implications of loop conditions to a more abstract grasp of the underlying algorithms.

The overall picture with regard to the teaching and learning of iteration at the upper secondary level of education is, however, still rather fragmentary and calls for more systematic studies. Thus, the present work is a contribution to stimulate such an undertaking, in the context of the Italian high school. This study has multiple purposes: i) to obtain some insight on teachers' perception and instructional practice; ii) to investigate students' perception, comprehension and self-confidence when engaging in programming tasks focused on iteration; iii) to develop methodological tools to enhance the understanding of the iteration constructs.

To begin with, we interviewed a few experienced upper secondary teachers of introductory programming in different kinds of schools. The interviews were mainly aimed at ascertaining teachers' beliefs about major sources of issues for basic programming concepts and their approach to the teaching and learning of iteration constructs. Once teachers' perception of students' difficulties were identified, we administered a survey to a sample of students, which included both questions on their subjective perception of difficulty and simple tasks probing their understanding of iteration. Data collected from teachers and students confirm that iteration is a central programming concept and indicate that loop conditions and nested constructs are major sources of difficulties with iteration.

These two preliminary explorations raised a number of questions worth further investigation. In addition, the feedback from teachers and students provided helpful clues leading to the development of a catalog of significant (small) programming tasks with the twofold aim of assessing students' understanding, as well as supporting the learning of iteration constructs.

We tried to answer some of the questions emerging from the previous steps by designing two online surveys, respectively addressed to teachers and students. The former survey was meant to collect information about teachers' pedagogical content knowledge and good practices. As to the pedagogical content knowledge, we distinguished two meaningful frameworks to structure the data analysis. On the one hand, an orientation towards 'conceptual versus practical objectives' and, on the other hand, between 'process-based versus product-based assessments'.

Finally, based on the preliminary feedback from students and drawing from the tasks listed in the catalog, we designed and administered a (new) student survey to investigate in more depth high-school students' understanding of iteration in terms of code reading abilities. The proposed tasks covered a variety of features in connection with iteration: technical program features, correlation between tracing effort and abstraction, implications of the use of flow-charts, and subjective perception of self-confidence.

To sum up, the main implications of this work are twofold. Firstly, the data collected from teachers and students provide a clearer and more detailed picture of how the teaching of iteration is approached in the high-school and of students' understanding of iterative programs. Secondly, the nature of the proposed tasks and the structure of the survey can lay the basis for further developments, both from an instructional perspective — in particular, a first draft of a catalog of programming tasks addressing different facets of the understanding of iteration — and from an educational research perspective — the structure of the student survey can be the starting point to broaden the scope of investigation.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis is the result of three years of research and development that have given me the opportunity to enrich myself as well as to participate in several collaborative projects, conferences and to interact with colleagues.

Primarily, I would like to thank my supervisors, Dr. Claudio Mirolo and Prof. Alberto Policriti, for this experience, for everything they taught me.

A heartfelt thanks to my school, I.T.T. G.Chilesotti (Thiene, VI, Italy) for having endured my absence for three years to my colleagues for their collaboration. A thought goes to my students who saw me leave for this adventure and yet were deprived of their teacher and his experience. I also thank all those high school colleagues whom I have had the opportunity to meet in recent years and who have participated in the activities related to this research project.

Special thanks to colleagues Lucia Carli and Eugenio Macor for their collaboration in the submission phase of the surveys and in the subsequent analysis of the results obtained.

Thanks also to my friend Luca Bognolo and to my colleague Nicola Dalla Pozza for their assistance in correcting the text of this thesis.

Finally, a big thank to my family, my partner Alessandra and my parents who have never stopped encouraging and supporting me during this experience.

Last but not least, I would like to thank Prof. Furio Honsell for the insights and suggestions that led me to embark on this experience.

Prologue

My teaching experience began suddenly, one day I was a software engineer in a large company in my area, the next day I became a Computer Science teacher in a small mountain high school.

The first day was weird for me — then I found out it was normal — they just told me which classes I should be teaching, gave me the register, and then I started the teaching adventure. During this first period of my teaching I never had the opportunity to attend courses, aimed at teachers, with a focus on the teaching of Computer Science, dealing in an in-depth and exhaustive way with students' difficulties, and exploring more effective strategies for teaching the various topics of the discipline. I realized that, for a teacher, to rely only on his own knowledge, experience, common sense and comparison with colleagues would not have been enough on every occasion, to create stimuli, curiosity, autonomy, self-confidence and the most appropriate path for each student. The only real and concrete exception was the attendance of the SSIS[1] Veneto courses (University of Venice), which, however, failed to satisfy all my curiosities and respond to all my questions.

A few years ago, by chance, I came across the text by O.Hazzan, N.Ragonis and T. Lapidot *"Guide to teaching Computer Science"* [HLR11]: an enlightening book that gave answers to some of the questions that I had set myself, a real and concrete vision, accompanied by laboratory phases and practical experiences. This reading also made me reflect on the scarcity of studies, texts, experiences, teaching methods, related to Computer Science in my country.

From here, as well as from the confrontation with Prof. Furio Honsell, and then with Dr. Claudio Mirolo, the idea of this research project arose, which I carried out despite the obstacles that the Covid-19 pandemic created.

[1]Scuola di Specializzazione all'Insegnamento Secondario (SSIS) was an Italian university specialization school, of bi-annual duration, aimed at training teachers of lower and upper secondary schools.

Introduction

Teaching students to program is a complex, “slow and gradual process”, as argued by Dijkstra [Dij89]. Robins et al. [RRR03, p. 137], who provided a first comprehensive overview of research on novice programmers, setting the general tone already in the first paragraph of their review:

> *Learning to program is hard [. . .] Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates.*

Moreover, in programming languages “the final product must be, at least to a certain functional level, complete, unambiguous, and error free” [Rob19].

A characteristic of programming is that it is *problem-solving intensive* [PSS88], demanding a significant amount of effort in several skill areas. According to some educators, indeed, programming requires “not a single, but a set of skills” [Jen02; GM07]. De Raadt [DR08] observes that “novices must learn the programming *knowledge* (syntax and language features) and programming *strategies* (ways to apply this *knowledge* in order to solve programming problems)”. In particular, a major challenge students face is being able to translate the solution strategy of a problem into an algorithm. The teacher is then required to grant adequate learning time as well as to present a variety of meaningful examples in order to motivate the students, who would otherwise perceive programming as difficult and boring [Rep16; Bec21].

Students’ difficulties in introductory courses are well known in Computer Science education, e.g. [LAMJ05; Lew+05a; QL17], as shown by the high dropout rates in tertiary education [Jen02; Lew+05a]. In particular, they appear to struggle with programming tasks. This may be ascribed to different reasons, such as lack of problem solving skills, failure to acquire adequate meta-cognitive knowledge [Cot06], or the peculiar study method required to learn programming — that, unlike other subjects, should mainly be based on intensive practice [Gom+20].

A number of studies report students’ difficulties with basic flow-control constructs, and specifically with iterations. Although such issues are less explored in the context of pre-tertiary education, this seems to be especially the case for high-school programming learning, where the difficulties concern both the “mechanical” features of the notional machine as well as the logical aspects connected with the constructs, ranging from the implications of loop conditions to a more abstract grasp of the underlying algorithms.

According to T. Wood [Woo06], who refers to the studies of L.S. Shulman [Shu86; Shu05a], another critical aspect of Computer Science instruction is that teachers' education is inadequate. While disciplines such as law or medicine have *"signature pedagogies"*, in other cases the development of teachers' competencies is left to their good sense, field experience, empathy toward the students and understanding of their needs and difficulties. However, J. Hattie's studies [Hat12] show that the main factor contributing to student learning is the teacher, hence the need to identify good practices and useful suggestions. And, in this respect, a remarkable source to draw inspiration can be found in Israeli approaches [HLR11], where the Methods of Teaching Computer Science (MTCS) course has been consolidated, and where high school Computer Science teachers specifically engage, in their studies, with CS teaching.

In light of this, we devised a project aimed at identifying methodological tools to enhance a comprehensive understanding of the iteration constructs. At first, we designed the following main steps:

1. Interviewing a pilot sample of instructors about their approach to the teaching of iteration and their perception of students' difficulties;
2. Collecting information about students' perception on the topic via a short survey;
3. Based on the outcome of steps 1 and 2, designing a survey to collect more focused information and good practices from teachers;
4. Identifying some methodological approaches to the teaching of iteration and building a catalog of significant program examples to support students' learning;
5. Testing the instructional strategies in class to assess their effectiveness.

The last step was intended to be carried out in the field, but unfortunately we were unable to plan appropriate interventions due to the difficulties of interacting with schools in the course of the Covid-19 pandemic. So, we opted for an alternative route; namely, we designed a set of small program comprehension tasks, or *tasklets*, addressing a broad range of issues concerning students' understanding of iteration constructs. Such tasklets, in fact inspired by the catalog developed in step 4, were then administered to high school classes by means of an online survey.

Motivations, scope and aims of this work

In my teaching experience I have often faced students' learning difficulties with basic programming concepts. Fundamental topics in introductory programming courses are the flow-control constructs and, in particular, the iteration (loop) constructs. Although a number of works have explored students' difficulties with iteration in tertiary education [FAO10; Kac+10; Mir12; CZP14; Cet15], these issues have not yet been extensively investigated in the upper secondary school context.

So, we felt the need to engage in a more systematic and in-depth investigation about the teaching and learning of iteration in high school. Therefore, it was important for us to investigate teachers' beliefs concerning their students' difficulties, both with general programming and, more specifically, with iterations. Teachers have specific opinions about the difficulties of their students, which they check daily, regarding some specific topics that are particularly challenging. By focusing on iterations we have tried to understand in which areas students struggle the most. To carry out this investigation, however, it was important to understand which teaching methodologies and which problems and examples teachers propose to students so that they can learn loops.

Furthermore, we wanted to investigate whether iterations are difficult for students to understand, especially in pre-tertiary education, but also to determine if students are aware of them. It was important for us to understand in which areas of the use of iterations students encounter the most difficulties. The outcomes may allow to identify learning support strategies for students or alternative teaching models for teachers.

Moreover, comparing teachers' opinions and students' perceptions of their learning difficulties could be useful to settle any differences of opinion, in order to suggest interventions to update teaching methodologies.

As an outcome of this endeavor, we expected to get detailed insights on the nature of students' difficulties, as well as to identify possible interventions and reinforcement strategies to be adopted by teachers. In this respect, we have also attempted to start the construction of a catalog of small tasks that teachers could use both as examples to illustrate different aspects connected to iteration and as instruments to assess students' understanding of this topic.

In addition, both the catalog and the survey, derived from it, can become the basis for designing useful teaching and evaluation tools, promoting the analysis of loops in relation to multiple topics, exploiting examples and non-trivial tasks.

Research Goals

This research was guided by several research questions.

RQ1 What are students' major difficulties with iteration in teachers' view?

RQ2 What is teachers' approach to teach and assess the learning of iteration?

RQ3 What are students' major difficulties with iteration in their subjective perception?

RQ4 Are there differences between students' and teachers' perceptions of difficulties?

RQ5 To what extent are students self-confident about their comprehension of iteration?

Methodology

From a methodological viewpoint, our work can be outlined by subdividing it into four major parts:

1. Literature review, and more specifically:
   (a) Analysis of the studies referenced in two broad reviews about the teaching and learning of programming, namely [RRR03] and [LR+18];
   (b) Review of papers discussing teachers' Pedagogical Content Knowledge (PCK) in Computer Science and programming — this was meant to inform both the structure of the interviews and the survey addressed to teachers;
   (c) Extension of the review of papers on novices' understanding of flow-control, conditions and iteration constructs (including Concept Inventories) — this was meant to develop the surveys addressed to high school students.
2. Pilot investigations to get some preliminary feedback from samples of teachers and students:
   (a) Interviews of high school teachers to get insights about their PCK and to know their perception of students' understanding of and difficulties with iteration;
   (b) Administration of a survey to get insights about students' subjective perception of difficulties with iteration and actual achievements in a small set of tasklets;
   (c) Comparison of students' vs. teachers' perception and of students' perception vs. performance in the preliminary tasks — the resulting feedback was meant to inform the design of two more focused surveys for teachers and students.
3. Design and administration of a survey to acquire a clearer characterization of teachers' PCK in connection with the teaching of iteration, in particular regarding:
   (a) Instructional practice and guidelines;
   (b) Code examples and programming tasks discussed in class;
   (c) Assessment criteria.
4. Design and administration of an online survey to assess students' program comprehension in a set of tasklets addressing a variety of aspects related to iteration:
   (a) Specific flow-control features;
   (b) Interaction with different data types;
   (c) Mastery of the underlying computation model (*notional machine*);

(d) Higher-order thinking skills (abstraction and grasp of the relationships with a problem domain);

(e) Self-confidence when engaging in tasks calling for abstract thinking;

(f) Impact of dealing with flow-charts vs. textual code.

Main contributions

The main implications of this work are twofold, pertaining to the knowledge area — about the teaching and learning of iteration — as well as to the future perspectives — concerning both the instructional practice and the educational research.

Firstly, the data collected from teachers and students provide a clearer and more detailed overview concerning how the teaching of iteration is approached in the high school and of students' degree of understanding of iterative programs — at least as far as Computer Science education in the North-East of Italy is concerned (but we believe that there are no differences with high schools in other areas of the country, which in any case must comply with the same ministerial guidelines). Probably the samples could have been larger if the Covid-19 pandemic had not occurred, but we think that the indications emerging from the study are sufficiently representative of a general situation.

Secondly, the nature of the proposed tasks and the structure of the survey can lay the basis for further interesting developments. And indeed, in our opinion, the major contribution of this study is to be found in its potential to lay the groundwork for further research. More specifically, from a pedagogical perspective, we have started the construction of a "catalog" of programming tasks suitable to address different facets of the understanding of iteration. It can certainly be enriched by including additional examples and by covering other possible aspects, with the aim of providing a useful instrument for the teaching (and assessment) practice. Similarly, the student survey, and in particular its structure, can be the starting point to extend the investigation to other cohorts — conceivably after introducing a few refinements.

Organization of this document

In what follows we briefly describe the content of each chapter. Part of it is based on the publications listed in Appendix A.

Chapter 1 – This chapter presents a summary of the literature review. A number of studies discuss novices' difficulties with basic flow-control constructs, which concern both the "mechanical" features of the notional machine and the related logical aspects, ranging from the implications of loop conditions to a more abstract grasp of the underlying algorithms. Although these issues have not been extensively explored for pre-tertiary education, it is conceivable that they are even stronger in secondary school contexts. Moreover, a few contributions regarding teachers' PCK, related Content Representation frameworks, and Concept Inventories provided helpful suggestions and methodologies to be exploited in order to design the pilot investigations (teachers' interviews and students' preliminary survey).

Chapter 2 – This chapter discusses what has emerged from the two pilot investigations, that involved 20 experienced high school teachers of introductory programming (interviews) and 164 students (survey) from high schools spread across a vast area in the North-East of Italy. The survey included both questions on their subjective perception of difficulty and small tasks probing their understanding of iteration. The data collected from teachers and students confirm that iteration is a central programming concept and seem to indicate that the treatment of conditions and nested constructs are major sources of students' difficulties with iteration. It can also be observed that the examples that teachers usually choose to explain the iteration constructs tend to be quite stereotypical and elusive of some complexities intrinsic to the topic. This suggests that a richer and more varied "catalog" of examples may be useful to challenge students to deepen their understanding of iteration.

Chapter 3 – The subject of this chapter concerns the design and the results of the survey addressed to the teachers, aimed at a better understanding of their instructional practices to teach iteration. It explores the overall view of introductory programming, including its basic prerequisites, and, above all, the strategies to teach and assess the learning of iteration. The survey protocol is partly inspired by the approaches proposed in the literature to elicit the PCK. Besides a few suggestions drawn from the concrete practice of the teachers, an interesting outcome of the analysis of the answers of 21 respondents is that their approach to programming tends to be more practical than conceptual, but also process-oriented rather than product-oriented.

Chapter 4 – This chapter discusses the design and the results of the online survey where the students were required to engage in a set of small tasks of program comprehension, involving reading, tracing, explaining, evaluating skills. To begin with, the rationale behind the structure of the investigation instrument is presented: at its basis are outcomes arisen from the pilot study, concerning the understanding of iteration and the perception of self-confidence (to be evaluated for each task on a Likert scale). Then, the data collected from 225 high school students are analyzed in terms of the areas of learning addressed by the tasklets. In essence, the findings confirm that loops and conditionals can be potential sources of novices' misconceptions, but also provide a broader picture of the issues connected with iteration from different learning perspectives.

Conclusions – This final chapter summarizes the achievement of the project, discusses some implications for instructors and presents potential future directions of work, both to broaden the research scope and to develop useful instruments for the teacher.

Appendices – The document ends with five appendices: the first appendix reports the abstracts of the published papers on which part of this document is based; the second one lists the programming tasks included in the catalog; the third one outlines the role of Computer Science in the Italian upper secondary schools; the fourth one presents the teachers' pilot interview protocol; the fifth one the students' pilot survey protocol.

References

[Bec21] Brett A. Becker. "What Does Saying That 'programming is Hard' Really Say, and about Whom?" In: *Commun. ACM* 64.8 (2021), pp. 27–29. DOI: 10.1145/3469115.

[Cet15] Ibrahim Cetin. "Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective". In: *Canadian Journal of Science, Mathematics and Technology Education* 15.2 (Feb. 2015), pp. 155–170. DOI: 10.1080/14926156.2015.1014075.

[Cot06] Lucio Cottini. *La didattica metacognitiva.* 2006.

[CZP14] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. "Identifying Challenging CS1 Concepts in a Large Problem Dataset". In: *Proc. of the 45th ACM Tech. Symp. on Computer Science Education.* SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 695–700.

[Dij89] Edsger W. Dijkstra. "On the cruelty of really teaching computing science". English. In: *Communications Of The Acm* 32.12 (1989), pp. 1398–1404.

[DR08] Michael De Raadt. "Teaching programming strategies explicitly to novice programmers". PhD thesis. University of Southern Queensland, 2008.

[FAO10] José Luis Fernández Alemán and Youssef Oufaska. "SAMtool, a Tool for Deducing and Implementing Loop Patterns". In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education.* ITiCSE '10. New York, NY, USA: ACM, 2010, pp. 68–72. DOI: 10.1145/1822090.1822111.

[GM07] Anabela Gomes and Antonio Mendes. "Learning to program - difficulties and solutions". In: *International Conference on Engineering Education – ICEE.* Jan. 2007, pp. 283–287.

[Gom+20] Anabela Gomes et al. “Study methods in introductory programming courses”. In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. 2020, pp. 898–904. DOI: `10.1109/EDUCON45650.2020.9125228`.

[Hat12] John Hattie. *Visible learning for teachers: Maximizing impact on learning*. Routledge, 2012.

[HLR11] Orit Hazzan, Tami Lapidot, and Noa Ragonis. *Guide to Teaching Computer Science: An Activity-Based Approach*. 1st. Springer Publishing Company, Incorporated, 2011.

[Jen02] Tony Jenkins. “On the Difficulty of Learning to Program”. In: *Proceedings of the 3rd annual LTSN ICS Conference*. Loughborough, UK, 2002.

[Kac+10] Lisa C. Kaczmarczyk et al. “Identifying Student Misconceptions of Programming”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. New York, NY, USA: ACM, 2010, pp. 107–111.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A Study of the Difficulties of Novice Programmers”. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 14–18. DOI: `10.1145/1067445.1067453`.

[Lew+05a] Gary Lewandowski et al. “What novice programmers don’t know”. eng. In: *Proceedings of the first international workshop on computing education research*. ICER ’05. ACM, 2005, pp. 1–12.

[LR+18] Andrew Luxton-Reilly et al. “Introductory Programming: A Systematic Literature Review”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 55–106.

[Mir12] Claudio Mirolo. “Is Iteration Really Easier to Learn Than Recursion for CS1 Students?” In: *Proc. of the 9th Annual International Conference on International Computing Education Research*. ICER ’12. New York, NY, USA: ACM, 2012, pp. 99–104.

[PSS88] D.N. Perkins, Steve Schwartz, and Rebecca Simmons. “Instructional Strategies for the Problems of Novice Programmers”. In: *Teaching and Learning Computer Programming*. Ed. by Richard E. Mayer. New York, USA: Routledge, 1988, pp. 153–178. DOI: `10.4324/9781315044347`.

[QL17] Yizhou Qian and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review”. In: *ACM Trans. Comput. Educ.* 18.1 (Oct. 2017). DOI: 10.1145/3077618.

[Rep16] Alexander Repenning. “Transforming "Hard and Boring" into "Accessible and Exciting"”. In: *CoPDA@NordiCHI*. 2016.

[Rob19] Anthony V. Robins. “Novice Programmers and Introductory Programming”. In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 327–376. DOI: 10.1017/9781108654555.013.

[RRR03] Anthony Robins, Janet Rountree, and Nathan Rountree. “Learning and Teaching Programming: A Review and Discussion”. In: *Computer Science Education* 13.2 (2003), pp. 137–172.

[Shu05a] Lee S. Shulman. *Signature pedagogies.* 2005.

[Shu86] Lee S. Shulman. “Those Who Understand: Knowledge Growth in Teaching”. eng. In: *Educational Researcher* 15.2 (Feb. 1986), pp. 4–14.

[Woo06] Terry Wood. “Teacher Education Does Not Exist”. eng. In: *Journal of Mathematics Teacher Education* 9.1 (2006), pp. 1–3.

Chapter 1

Literature review

This chapter sets the background of our work by summarizing what emerged from the literature review. After introducing the main objectives of the review and how they have been pursued in Section 1.1, Section 1.2 addresses students' difficulties to learn basic programming concepts. Then, in Section 1.4 we consider students' specific difficulties with iteration, which are the focus of the present work. Section 1.5 provides an overview of topics regarding abstraction skills, which are clearly implied in program comprehension tasks. In particular, Section 1.5.1 explores a useful distinction between abstraction as a product and abstraction as a process, whereas Section 1.5.2 discusses the role of abstraction specifically in the context of iteration. Finally, Section 1.6 reviews work aimed at eliciting teachers' Pedagogical Content Knowledge (PCK), in particular in introductory programming, which has been the basis to set up the structure of the interviews and of the survey addressed to teachers.

1.1 Aims and scope of the review

With the aim of drawing a picture that is as comprehensive as possible of teaching and learning iteration, we approached the literature review with the following questions in mind:

- What learners' difficulties and misconceptions about the basic flow-control constructs, and in particular iteration, have been investigated?
- Which studies focus specifically on (upper) secondary instruction?
- What types of programming knowledge, specifically pertaining to the basic flow-control constructs, are addressed by existing Concept Inventories?

- Are there models of abstract thinking that could be used to explain students' understanding of iteration?
- How can interviews or surveys be structured in order to collect meaningful information on teachers' actual practice and pedagogical content knowledge (PCK)?

In this light, we will start by analyzing studies discussing novice programmers' basic difficulties and misconceptions, in particular among the contributions referenced in two broad reviews about the educational research in the field of programming, namely [RRR03] and [LR+18]. Moreover, we will look for the issues addressed by Concept Inventories of programming topics. As a next step, we will consider potential models to characterize the abstraction abilities which may play a role in program comprehension. Finally, we will conclude the review with a brief discussion of methodologies to elicit teachers' PCK relative to programming instruction.

1.2 Research on novice programmers

Students' difficulties to learn programming are well known to Computer Science educators, e.g. [DB86; RRR03; QL17; LR+18], and are also witnessed by the high drop out rates [Jen02; Lew+05b]. The reasons may be manifold, ranging from lack of problem solving skills to the need for accuracy and intensive practice. As mentioned in the introduction, programming is indeed *problem-solving intensive* [PSS88] and demands a significant amount of effort in several skill areas. According to Gomes and Mendes [GM07] it requires "not a single, but a set of skills", and students may fail to develop a viable model of the underlying *notional machine* [Sor13] or to be able to connect code execution with its functional purpose [Lis+06].

In particular Sorva [Sor13] ascribes several issues to students' misconceptions, ranging from syntax errors to misunderstandings about code execution — some of which may also be related to the habits and expectations of both teachers and learners [Jen02; RHG06]. In their analysis of recurrent problems faced by students in laboratory tasks, Robins et al. [RHG06] report as highly frequent those issues concerning the understanding of the "trivial mechanics" of code execution, the understanding of the task at hand, as well as "general matters of program design". As pointed out by Dijkstra [Dij89], learning to program is a slow and gradual process to accomplish, to which the teacher is required to grant adequate learning time.

Significant misconceptions are reported even for basic flow-control constructs such as conditionals and loops. Kaczmarczyk et al. [Kac+10], for instance, observed that "students misunderstand the process of `While`-loop iteration". Cherenkova et al. [CZP14] found that students' most common errors refer to Booleans, conditionals, loops and loops with conditionals. Furthermore, iteration and conditional expressions have indeed to be regarded as strictly connected [Kon19].

In addition, independent findings indicate that students can often be misled when the "else" branch of a conditional construct is not coded explicitly, e.g. [Van+10; IMW18]. Another problematic endeavor for novices concerns the indexed access to arrays, in particular in connection with iteration, as has also been confirmed by some recent studies [RDLR20; MS21].

Problems that can be solved at low levels of abstraction by tracing the code execution, for specific input data, are certainly among the most common programming tasks in which novices are required to engage. Ordinarily, tracing is deemed to be a basic ability "to build [...] higher-level comprehension skills upon" it [Lis+04], even though it is not a sufficient prerequisite in this respect [TL14a]. Nevertheless, this ability should not be taken for granted, since many students struggle, for instance, with tracing loops, especially `While`-loops [Lop+08]. At a higher level of abstraction, students are often asked to identify the purpose of a given program, i.e. to answer "Explain in Plain English" questions in the work of Lister and colleagues [Lis+06].

Jenkins [Jen02] has tried to explain part of students' difficulties by distinguishing between "deep" and "surface" learning styles. According to him, deep learners focus on gaining an understanding of the concept at hand, while surface learners tend to base their study on merely memorizing information. A particular approach to teaching can therefore influence the adoption of a deep versus surface learning style.

Lewandowski at al. [Lew+05b] have attempted to characterize novice programmers' knowledge in terms of "a mixture of well-formed, in-transition and muddled conceptual structures". In particular, they asked a sample of students to use *rag-bag categories* such as "don't know", "not sure", or "not applicable" to classify their (perceived) level of knowledge for a number of programming concepts.

Although most studies on the above areas focus on university-level courses, it is reasonable to expect that similar issues are even more significant at the upper secondary level of education.

1.3 Concept Inventories

Scientific Concept Inventories "are criterion-referenced tests that are usually designed to evaluate students' understanding of current accepted scientific concepts. These criterion-referenced assessments are developed around a set of recognized performance standards [...]" [Sch12].

According to Almstrum et al. [Alm+06, p. 133],

> a concept inventory (CI) is a validated assessment tool with a focus on misconceptions that students may have about fundamental concepts in a domain. Concept inventories are commonly created using a student-driven process that elicits students' perceptions and understandings about a given concept.
>
> A concept inventory has the following characteristics:
>
> - It is a reliable, validated assessment instrument.

- It focuses on common student misconceptions.
- It covers a specific domain, but is not a comprehensive instrument (i.e., it is not a final exam).
- It is composed of multiple-choice items.
- It is designed to require at most 30 minutes to complete.
- Its scope may or may not match the scope of the corresponding course. Thus, it could be necessary to develop more than one CI to cover the topics of a course such as CS1.
- It can be administered as a post-test (for example, at the end of the course). Some concept inventories can also be given as pretests, allowing "before and after" comparisons.
- It can be used by instructors to identify aspects of instruction that would benefit from change, to assess the impact of modifications, and to compare pedagogical approaches.
- It should not be used by college administrators to assess teaching performance or by individual instructors for determining grades for individual students.

> [...] The primary purpose for existing CIs has been to investigate the disparity between the concepts a student should be learning and what the student actually is learning. [...] These assessment instruments have provided benchmarks for comparing the effect of innovative teaching techniques on students' understanding of fundamental concepts [...].

Libarkin [Lib08] presents a comprehensive list of published concept inventories for science, which does not cover, however, repositories for Computer Science and information technology.

1.3.1 Concept Inventories in Computer Science

Almstrum et al. [Alm+06] outline a framework for creating a CIs for specific mathematical concepts. Our aim, here, is to find similar proposals relative to Computer Science and programming.

Goldman et al. [Gol+08; Gol+10], following Shaffer [Sch12], present a methodology for creating and validating a CI in Computer Science; the identified concepts are divided into the broader areas reported in Table 1.1.

Caceffo et al. [Cac+16], on the other hand, essentially expand the first area of Table 1.1, amounting to the main topics covered in an introductory programming course. More specifically, the concepts listed in Table 1.2 were identified via the analysis of exam assignments and interviews of instructors. The list thus elaborated presents most of the basic concepts included in a course syllabus both at university level and at high-school level. (The items are ordered in accordance with the related discussion in [Cac+16].)

Table 1.1: Programming fundamental topics.

1	Procedural Programming Concepts
2	Object Oriented Concepts
3	Algorithmic Design Concepts
4	Program Design Concepts

Table 1.2: Concept Inventory for Introductory Programming.

Function Parameter Use and Scope
Variables, Identifiers, and Scope
Recursion
Iteration
Structures
Pointers
Boolean expressions
Syntax vs. Conceptual Understanding

1.4 Students' difficulties with Iteration

Starting from the pioneering work on the cognitive implications of programming tasks in the early 1980s, e.g. [SBE83; DB86], empirical research has consistently shown that flow-control constructs such as conditionals and loops tend to be approached in stereotypical ways and are common sources of errors and misconceptions for novice learners [KD03; Kac+10; CZP14], especially when combined through nested constructs [Cet+20]. To put it in Winslow's words [Win96], in light of the evidence gained in more than a decade of investigations, if "almost any undergraduate can add a set of numbers or compute an average of a set of numbers; why can't over half of them write a loop to do the same operations?"

Soloway et al. [SBE83] analyzed student strategies for solving loop problems, and subsequent studies by Bonar and Soloway [BS83; BS85] seemed to indicate that misconceptions regarding the while-loop could be ascribed to natural language interpretations of the word "while". In particular, these authors suggested that novice programmers have fewer problems with post-test "repeat-until" loops, since the common-sense use of "until" better matches the actual behavior of such loops. Sleeman et al. [Sle+86] observed that the iteration constructs can confuse novice students, since they may misunderstand the scope of the loop, may be unable to recognize which lines will actually be repeated and/or how many times they will be executed. Du Boulay [DB86], on the other hand, reported that the treatment of loop-control variables may be problematic. Moreover, regarding the nesting of flow-control constructs, Pea [Pea86] remarked that students may mistakenly believe that a nested statement inside a loop is in connection with the loop condition.

More recently, several specific language-related problems observed by Robins et al. [RHG06] were related to loops. While analyzing the programming strategies to achieve simple tasks, in his PhD thesis, De Raadt [DR08] found that "less than half of students produced correct primed sentinel-controlled loops for the summing or counting or both". In this respect, De Raadt argued that students may deliberately be "led to practice application of particular strategies for these problems in the same way that an instructor might encourage students to use a particular language construct, such as a for loop", therefore indicating the possibility that teachers' insistence on some patterns can generate stereotypes.

Dehnadi [Deh09] investigated the mental mechanisms manifested when thinking about a simple iteration and suggested that novices can stumble "at an early stage, particularly when all the alternative kinds of iteration (while, do-while, for) are introduced at the same time. In fact iteration is conceptually difficult, and the proper treatment of iteration is mathematically complicated." In accordance with similar results by Simon et al. [Sim+06], Craig et al. [CPP12] confirmed students' preference for post-test loops, especially in the form of a terminating condition ("stop when") rather than a continuing condition ("continue while"). Moreover, Grover & Basu [GB17b] argue that students' difficulties are often related to "understanding how and when to terminate loops" and note that "Boolean AND/OR operators are often mistakenly interpreted", as previously pointed out by Herman et al. [Her+12].

A widespread source of students' difficulties is lack of strategic knowledge, in particular when they are required to choose a correct iteration construct to solve some specific problem. Related issues have been addressed, among others, by De Raadt [DR08], Simon [Sim13], and Fisler [Fis14]. The different level of mastery observed for downward (or down-counting) loops with respect to more stereotypical upward (up-counting) loops, pointed out e.g. in [KD03], can probably be ascribed to weaknesses related to plans. Additionally, widespread troubles can be engendered by the nesting of flow-control constructs [Gin04; MBŽ18; Cet+20], what led Cetin [Cet15] to propose a theoretical "action-process-object-schema" framework to analyze student's cognitive obstacles in this respect. Moreover, Koppelman and van Dijk [KD10] discuss potential causes of students' difficulties with nested loops and report about experiments where "novices did not usually recognize the need for a nested loop", or "if they did, they had problems separating the conditions that control both loops".

Other authors have pointed out the role of terminology or the potential mismatch between technical versus common-sense use of the implied words. According to Lewandowski et al. [Lew+05b, p. 11], for instance:

> When teaching programming, If-then-else is a frequently-used term while Choice is less likely to be mentioned with high frequency. Similarly, while one may teach iteration without frequently saying 'iteration', it is rare to teach

> recursion without using the term. Hearing the terms less frequently, students have fewer opportunities to consciously consider and integrate (consciously or subconsciously) them into their conceptual structures. This suggests instructors may be able to help students build their conceptual organization by using key abstract terms frequently.

Stefik et al. [SG11; SS13] focused their investigation on word choice and noted that "for" and "while", the most common keywords for loops, appear to be rather counter-intuitive to novices, as opposed to "repeat", which is deemed as more intuitive. In the context of primary education, on the other hand, Mannila et al. [Man+20] posit that children's "familiarity with loops can be explained by the word 'loop' being used in other non-programming contexts as well".

A number of educators have also addressed the potential impact of non-textual program representations on either preventing or diagnosing students' misconceptions. To mention some, Ma et al. [Ma+09] elaborated on the results of their investigation, revealing students' inappropriate models of conditionals and loops, and suggested that such constructs could be explained more effectively by means of visualization tools. Weintrop and Wilensky [WW15] deem that graphical programming can help students gain a deeper understanding of iteration. At the secondary school level, Rahimi et al. [RBH17] discuss the use of flow-chart analysis as an insightful instrument to identify students' misconceptions about algorithmic concepts, including conditionals, loops and plan composition. According to Mladenovič et al. [MBŽ18], on the other hand, K–12 "students misconceptions about loops are minimized when using a block-based programming language", namely Scratch (instead of Logo or Python), and "the differences become" even "more apparent" for complex tasks involving nested loops.

To conclude this section, we can observe that most research on the understanding of iteration has been carried out in relation to university introductory courses, whereas far less work addresses pre-tertiary education [TG10; Gol+08].

1.5 The Role of Abstraction

Notions of abstraction apply to a broad variety of thought processes. In some way, virtually every useful piece of learning implies some sort of abstraction *from* contingent experiences. Besides the cognitive implications of dealing with abstract vs. concrete entities [Ade85], several philosophical, mathematical, and scientific views of abstraction give prominence to the elimination of non-essential features in connection with some specific objectives. However, this may scratch only the surface of the matter.

Computer Science educators largely agree that abstraction skills play a major role in their field [Dij72; SW80; BLW01; Kra07; Haz08; Arm13; SA16; GP18a]. Jeannette Wing, for instance, claims that "[t]he most important and high-level thought process in computational thinking is the abstraction process", especially in that it "gives us the power to scale and deal with complexity" [Win11]. However, this concept is usually referred to in overly generalized terms, leading Bennedsen and Caspersen to assert that "no one has defined what is meant by abstraction" [BC06]. Indeed, a first problem we face in the attempt to provide a definition of "abstraction" is that, by this concept, different educators mean different things.

Although we can actually find a few attempts to provide accurate characterizations of abstraction both from a general cognitive standpoint and according to the objectives of specific disciplinary fields, it is still far from clear how the development of abstraction skills could be fostered and assessed. To put it in Verhoeff's words, "[s]aying that abstraction is important is one thing. Explaining what abstraction is and how to teach it are quite other matters" [Ver11]. In light of the lack of consensus about the precise meaning of abstraction, we can recognize with [Haz99] that there is at least some "agreement that the notion of abstraction can be examined from various perspectives, that certain types of concepts are more abstract than others, and that the ability 'to abstract' is an important skill."

The role of abstraction in computing is however very pervasive, sometimes referred to implicitly, but more often explicitly, as observed by [Eck+06]. From an educational viewpoint, the meanings assigned to the term "abstraction" in Computer Science — either explicitly or implicitly — are manifold, also depending on the subfield of interest. Some focus on the learning of theoretical aspects [AGE06; GA12] or draw from experiences in Mathematics [SH08; GB17a]; others point out the ability, peculiar to programming, of managing different levels of abstraction [Hab04; GB17a]. Still others reflect the goals of software engineering [HT05; Wan08; Ste18]. Hazzan and colleagues [Haz03; SH08], on the other hand, apply their "reducing abstraction" framework in order to analyze students' approach to computing tasks in terms of the polarity *abstract* vs. *concrete*.

Based on ISTE/CSTA operational definition of Computational Thinking [IST11], *abstraction* is one of the key abilities and it is defined in terms of "reducing complexity to define main ideas". Similarly, Csizmadia et al. [Csi+15] and Curzon et al. [Cur+19] refer to reducing unnecessary detail to "get at the essence". According to Rijke et al. [Rij+18], abstraction in the primary school context means "having the ability to determine which aspects are important and which are not". While applying computational thinking to approach biological problems, Peel and Friedrichsen [PF18] characterize abstraction as "simplifying information; displaying only the information that is needed". Moreover, Chaabi et al. [CAD19] have investigated the relationships between mathematical abstraction and computational thinking.

In the remainder of this text work we refer to abstraction as *reducing complexity to define main ideas* — in order to define general solution models —, following the ISTE/CSTA definition, eliminating non-essential details [Che18]. However, the characterization of 'abstraction' is not unique.

1.5.1 Abstraction as a process vs. abstraction as a product

A major concern, from an educational viewpoint, is the orientation between the polarities of "abstraction as a *process*" versus "abstraction as a *product*" i.e. the active or passive cognitive role while learning abstract concepts. The need to bring about suitable conditions in order for the students to be able to experience the *process* of abstraction has been put forward by several researcher with regard to learning mathematical concepts. In this respect, for instance, Mason [Mas89, p. 1] wrote that:

> [S]tudent's sense of abstract as *removed from* or *divorced from* reality (or perhaps, more accurately, from meaning, since our reality consists in that which we find meaningful) [...] arises because there has been little or no participation in the process of abstraction [...]. Despite current emphasis on exploration and investigation, many students may still not experience the shift of abstraction unless they receive explicit assistance.

Similarly, Dreyfus [Dre91] claimed that students "have been taught the products of the activity of scores of mathematicians in their final form, but they have not gained insight into the processes that have led mathematicians to create these products."

Later, White and Mitchelmore [WM99] dug a little more deeply into the implications of process vs. product approaches to introduce new mathematical concepts. They remarked that most teaching practices introduce abstractions as ready-to-use products, in a (context-free) *abstract-apart* way, rather than focusing on the context-situated processes which could give rise to them and so promoting the learning of what they called *abstract-general* concepts. As a result, in their opinion, "concepts and procedures learnt in an abstract-apart manner are limited because they can only be applied in situations which look suitably similar to the context-free way in which they were learnt".

The dichotomy abstraction as a process versus abstraction as a product applies also to the basic forms of *procedural*, *data*, and *control* abstraction, so pervasive in the programming activity, with the additional burden that the learner is often required to conceive from scratch suitable abstractions of these sorts as part of the tasks assigned to them. Although the opportunity of being exposed to abstraction as a process is clearly valuable for a solid learning, these kinds of abstraction tend to be dealt with as just a repertory of technical notions, and risk being perceived by students as mere labels of programming language constructs.

As opposed to most other fields (and namely Mathematics), where abstractions are chosen by the teacher within a pre-established repertory and viewed as learning objectives, in programming the learners themselves are expected to envisage suitable abstractions based on the known solutions of similar tasks they were exposed to. However, although abstractions "are what give software elegance", we "can teach the use of pre-packaged abstractions", but it "is more difficult to teach the self-awareness necessary for inventing new abstractions" [AS08].

In this respect, Hazzan [Haz08, p. 41] suggests that teachers should be explicit when they use abstraction:

> [F]or example, with respect to abstraction applied in front of the class, instructors can specifically make statements such as: "I am ignoring this aspect here because...", "Now, let's move one level of abstraction down and elaborate on...", "Similarly to what we did last week, abstraction is expressed here because...", "If we hadn't used abstraction, the solution would have been...", and so on. This can be done, for example, while developing a solution in stages in front of the class, specifying how abstraction is expressed and how it guides the solution process, instead of presenting the students with complete solutions.

1.5.2 Relationships between abstraction and iteration

A variety of abstraction processes are intrinsically involved in any meaningful programming task, due to the very nature of this kind of activity, which builds upon subsequent steps of abstraction from the details of the underlying processes or phenomena. Nevertheless, it often appears to be taken for granted that almost every programming activity should imply and/or develop abstract thinking abilities, without being aware of this tacit assumption.

According to the model developed by a group of "Thought Leaders" to incorporate computational thinking in K–12 education [BS11], the use of conditionals and loops is part of the characterization of the core *abstraction* concept, and a number of works explore the relationships between abstraction and iterations from a variety of perspectives. Abbott and Sun [AS08], for instance, see iteration in terms of *abstraction by parameterization*, in that "the while-loop construct [...] abstracts the notion of iteration and by providing slots into which a Boolean expression and a code segment may be inserted makes iteration available generically". Furthermore, they note that any implied activity has both *top-down* as well as *bottom-up features*: it is top-down since the programmer starts from the idea of some specific purpose to achieve; it is bottom-up since the code unit results from assembling existing components.

The abstraction ability in connection with iteration is addressed by Lister et al. [Lis+06]. Based on think-aloud responses collected from students (novices) and educators (experts) while engaging in a small code reading task, Lister and colleagues found that "educators tended to articulate an abstraction of the loop structure" (e.g., *"going backward through these arrays", "we're starting from the high end", "It looks like the code is assuming the arrays are in sorted order from smallest to largest"*) or of portions of it (e.g., *"I'm always decrementing the index of the bigger one"*), whereas "students generally articulated nothing more than the presence of a loop, and sometimes also a literal statement about the terminating condition." In SOLO terms [BFC82], educators usually thought at *relational* and sometimes even *extended abstract* levels; by contrast, students' explanations tended to be restricted to the *multistructural* level. In summary, the authors concluded that [Lis+06, p. 120]:

> It is apparent that, even when initially hand executing the code, most educators are actively seeking to abstract beyond the concrete code. In contrast, most novices did not seek to abstract.

Gries [Gri02] and Koppelman & van Dijk [KD10] advocate the explicit use of abstraction in order to deal with nested loops in a cleaner way. Koppelman and van Dijk, in particular, suggest that the levels of abstraction relative to the outer and inner loops could be separated by introducing a function that "hides" the nested construct at the higher level. Quite peculiarly, Ade-Ibijola et al. [AIES14] consider the potential of "narrative abstractions" of loops to support novices' program comprehension.

A few authors have also proposed small tasks intended to compel students to think of iteration constructs at a higher abstraction level (than that implied by simply tracing the code), but avoiding the additional cognitive load of some specific problem to solve. Examples of similar challenges include *reversibility* tasks [TL14b; MIS20] and *equivalence* tasks [IM20].

Reversibility is a property of a program or function that indicates it could be brought back to its original state. It is a topic that few scholars explored, and only rarely in high school. Recently, papers have been presented on the subject, e.g. Ginat & Armoni [GA06] consider the central role of "reverse thinking" in Computer Science, whilst Teague & Lister [TL14b] investigate to what degree novice programmers manifest the ability to work with this concept of reversibility. Teague & Lister started from the assumption that Piaget [Pia+69] had identified reversibility as an indicator of the ability to reason at a concrete operational level. Their results suggested "that many students remain at the sensorimotor and pre-operational levels because all the instruction they receive is at the concrete operational level". They conclude the analysis arguing that when students worked with concepts like reversibility then could reason abstractly.

Izu, Weerasinghe and Pope [IWP16] have shown that only a small percentage of students are able to correctly manage the problem of reversibility.

Furthermore, Izu et al. [IPW17] argued that "reasoning about reversibility requires students to have a mental model of the state, thus they should reason about program behavior as a whole, compared with reasoning about concrete cases using testing and tracing", confirming that students often fail to correctly reason about reversibility.

Several works have proposed tasks that require verifying reversibility or writing the reversal code [Lis11; TL14b; IMW18; MI19]. Moreover, Izu et al. [IMW18] suggested that the concept of reversibility could be a useful resource for educators to assess and develop students' understanding of program behavior.

1.6 Teachers' Pedagogical Content Knowledge

Teaching is a highly complex activity. In fact, whatever the subject, the learning process is cognitively demanding and the teacher must apply knowledge from multiple domains (Resnick [RM87], Leinhardt and Greeno [LGG86], Wilson, Shulman and Richert [WSR87]). Teachers with more differentiated and integrated knowledge are therefore better equipped to teach than those whose knowledge is specialized or fragmented (Magnusson, Borko and Krajcik [MKB99]).

In the attempt to characterize the instructional strategies that experienced teachers resort to in their practice within a specific subject, Shulman [Shu86] introduced the notion of *Pedagogical Content Knowledge* (PCK), namely the "blending of content and pedagogy into an understanding of how particular aspects of subject matter are organized, adapted, and represented for instruction".

Abell [Abe08] argued that "PCK is not merely the amount of knowledge in a number of component categories, it is also about the quality of that knowledge and how it is put into action". Therefore, it is possible to differentiate between "declarative PCK or knowing that" and dynamic forms of PCK "that cover teachers' activities during a lesson, for example, if a teacher is able to react appropriately to students' questions and mistakes" (Schmelzing et al. [Sch+13]). Both approaches have been investigated in the field, for example, by Alonzo & Kim [AK16], Gunckel et al. [GCS18] and Nijenhuis-Voogt at al. [NV+21].

The usual instruments to elicit teachers' PCK are interviews, which can be conducted in various ways, but often follow a semi-structured plan, such as that built around the eight standard questions of the Content Representation (CoRe) format [MKB99; LMB08], meant to capture teachers' knowledge about key ideas in connection with the topic at hand.

Similar interviews are an established research tool in Mathematics and in Physics, see e.g. Erlwanger [Erl73] and Ginsburg [Gin97]. Although less common in Computer Science education, we can however find a few attempts to investigate teachers' PCK in the areas of computing [Bar+14] and, more specifically, programming [BSS13; Sae+11; Bar+15]. We will next consider in further detail Shulman's guidelines to explore teachers' PCK and the use of the CoRe format, which have been insightful to prepare the face-to-face interviews discussed in chapter 2.

1.6.1 Questions to elicit PCK

Pedagogical content knowledge is an *"amalgam"* of content and pedagogy [Shu05a; Shu05b]. In Shulman's words [Shu86, p. 9]:

> Within the category of pedagogical content knowledge I include, for the most regularly taught topics in one's subject area, the most useful forms of representation of those ideas, the most powerful analogies, illustrations, examples, explanations, and demonstrations — in a word, the ways of representing and

> formulating the subject that make it comprehensible to others. [...]
> Pedagogical content knowledge includes an understanding of what makes the learning of specific topics easy or difficult: the conceptions and preconceptions that students of different ages and backgrounds bring with them to the learning of those most frequently taught topics and lessons.

In general, according to Magnusson's et al. model for science teaching [MKB99], the following components of PCK can be identified:

(a) Orientation to teaching science,
(b) Knowledge of science curriculum,
(c) Knowledge of students' understanding of science,
(d) Knowledge of instructional strategy,
(e) Knowledge of assessment of scientific literacy.

It is common to refer to core teaching topics, such as concepts, principles, methodologies, etc., as *Big Ideas* — see e.g. [LMB04]. Identifying suitable questions to investigate the pedagogical treatment of each big idea can help to characterize different teaching styles. The questions listed in Table 1.3, in particular, result from Grossman's approach [Gro89], revised by Saeli et al. [Sae+11, p. 76], and allow to define the PCK of a specific subject:

Table 1.3: Questions to elicit the PCK for a specific subject.

Why teach ... ?
What should be taught?
What are the learning difficulties?
How should the topic be taught?

None of the above questions, however, focuses on assessment, whereas the assessment of students' learning is becoming more and more significant for its implications to direct the instructional practice. In *our* perspective, it is then important to cover the following points as well (Table 1.4):

Table 1.4: Questions about the assessment of the learning of a Big Idea.

How should this topic be assessed?
What could be a fair method to assess this topic?
What aspects of the topic should be assessed?

In addition, we can mention the questions raised by Schulte and Bennedsen [SB06] in order to explore how different topics covered in introductory programming courses are taught:

1. What is the importance, difficulty and current teaching level of some often discussed learning topics in introductory programming courses? (Important teaching issues)
2. What is the relevance of areas emerged from the discussion in the pre-Object-Oriented era — explicated by the five domains described by du Boulay[1] [DB86] (Role of areas)
3. How important is teaching and learning object interaction evaluated in introductory programming courses? (Role of object interaction)

1.6.2 Content Representation (CoRe) questions

The Content Representation (CoRe) format is an instrument to investigate teachers' PCK of a specific topic (Loughran et al. [LMB04]). It captures the key ideas connected to the topic, and characterizes the teachers' knowledge about each idea through the 8 standard questions outlined in Table 1.5. The questions cover the above components addressed by Magnusson et al. [MKB99]: question 0 is somehow linked to point (a), questions 1, 2 and 3 refer to point (b), 4 and 5 to point (c), 6 and 7 to point (d), and 8 to point (e).

Table 1.5: CoRe questions.

0.	What are important ideas/concepts ('Big Ideas') concerning this topic?
	For each Big Idea:
1.	What do you intend the students to learn about this Big Idea?
2.	Why is it important for the students to know this Big Idea?
3.	What else do you know about this Big Idea (and you don't intend students to know yet)?
4.	What are the difficulties/limitations connected with the teaching of this Big Idea?
5.	What knowledge about students' thinking influences your teaching of this Big Idea?
6.	Which factors influence your teaching of this Big Idea?
7.	What are your teaching methods (any particular reasons for using these to engage with this Big Idea)?
8.	What are your specific ways of assessing students' understanding or confusion around this Big Idea?

Loughran et al. [LMB08] originally introduced the CoRe format as an interview tool. Its use is essentially related to the identification of a topic, then questions 1–8 of Table 1.5 are asked.

[1]The five areas identified by du Boulay as potential sources of students' difficulties are: *orientation*, *notional machine*, *notation*, *structures* and *pragmatics*.

Drawing inspiration from the work of Buchholz, Saeli and colleagues [BSS13; Sae+11; Sae12], for instance, Big Ideas for introductory programming can be chosen among those listed in Table 1.6, as proposed by [BSS13; Sae+11; Sae12]:

Table 1.6: Big Ideas for introductory programming.

Control Structures: loops, conditions and sequence
Functions, procedures and methods
Algorithms
Variables and constants
Parameters
Data structures
Decomposition
Re-usability
Arrays
Logical thinking
Formal languages: grammar and syntax

Additional relevant topics within the sphere of introductory programming can be extrapolated from a number of collective studies, such as: Lister et al. [Lis+12], Barendsen et al. [Bar+15], Luxton-Reilly et al. [LR+17a], Izu et al. [Izu+19].

This instrument has, however, some limitations. Saeli [Sad10] argued that "teachers sometimes have no answers to the questions, for example for 'problem-solving skills"'. Furthermore, Saeli [Sae12] pointed out that "teachers from different countries could also report different teaching methods or teaching beliefs".

1.7 Summary of the review

The central theme of this work is the teaching and learning of iteration in the context of the Italian upper-secondary school, which we are trying to explore from the perspectives of both educators and students. To this aim, we have reviewed the literature about a range of topics, covering the difficulties faced by novices, the reported insights on potential cognitive challenges and misconceptions, the instruments to explore instructional pedagogies/approaches in teachers' practice. All this material will contribute to set out the investigations presented in the next chapters.

In the attempt to summarize the major insights gained from the review, we recall the following points:

– To begin with, iteration is generally regarded as a central topic for introductory programming.

– Conditionals and loops are frequent sources of difficulties for novices. More specifically, the known issues include: Boolean operators in conditions, loop termination conditions, loop control variables, down-counting loops, nested constructs within loops. Besides, when considering higher-level thinking, often the lack of adequate strategic knowledge (plans) is pointed out. It is also observed that novices tend to be more at ease with some loop structures than with others (e.g., exit condition vs. continue condition).

– Most of the studies documented in the literature were conducted in the context of (undergraduate) CS0/CS1 courses, whereas the state of affairs in the high school does not yet appear to have been extensively investigated.

– The centrality of the theme is also witnessed by the fact that iteration and Boolean expressions are addressed in one of the (few) available Concept Inventories for introductory programming.

– Apparently, mastery of conditionals and loops — i.e. to go beyond the mere ability to trace code execution — implies the exertions of abstraction skills. According to some educators, for instance, iteration can be viewed as a form of abstraction by parameterization; moreover, in order to get a deep understanding of nested loops it is necessary to separate and move between different levels of abstraction. Tasks asking to determine equivalence or reversibility of (small) programs may help to foster the development of abstract thinking skills and, more in general, students should be guided to experience abstraction as a process, rather than as a product.

– The Content Representation format is an inspiring model to prepare the interviews aimed at eliciting teachers' pedagogical content knowledge.

References

[Abe08] Sandra K. Abell. “Twenty Years Later: Does pedagogical content knowledge remain a useful idea?” eng. In: *International journal of science education* 30.10 (2008), pp. 1405–1416.

[Ade85] Beth Adelson. “Comparing Natural and Abstract Categories: A Case Study from Computer Science”. In: *Cognitive Science* 9.4 (1985), pp. 417–430. DOI: `https://doi.org/10.1207/s15516709cog0904\_3`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog0904_3`.

[AGE06] Michal Armoni and Judith Gal-Ezer. “Reduction – an abstract thinking pattern: the case of the computational models course”. In: *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2006, pp. 389–393. DOI: `http://doi.acm.org/10.1145/1121341.1121461`.

[AIES14] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. “Abstracting and Narrating Novice Programs Using Regular Expressions”. In: *Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology*. SAICSIT ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 19–28. DOI: 10.1145/2664591.2664601.

[AK16] Alicia C. Alonzo and Jiwon Kim. “Declarative and dynamic pedagogical content knowledge as elicited through two video-based interview methods”. eng. In: *Journal of research in science teaching* 53.8 (2016), pp. 1259–1286.

[Alm+06] Vicki Almstrum et al. “Concept Inventories in Computer Science for the Topic Discrete Mathematics”. In: *ACM SIGCSE Bulletin Inroads* 38 (Dec. 2006), pp. 132–145. DOI: 10.1145/1189136.1189182.

[Arm13] Michal Armoni. “On teaching abstraction in computer science to novices”. In: *Journal of Computers in Mathematics and Science Teaching* 32.3 (July 2013), pp. 265–284.

[AS08] Russ Abbott and Chengyu Sun. “Abstraction abstracted”. en. In: *Proceedings of the 2nd international workshop on The role of abstraction in software engineering - ROA ’08*. Leipzig, Germany: ACM Press, 2008, p. 23. DOI: 10.1145/1370164.1370171.

[Bar+14] Erik Barendsen et al. “Eliciting computer science teachers’ PCK using the Content Representation format: Experiences and future directions”. In: *Proceedings of the 6th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP’14) – Teaching and Learning Perspectives*. Ed. by Yasemin Gülbahar, Erinç Karataş, and Müge Adnan. Vol. 8730. Istanbul, Turkey: Ankara University Press, Sept. 2014, pp. 71–82.

[Bar+15] Erik Barendsen et al. “Concepts in K-9 Computer Science Education”. In: *Proceedings of the 2015 ITiCSE on Working Group Reports*. ITICSE-WGR ’15. New York, NY, USA: ACM, 2015, pp. 85–116. DOI: 10.1145/2858796.2858800.

[BC06] Jens Bennedsen and Michael E. Caspersen. “Abstraction Power in Computer Science Education”. In: *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2006*. University of Sussex, Brighton, UK, Sept. 2006.

[BFC82] John Biggs and Kevin F Collis. “Evaluating the Quality of Learning: the SOLO Taxonomy”. In: *SERBIULA (sistema Librum 2.0)* (Jan. 1982).

[BLW01] Paolo Bucci, Timothy J. Long, and Bruce W. Weide. “Do we really teach abstraction?” In: *SIGCSE ’01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*. New York, NY, USA: ACM, 2001, pp. 26–30. DOI: 10.1145/364447.364531.

[BS11] Valerie Barr and Chris Stephenson. “Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community?” In: *ACM Inroads* 2 (Mar. 2011). DOI: 10.1145/1929887.1929905.

[BS83] Jeffrey Bonar and Elliot M. Soloway. “Uncovering principles of novice programming”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1983, pp. 10–13.

[BS85] Jeffrey Bonar and Elliot M. Soloway. “Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers”. In: *Human-Computer Interaction* 1 (June 1985), pp. 133–161. DOI: 10.1207/s15327051hci0102_3.

[BSS13] Malte Buchholz, Mara Saeli, and Carsten Schulte. “PCK and reflection in computer science teacher education”. In: *ACM International Conference Proceeding Series* (Nov. 2013). DOI: 10.1145/2532748.2532752.

[Cac+16] Ricardo Caceffo et al. “Developing a Computer Science Concept Inventory for Introductory Programming”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. New York, NY, USA: ACM, 2016, pp. 364–369. DOI: 10.1145/2839509.2844559.

[CAD19] Hasnaa Chaabi, Amina Azmani, and Juan Manuel Dodero. “Analysis of the relationship between computational thinking and mathematical abstraction in primary education”. In: *Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality*. 2019, pp. 981–986.

[Cet15] Ibrahim Cetin. “Student’s Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective”. In: *Canadian Journal of Science, Mathematics and Technology Education* 15.2 (Feb. 2015), pp. 155–170. DOI: 10.1080/14926156.2015.1014075.

[Cet+20] Ibrahim Cetin et al. “Teaching Loops Concept through Visualization Construction”. In: *Informatics in Education-An International Journal* 19.4 (2020), pp. 589–609.

[Che18] Eugenia Cheng. *The Art of Logic: How to Make Sense in a World that Doesn’t*. Profile, 2018.

[CPP12] Michelle Craig, Sarah Petersen, and Andrew Petersen. "Following a Thread: Knitting Patterns and Program Tracing". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education.* SIGCSE '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 233–238. DOI: 10.1145/2157136.2157204.

[Csi+15] Andrew Csizmadia et al. "Computational thinking - a guide for teachers". In: *Computing At School* (Jan. 2015).

[Cur+19] Paul Curzon et al. "Computational thinking". In: *The Cambridge Handbook of Computing Education Research* (2019), pp. 513–546.

[CZP14] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. "Identifying Challenging CS1 Concepts in a Large Problem Dataset". In: *Proc. of the 45th ACM Tech. Symp. on Computer Science Education.* SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 695–700.

[DB86] Benedict Du Boulay. "Some Difficulties of Learning to Program". In: *Journal of Educational Computing Research* 2 (Jan. 1986), pp. 57–73.

[Deh09] Saeed Dehnadi. "A cognitive study of learning to program in introductory programming courses." PhD thesis. Middlesex University, 2009.

[Dij72] Edsger W. Dijkstra. "The Humble Programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[Dij89] Edsger W. Dijkstra. "On the cruelty of really teaching computing science". English. In: *Communications Of The Acm* 32.12 (1989), pp. 1398–1404.

[DR08] Michael De Raadt. "Teaching programming strategies explicitly to novice programmers". PhD thesis. University of Southern Queensland, 2008.

[Dre91] Tommy Dreyfus. "Advanced Mathematical Thinking Processes". In: *Advanced Mathematical Thinking.* Ed. by David Tall. Dordrecht: Springer Netherlands, 1991, pp. 25–41. DOI: 10.1007/0-306-47203-1_2.

[Eck+06] Anna Eckerdal et al. "Putting threshold concepts into context in computer science education". In: *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education.* ITICSE '06. New York, NY, USA: ACM, 2006, pp. 103–107.

[Erl73] Stanley H. Erlwanger. "Benny's conception of rules and answers in IPI mathematics". In: *Journal of Children's Mathematical Behaviour 1, 2, Autumn* (1973), pp. 7–26.

[Fis14] Kathi Fisler. “The Recurring Rainfall Problem”. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 35–42. DOI: 10.1145/2632320.2632346.

[GA06] David Ginat and Michal Armoni. “Reversing: An Essential Heuristic in Program and Proof Design”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’06. New York, NY, USA: ACM, 2006, pp. 469–473. DOI: 10.1145/1121341.1121488.

[GA12] David Ginat and Ronnie Alankry. “Pseudo Abstract Composition: The Case of Language Concatenation”. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 28–33. DOI: 10.1145/2325296.2325307.

[GB17a] David Ginat and Yoav Blau. “Multiple Levels of Abstraction in Algorithmic Problem Solving”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 237–242. DOI: 10.1145/3017680.3017801.

[GB17b] Shuchi Grover and Satabdi Basu. “Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 267–272. DOI: 10.1145/3017680.3017723.

[GCS18] Kristin L. Gunckel, Beth A. Covitt, and Ivan Salinas. “Learning progressions as tools for supporting teacher content knowledge and pedagogical content knowledge about water in environmental systems”. In: *Journal of Research in Science Teaching* 55.9 (2018), pp. 1339–1362. DOI: https://doi.org/10.1002/tea.21454. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/tea.21454.

[Gin04] David Ginat. “On Novice Loop Boundaries and Range Conceptions”. In: *Computer Science Education* 14 (Sept. 2004), pp. 165–181. DOI: 10.1080/0899340042000302709.

[Gin97] Herbert Ginsburg. *Entering the Child’s Mind: The Clinical Interview In Psychological Research and Practice*. Cambridge books online. Cambridge University Press, 1997.

[GM07] Anabela Gomes and Antonio Mendes. “Learning to program - difficulties and solutions”. In: *International Conference on Engineering Education – ICEE*. Jan. 2007, pp. 283–287.

[Gol+08] Ken Goldman et al. “Identifying important and difficult concepts in introductory computing courses using a delphi process”. eng. In: *Proceedings of the 39th SIGCSE technical symposium on computer science education*. SIGCSE ’08. ACM, 2008, pp. 256–260.

[Gol+10] Ken Goldman et al. “Setting the Scope of Concept Inventories for Introductory Computing Subjects”. eng. In: *ACM Transactions on Computing Education* 10.2 (2010).

[GP18a] Shuchi Grover and Roy Pea. “Computational Thinking: A Competency Whose Time Has Come”. In: *Computer Science Education: Perspectives on teaching and learning in school*. Ed. by S. Sentance, E. Barendsen, and S. Carsten. London, UK: Bloomsbury Academic, 2018, pp. 19–38.

[Gri02] David Gries. “Where is Programming Methodology These Days?” In: *SIGCSE Bull.* 34.4 (Dec. 2002), pp. 5–7. DOI: 10.1145/820127.820129.

[Gro89] Pamela L. Grossman. “A study in contrast: sources of pedagogical content knowledge for secondary English”. In: *Journal of Teacher Education* 40.5 (1989), pp. 24–31.

[Hab04] Bruria Haberman. “High-School Students’ Attitudes Regarding Procedural Abstraction”. In: *Education and Information Technologies* 9.2 (May 2004), pp. 131–145. DOI: 10.1023/B:EAIT.0000027926.99053.6f.

[Haz03] Orit Hazzan. “How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science”. In: *Computer Science Education* 13.2 (2003), pp. 95–122. DOI: 10.1076/csed.13.2.95.14202.

[Haz08] Orit Hazzan. “Reflections on Teaching Abstraction and Other Soft Ideas”. In: *SIGCSE Bull.* 40.2 (June 2008), pp. 40–43. DOI: 10.1145/1383602.1383631.

[Haz99] Orit Hazzan. “Reducing Abstraction Level When Learning Abstract Algebra Concepts”. In: *Educational Studies in Mathematics* 40.1 (1999), pp. 71–90.

[Her+12] Geoffrey L. Herman et al. “Describing the What and Why of Students’ Difficulties in Boolean Logic”. In: *ACM Trans. Comput. Educ.* 12.1 (2012). DOI: 10.1145/2133797.2133800.

[HT05] Orit Hazzan and James Tomayko. “Reflection and abstraction in learning software engineering’s human aspects”. In: *Computer* 38 (June 2005), pp. 39–45. DOI: 10.1109/MC.2005.200.

[IM20] Cruz Izu and Claudio Mirolo. “Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers”. In: *Proc. of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’20. New York, NY, USA: ACM, 2020, pp. 466–472.

[IMW18] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. “Novice Programmers’ Reasoning About Reversing Conditional Statements”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE ’18. New York, NY, USA: ACM, 2018, pp. 646–651. DOI: 10.1145/3159450.3159499.

[IPW17] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. “On the Ability to Reason About Program Behaviour: A Think-Aloud Study”. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’17. New York, USA: ACM, 2017, pp. 305–310. DOI: 10.1145/3059009.3059036.

[IST11] ISTE/CSTA Steering Committee. *Computational Thinking Teacher Resources, 2nd ed.* ISTE/CSTA. retrieved: april 2022. 2011.

[IWP16] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. “A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research.* ICER ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 251–259. DOI: 10.1145/2960310.2960324.

[Izu+19] Cruz Izu et al. “Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories”. In: *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education.* ITiCSE-WGR ’19. New York, NY, USA: ACM, 2019, pp. 27–52.

[Jen02] Tony Jenkins. “On the Difficulty of Learning to Program”. In: *Proceedings of the 3rd annual LTSN ICS Conference.* Loughborough, UK, 2002.

[Kac+10] Lisa C. Kaczmarczyk et al. “Identifying Student Misconceptions of Programming”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education.* SIGCSE ’10. New York, NY, USA: ACM, 2010, pp. 107–111.

[KD03] Amruth Kumar and Garrett Dancik. “A tutor for counter-controlled loop concepts and its evaluation”. In: *33rd Annual Frontiers in Education, 2003. FIE 2003.* Vol. 1. Nov. 2003, T3C–7. DOI: 10.1109/FIE.2003.1263331.

[KD10] Herman Koppelman and Betsy van Dijk. “Teaching Abstraction in Introductory Courses”. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 174–178. DOI: 10.1145/1822090.1822140.

[Kon19] Siu-cheung Kong. “Components and Methods of Evaluating Computational Thinking for Fostering Creative Problem-Solvers in Senior Primary School Education”. In: *Computational thinking education.* Springer, Singapore, May 2019, pp. 119–141. DOI: 10.1007/978-981-13-6528-7_8.

[Kra07] Jeff Kramer. “Is abstraction the key to computing?” In: *Commun. ACM* 50 (Apr. 2007), pp. 36–42. DOI: 10.1145/1232743.1232745.

[Lew+05b] Gary Lewandowski et al. “What Novice Programmers Don’T Know”. In: *Proceedings of the First International Workshop on Computing Education Research.* ICER ’05. New York, NY, USA: ACM, 2005, pp. 1–12. DOI: 10.1145/1089786.1089787.

[LGG86] Gaea Leinhardt and James G. Greeno. “The Cognitive Skill of Teaching”. In: *Journal of Educational Psychology* 78 (Apr. 1986), pp. 75–95. DOI: 10.1037/0022-0663.78.2.75.

[Lib08] Julie Libarkin. *Concept Inventories in Higher Education Science.* Jan. 2008.

[Lis+04] Raymond Lister et al. “A Multi-national Study of Reading and Tracing Skills in Novice Programmers”. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education.* ITiCSE-WGR ’04. New York, NY, USA: ACM, 2004, pp. 119–150. DOI: 10.1145/1044550.1041673.

[Lis+06] Raymond Lister et al. “Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy”. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.* ITICSE ’06. New York, NY, USA: ACM, 2006, pp. 118–122.

[Lis11] Raymond Lister. “Concrete and other neo-piagetian forms of reasoning in the novice programmer”. In: *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), pp. 9–18.

[Lis+12] Raymond Lister et al. “Toward a Shared Understanding of Competency in Programming: An Invitation to the BABELnot Project”. In: *Proceedings of the 14th Australasian Computing Education Conference (ACE 2012).* Ed. by Michael de Raadt and Angela Carbone. RMIT University, Melbourne: Australian Computer Society, Jan. 2012.

[LMB04] John Loughran, Pamela Mulhall, and Amanda Berry. “In Search of Pedagogical Content Knowledge in Science: Developing Ways of Articulating and Documenting Professional Practice”. In: *Journal of Research in Science Teaching* 41 (Apr. 2004), pp. 370 –391. DOI: `10.1002/tea.20007`.

[LMB08] John Loughran, Pamela Mulhall, and Amanda Berry. “Exploring Pedagogical Content Knowledge in Science Teacher Education”. In: *International Journal of Science Education - INT J SCI EDUC* 30 (Aug. 2008), pp. 1301–1320. DOI: `10.1080/09500690802187009`.

[Lop+08] Mike Lopez et al. “Relationships Between Reading, Tracing and Writing Skills in Introductory Programming”. In: *Proc. 4th Int. Workshop on Comput. Educ. Research.* ICER ’08. New York, USA: ACM, 2008, pp. 101–112.

[LR+17a] Andrew Luxton-Reilly et al. “Developing Assessments to Determine Mastery of Programming Fundamentals”. In: *Proceedings of the 2017 ITiCSE Conference on Working Group Reports.* ITiCSE-WGR ’17. New York, USA: ACM, 2017, pp. 47–69. DOI: `10.1145/3174781.3174784`.

[LR+18] Andrew Luxton-Reilly et al. “Introductory Programming: A Systematic Literature Review”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 55–106.

[Ma+09] Linxiao Ma et al. “Improving the Mental Models Held by Novice Programmers Using Cognitive Conflict and Jeliot Visualisations”. In: *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 166–170. DOI: `10.1145/1562877.1562931`.

[Man+20] Linda Mannila et al. “Programming in Primary Education: Towards a Research Based Assessment Framework”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education.* WiPSCE ’20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: `10.1145/3421590.3421598`.

[Mas89] John Mason. “Mathematical abstraction as the result of a delicate shift of attention”. In: *Learn. Math.* 9.2 (1989), pp. 2–8.

[MBŽ18] Monika Mladenovic, Ivica Boljat, and Žana Žanko. “Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level”. In: *Education and Information Technologies* 23 (July 2018), pp. 1483–1500. DOI: 10.1007/s10639-017-9673-3.

[MI19] Claudio Mirolo and Cruz Izu. “An Exploration of Novice Programmers’ Comprehension of Conditionals in Imperative and Functional Programming”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 436–442. DOI: 10.1145/3304221.3319746.

[MIS20] Claudio Mirolo, Cruz Izu, and Emanuele Scapin. “High-School Students’ Mastery of Basic Flow-Control Constructs through the Lens of Reversibility”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education.* WiPSCE ’20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421603.

[MKB99] Shirley Magnusson, Joseph Krajcik, and Hilda Borko. “Nature, Sources, and Development of Pedagogical Content Knowledge for Science Teaching”. In: *Examining Pedagogical Content Knowledge.* Ed. by Julie Gess-Newsome and NormanG. Lederman. Vol. 6. Science & Technology Education Library. Springer Netherlands, 1999, pp. 95–132. DOI: 10.1007/0-306-47217-1_4.

[MS21] Craig S. Miller and Amber Settle. “Mixing and Matching Loop Strategies: By Value or By Index?” In: *Proc. of the 52nd SIGCSE.* SIGCSE ’21. Virtual Event, USA, 2021, pp. 1048–1054.

[NV+21] Jacqueline Nijenhuis-Voogt et al. “Teaching algorithms in upper secondary education: a study of teachers’ pedagogical content knowledge”. In: *Computer Science Education* 0.0 (2021), pp. 1–33. DOI: 10.1080/08993408.2021.1935554. eprint: https://doi.org/10.1080/08993408.2021.1935554.

[Pea86] Roy D. Pea. “Language-Independent Conceptual “Bugs” in Novice Programming”. In: *Journal of Educational Computing Research* 2 (1986), pp. 25–36.

[PF18] Amanda Peel and Patricia Friedrichsen. “Algorithms, Abstractions, and Iterations: Teaching Computational Thinking Using Protein Synthesis Translation”. In: *The American Biology Teacher* 80 (Jan. 2018), pp. 21–28. DOI: 10.1525/abt.2018.80.1.21.

[Pia+69] Jean Piaget et al. *Psychology Of The Child*. The Psychology of the Child. Basic Books, 1969.

[PSS88] D.N. Perkins, Steve Schwartz, and Rebecca Simmons. “Instructional Strategies for the Problems of Novice Programmers”. In: *Teaching and Learning Computer Programming*. Ed. by Richard E. Mayer. New York, USA: Routledge, 1988, pp. 153–178. DOI: 10.4324/9781315044347.

[QL17] Yizhou Qian and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review”. In: *ACM Trans. Comput. Educ.* 18.1 (Oct. 2017). DOI: 10.1145/3077618.

[RBH17] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. “Identifying Students’ Misconceptions on Basic Algorithmic Concepts Through Flowchart Analysis”. In: *Informatics in Schools: Focus on Learning Programming*. Ed. by Valentina Dagienė and Arto Hellas. Cham: Springer International Publishing, 2017, pp. 155–168.

[RDLR20] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. “A Miss is as Good as a Mile: Off-By-One Errors and Arrays in an Introductory Programming Course”. In: *Proc. of the 22nd Australasian Computing Education Conference*. 2020, pp. 31–38.

[RHG06] Anthony Robins, Patricia Haden, and Sandy Garner. “Problem Distributions in a CS1 Course”. In: *Proc. of the 8th Australasian Conference on Computing Education - Volume 52*. ACE ’06. Darlinghurst, Australia: Australian Computer Society, Inc., 2006, pp. 165–173.

[Rij+18] Wouter J. Rijke et al. “Computational thinking in primary school: An examination of abstraction and decomposition in different age groups”. In: *Informatics in education* 17.1 (2018), pp. 77–92.

[RM87] Lauren B. Resnick and S.T.E. Committee on Research in Mathematics. *Education and Learning to Think*. Online access: National Academy of Sciences National Academies Press. National Academies Press, 1987.

[RRR03] Anthony Robins, Janet Rountree, and Nathan Rountree. “Learning and Teaching Programming: A Review and Discussion”. In: *Computer Science Education* 13.2 (2003), pp. 137–172.

[SA16] David Statter and Michal Armoni. “Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders”. In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. WiPSCE ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 80–83. DOI: 10.1145/2978249.2978261.

[Sad10] D. Royce Sadler. "Beyond feedback: Developing student capability in complex appraisal". In: *Assessment & evaluation in higher education* 35.5 (2010), pp. 535–550.

[Sae+11] Mara Saeli et al. "Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective". In: *Informatics in Education* 10 (Apr. 2011), pp. 73–88.

[Sae12] Mara Saeli. "Teaching programming for secondary school : a pedagogical content knowledge based approach". English. Proefschrift. PhD thesis. Eindhoven School of Education, 2012. DOI: `10.6100/IR724491`.

[SB06] Carsten Schulte and Jens Bennedsen. "What do teachers teach in introductory programming?" eng. In: *Proceedings of the second international workshop on computing education research*. Vol. 2006. ICER '06. ACM, 2006, pp. 17–28.

[SBE83] Elliot M. Soloway, Jeffrey Bonar, and Kate Ehrlich. "Cognitive Strategies and Looping Constructs: An Empirical Study". In: *Commun. ACM* 26.11 (Nov. 1983), pp. 853–860. DOI: `10.1145/182.358436`.

[Sch12] Dane Schaffer. "An Analysis of Science Concept Inventories and Diagnostic Tests: Commonalities and Differences". In: *Annual International Conference of the National Association for Research in Science Teaching*. Apr. 2012.

[Sch+13] Stephan Schmelzing et al. "Development, evaluation, and validation of a paper-and-pencil test for measuring two components of biology teachers' pedagogical content knowledge concerning the "cardiovascular system"". In: *International Journal of Science and Mathematics Education* 11 (Dec. 2013). DOI: `10.1007/s10763-012-9384-6`.

[SG11] Andreas Stefik and Ed Gellenbeck. "Empirical studies on programming language stimuli". In: *Software Quality Journal* 19 (Mar. 2011), pp. 65–99. DOI: `10.1007/s11219-010-9106-7`.

[SH08] Victoria Sakhnini and Orit Hazzan. "Reducing Abstraction in High School Computer Science Education: The Case of Definition, Implementation, and Use of Abstract Data Types". In: *J. Educ. Resour. Comput.* 8.2 (May 2008). DOI: `10.1145/1362787.1362789`.

[Shu05a] Lee S. Shulman. *Signature pedagogies*. 2005.

[Shu05b] Lee S. Shulman. "Teacher education does not exist". In: *Stanford Educator* 7 (2005).

[Shu86] Lee S. Shulman. "Those Who Understand: Knowledge Growth in Teaching". eng. In: *Educational Researcher* 15.2 (Feb. 1986), pp. 4–14.

[Sim+06] Beth Simon et al. “Commonsense Computing: What Students Know before We Teach (Episode 1: Sorting)”. In: *Proceedings of the Second International Workshop on Computing Education Research.* ICER ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 29–40. DOI: 10.1145/1151588.1151594.

[Sim13] Simon. “Soloway’s Rainfall Problem Has Become Harder”. In: *2013 Learning and Teaching in Computing and Engineering.* 2013, pp. 130–135. DOI: 10.1109/LaTiCE.2013.44.

[Sle+86] D. Sleeman et al. “Pascal and High School Students: A Study of Errors”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 5–23. DOI: 10.2190/2XPP-LTYH-98NQ-BU77. eprint: https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77.

[Sor13] Juha Sorva. “Notional Machines and Introductory Programming Education”. In: *Trans. Comput. Educ.* 13.2 (2013), 8:1–8:31.

[SS13] Andreas Stefik and Susanna Siebert. “An empirical investigation into programming language syntax”. In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), pp. 1–40.

[Ste18] Friedrich Steimann. “Fatal Abstraction”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Onward! 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 125–130. DOI: 10.1145/3276954.3276966.

[SW80] Elliot M. Soloway and Beverly Woolf. “Problems, Plans, and Programs”. In: *Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education.* SIGCSE ’80. New York, NY, USA: ACM, 1980, pp. 16–24. DOI: 10.1145/800140.804605.

[TG10] Allison Tew and Mark Guzdial. “Developing a validated assessment of fundamental CS1 concepts”. In: Jan. 2010, pp. 97–101. DOI: 10.1145/1734263.1734297.

[TL14a] Donna Teague and Raymond Lister. “Blinded by their Plight: Tracing and the Preoperational Programmer”. In: *PPIG.* June 2014.

[TL14b] Donna Teague and Raymond Lister. “Programming: Reading, Writing and Reversing”. In: *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’14. New York, USA: ACM, 2014, pp. 285–290. DOI: 10.1145/2591708.2591712.

[Van+10] Tammy Vandegrift et al. “Commonsense computing (episode 6): Logic is harder than pie”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling’10* (Jan. 2010). DOI: 10.1145/1930464.1930479.

[Ver11] Tom Verhoeff. “On Abstraction and Informatics”. In: *Informatics in Schools. Contributing to 21st Century Education: 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives, ISSEP 2011, Bratislava, Slovakia, October 26-29, 2011. Proceedings.* Ed. by Ivan Kalaš and Roland T. Mittermeir. 2011, pp. 1–12. DOI: `www.issep2011.org`.

[Wan08] Yingxu Wang. “A Hierarchical Abstraction Model for Software Engineering”. In: *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering.* ROA ’08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 43–48. DOI: `10.1145/1370164.1370174`.

[Win11] Jeannette M. Wing. “Computational Thinking: What and Why?” In: *The Link Magazine* (2011).

[Win96] Leon E. Winslow. “Programming pedagogy—a psychological overview”. In: *ACM Sigcse Bulletin* 28.3 (1996), pp. 17–22.

[WM99] Paul White and Michael Mitchelmore. “Learning mathematics: A New Look at Generalisation and Abstraction”. In: *AARE Annual Conference.* AARE ’99. deakin, ACT, Australia: Australian Association for Research in Education, 1999, pp. 1–12.

[WSR87] Suzanne Wilson, Lee S. Shulman, and AE Richert. “" 150 different ways" of knowing: Representations of knowledge in teaching”. In: *Exploring Teachers’ Thinking* (Jan. 1987), pp. 104–124.

[WW15] David Weintrop and Uri Wilensky. “Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research.* ICER ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 101–110. DOI: `10.1145/2787622.2787721`.

Chapter 2

Pilot studies

The literature review summarized in the previous chapter confirms that iteration is a central topic in introductory courses. In particular, it pinpoints students' difficulties with loop conditions and nested flow-control constructs. This chapter presents the two preparatory "pilot" studies aimed at eliciting some aspects of teachers' PCK, as well as the teachers' and students' subjective perceptions of difficulties with programming tasks, and specifically with iteration, in the high school context.

Based on the framework introduced in Section 1.6, we outline in Section 2.1 an interview protocol drawing inspiration from the Content Representation (CoRe) format [LMB08]. Some of the insights emerging from the teachers' interviews have been used to develop the pilot survey addressed to students, which includes subjective perception questions and three small programming tasks on iteration constructs. The structure of the survey and its main outcomes are presented in Section 2.2. Finally, in Section 2.3 the subjective perception of students is compared with that of teachers.

2.1 Teacher interviews

In this chapter the goal is to catalog a series of questions to be asked to high school Computer Science teachers, to both technical and lyceum schools (for more about computing in the Italian high schools see Appendix C). The various questions formulated are not inspired by a single starting point, but we tried to identify questions from different points of view: Concept Inventory, Taxonomy, and students point of view. The choice of asking questions according to the students' point of view arises from our personal experience in the field, where students often complain about teaching methods and the time allocated to them for learning, as well as the objections that are increasingly present in the evaluation phase.

2.1.1 Aims and scope of the interviews

Interviewing teachers has become a popular way of data-collection in STEM fields, in particular in Mathematics and Physics education.

In our study, the interviews are designed for high school teachers, teachers of technical institutes, lyceums, and possibly of professional institutes. We have requested some additional information on the training of each single teacher in order to make a more accurate analysis. The additional information could be the type of school and the address at which the teacher works, whether they have a fixed term contract, age or age group, the curriculum of studies, or in what discipline they graduated (Computer Science, Mathematics, Statistics, Electronic engineering, Computer engineering, Management engineering, etc.). The curriculum of studies could highlight different ideas and methodologies, based on the degree obtained; and even age could be related to different approaches amongst teacher.

In this work, as an initial part of a larger research project, we tried to understand the issues related to iteration through direct interviews with teachers. The results of the interviews allowed us to identify strategies and new teaching methods useful to improve existing teaching approaches. Moreover, the results obtained from the interviews allowed us to plan and design the subsequent student survey with greater care, identifying more effectively the questions and tasks proposed.

In summary, besides getting some insights on the teachers' general pedagogical approach to programming instruction, the interviews aim at answering the following research questions:

- How relevant do teachers deem iteration as compared to other introductory programming topics?
- What are students' major difficulties with iteration in their view?
- What is their approach to teach and assess the learning of iteration?

2.1.2 Methodology

As far as teachers' instructional experience and practice are concerned, the usual reference framework is that of Pedagogical Content Knowledge (PCK), originally proposed by Shulman [Shu86], to characterize the "blending of content and pedagogy into an understanding of how particular aspects of subject matter are organized, adapted, and represented for instruction". In this respect, the Content Representation (CoRe) format is an instrument to investigate teachers' PCK of a specific topic [MKB99; LMB08] through 8 standard questions, which are meant to capture teachers' knowledge about key ideas in connection with the topic. Although this approach has been mainly applied in science education research, there have also been a few attempts to exploit it to investigate the teaching of programming [BSS13; Sae+11; Bar+15].

Several authors have used interviews as a survey tool. Generic knowledge about pedagogy, how students learn, teaching approaches, methods of assessment and knowledge of different theories about learning.

The interview model that we propose here differs from those presented in other works, in fact we incorporate concepts not only related to Computer Science (Zazkis and Hazzan [ZH98], Hazzan et al. [HLR11]), but also to Mathematics and other subjects.

Having set the general objectives and background of this main section, the rest of the section is organized as follows. Section 2.1.3 presents the survey organization. Section 2.1.4 describes general data collection, while Section 2.1.5 summarizes the results of the analysis. In Section 2.1.6 we discuss the findings and outline some future perspectives.

2.1.3 Characterization of the instrument

To begin with, the structure of the interview protocol, partly inspired by the approaches to elicit the PCK presented in the literature, is outlined in Figure 2.1, where the most significant questions are reported verbatim. A set of questions (point 1) is aimed at framing iteration within the general context of introductory programming and the related prerequisites. Other questions (2) attempt to ascertain how central the learning of iteration is in the teacher's perspective, but without mentioning this topic explicitly (the wording of question 2.2 being meant to prevent discussion of issues arising with object-orientation, when covered in an introductory course). Then, the focus moves specifically to the teaching, learning and assessment of iteration (3). Some questions about more general educational issues (4) and a final open question (5) to collect further ideas not covered in the previous points conclude the interview session.

1. Course organization (5 questions)	
programming languages, key programming concepts, related lesson plan, how much time for each concept, extra-computing prerequisites	
2. Introductory programming in general (6 questions)	
2.1. teaching	Are the tasks assigned to students simple variations of those dealt with in class? Or do they cover unfamiliar situations as well?
	What are your more frequent suggestions to students for improving their programming performance?
2.2. learning	What is the major learning obstacle that students face before being introduced to object-oriented programming?
2.3. assessment	How do you assess a working solution if it is inefficient, or convoluted, or somehow at odds with what you expected?
	While trying to achieve the assigned tasks, do you expect your students to apply the models introduced in class? Or do you also appreciate "creative" solutions?
	Are the different solutions by students compared in class? How?
3. Focus on iteration (5 questions)	
3.1. teaching	Can you show some of your favorite examples to make students learn how to apply the iteration constructs?
	In your teaching, do you cover the mappings between different iteration constructs (for, while, do-while/repeat-until)?
3.2. learning	In your experience, to what extent can students master the termination condition of a loop?
	Which features of the iteration constructs are usually understood by (most) students, and which are more difficult for them?
3.3. assessment	How do you usually assess an incorrect termination condition? And oversights about the first or last iteration?
4. General educational issues (3 questions)	
strategies to motivate students, manage different learning styles, deal with students' criticisms	
5. Other thoughts (1 question)	
Any other issues you deem important to consider about the teaching/learning of programming?	

Figure 2.1: Structure of the teacher interview protocol.

As noted in Section 1.3 several types of questions about PCK, CoRe have been proposed to Computer Science teachers, focusing on fundamental concepts rather than a particular concept or Big Idea. The questions elaborated, and seen previously, have undergone a re-elaboration and selection phase which eventually led to the elaboration of 19 questions, listed in Table 2.1 below [1].

[1] In the list (Table 2.1), the questions, already enumerated previously, have been grouped by topic, thus altering the numerical order.

Table 2.1: Selected questions.

1. What programming language do you use to introduce the basic aspects of programming?
2. What are the conceptual issues that you consider important as a teacher?
4. In what order do you introduce the concepts related to the basic aspects of programming?
5. How do you organize the time to introduce individual concepts?
8. What are the prerequisites, not strictly related to Computer Science, that you consider necessary or useful for understanding the basic aspects of programming and the examples you propose?
3. Before introducing object-oriented programming, what is the obstacle you find most difficult to overcome for the students?
6. Can you describe any of the examples that you propose to the students to have them apply the iteration?
7. Do the proposed problems deal with situations similar to a series already dealt with or do they also propose unexpected situations?
9. What do you think students do about finding the termination condition for the loop?
10. Are transformation schemes being addressed to move from one iterative instruction to another?
11. What weight do you give in the evaluation to an incorrect termination condition, or to errors related to the first or last iteration?
12. How do you evaluate solutions that are functional but inefficient, or involved, or different from those you would have expected?
13. Do you expect your students to apply models of examples already seen, or do you also appreciate creative solutions that move away from the examples presented?
14. What are the aspects of the iteration that the majority of students learn, and which are more difficult?
15. What kind of suggestions do you often give to a student so that they can improve their performance?
16. Are the different solutions proposed by the students compared and discussed. How?
17. What are your strategies to motivate students?
18. What strategies can you implement to manage the various learning styles of students?
19. How do you take into account the criticisms that students can make to address their needs?

Finally, at the end of the interview, with the question *Are there any aspects, not touched by the questions, that you would like to point out?* The interviewee could identify some issues not previously identified by the questions.

The complete set of questions submitted to the teachers is presented in Appendix D.

2.1.4 Data collection

We conducted accurate face-to-face interviews with 20 experienced high school teachers of Informatics, working in 10 technical institutes and lyceums from a large area in the North-East of Italy. Each such session lasted one to two hours and was audio-recorded and (partly) transcribed with the interviewee's agreement.

The works of Wiedenbeek [Wie89] and Luxton-Reilly [LR+17b] have been used as a reference to identify specific questions (see Table 2.1) about the interactions.

Most of the teachers interviewed are Computer Science graduates, see Figure 2.2. Furthermore, their teaching experience, for years of service, is adequately distributed (see Figure 2.3), so the views and practices of both experienced and early career teachers are then collected. However, no teacher interviewed was a novice, with less than four years of teaching experience.

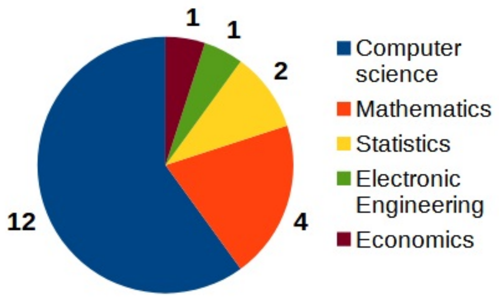


Figure 2.2: Educational qualifications of the teachers interviewed.

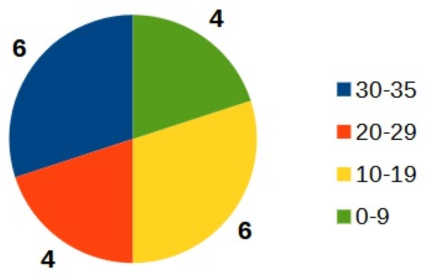


Figure 2.3: Teaching experience, years of service.

2.1.5 Results

To present the main results of our investigation we follow the general structure outlined in Figure 2.1 and, about specific questions, Table 2.1.

Course organization

The most significant insight from this general section of the interviews is the (weighted) list of key concepts identified by the teachers.

Figure 2.4 lists the key programming concepts indicated by the teachers via a selection amongst multiple options items, later aggregated into tightly related concepts.

What emerges clearly from the data in Figure 2.4 is that almost all the teachers mention precisely flow control constructs and *iteration* among the most important concepts in introductory programming — the second most popular choice being variables and assignment.

Concerning teachers' adopted programming languages there is a fairly high consensus, with C, C++ and C# at the top of the ranking , see Figure 2.5.

It may also be worth remarking that several teachers introduce different programming languages and other design languages, such as *flow-charts*, to analyze the control constructs.

Another general issue of relevance here concerns the extra-computing prerequisites, see Figure 2.6. Large percentages of teachers consider the mathematical/logic background, as well as to *text comprehension*[2], as critical — although often insufficiently developed — to the practice of programming. Moreover, and quite surprisingly, about one quarter of them revealed to have faced problems with geometry, probably because of its connections with particular problem domains [3].

[2]Text comprehension is an issue for 8 teachers.

[3]Concerning *problem domain* we mean the area of expertise or application that needs to be investigated to resolve a problem.

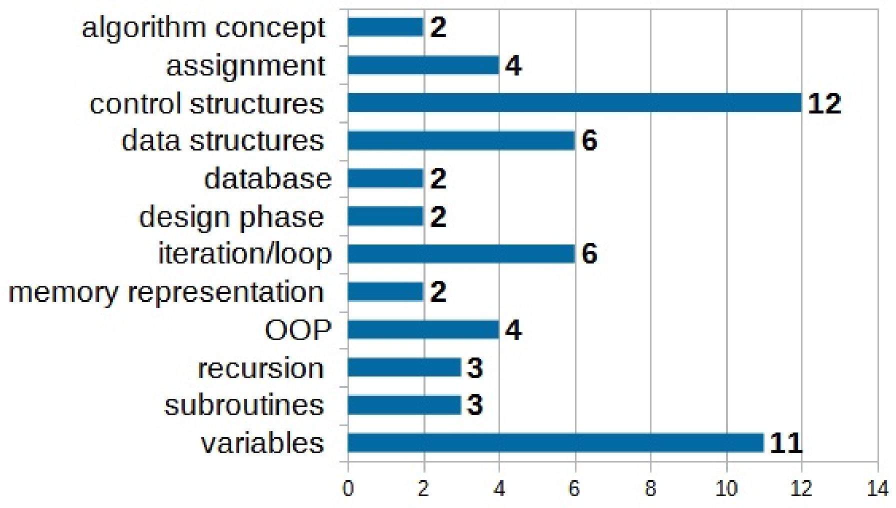


Figure 2.4: Key programming concepts for teachers (more options were possible).

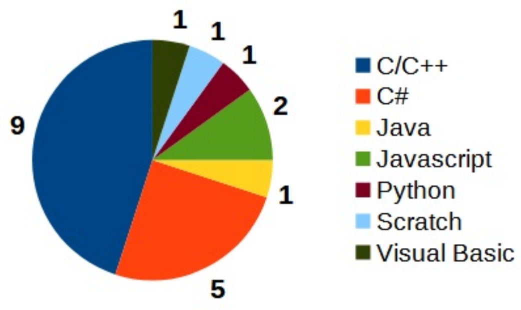


Figure 2.5: Languages used in introductory courses.

Other prerequisites that have been defined as important are: understanding formalism, knowledge of the English language, Physics, precision, creativity, knowledge of data types, concept of function, knowledge of fundamental operations, even and odd numbers, passion in solving problems.

Concerning the mathematical prerequisites a teacher argued that "*students think in watertight compartments, those who learn in one subject fail to use it and put it into practice in another one, do not have an interdisciplinary vision*".

Introductory programming in general

Teachers think serious learning obstacles are present in students' learning before introducing OOP paradigm [4], as shown in Figure 2.7

On the other hand, beside indicating a few of the key concepts taught, several

[4]We single out the OOP paradigm as the topic that delimits the basic contents, present in the introductory courses, from the more advanced ones.

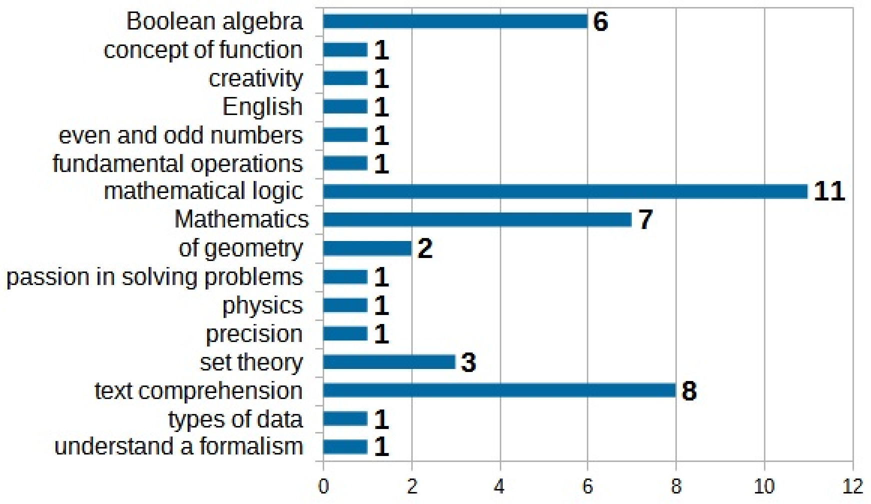


Figure 2.6: Most relevant extra-computing prerequisites (more options were possible).

interviewees emphasize the high-level thinking skills of abstraction and generalization.

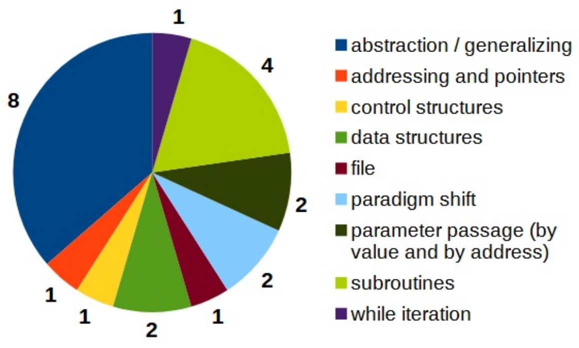


Figure 2.7: Major obstacles in student learning, before introducing OOP (more options were possible).

Hence, serious learning hurdles are recursion, arrays and data structures, subroutines, i.e. precisely the last topics introduced by teachers shortly before the end of a introductory course. Among the suggestions to the students, about half of the teachers give prominence to the use of "paper and pencil", to clarify ideas before starting to work with a computer. In the words of a teacher: *"read the text carefully, then analyze the problem and check a preliminary solution with paper and pencil"*. In addition, students are often encouraged to compare their programs with those of their peers.

Most of the teachers also assign unfamiliar tasks, and this occurs when their students have reached a sufficient degree of mastery of programming basics. The fact that teachers insist on the use of “paper and pencil” because they believe that the students neglect this phase by privileging directly the implementation one, which however can lead them to run into errors.

The emerged data show how teachers tend to favor the request for constancy in the study and the desire that students would do more exercises at home.

Regarding assessment, it emerged that the penalty for inefficient solutions can amount up to 20–25% of the marks, whereas the instructors tend to be less strict about programming style, so giving prominence to the fact that a program can work properly. In many cases the teachers say they do not penalize excessively in the initial moments of the course, but the more time passes, the more they also consider inefficiency errors. Many teachers tend to explain their preferred solutions so that students realize the difference with their own proposals.

All the interviewees, in different ways, stated that they stimulate discussion, often have their proposals analyzed by the students, trying to highlight the advantages and disadvantages, and trying to get them to improve the solution. Often the discussion is motivated by the correction of previously assigned exercises, perhaps highlighting the most original solutions and motivating the student to convince teachers and classmates of the benefits of the proposed solution. Furthermore, several teachers are convinced of the importance of getting students to work in pairs, then pushing them to present their proposals to the class. Sharing and peer discussion of solutions is therefore positively evaluated.

Focus on iteration

Figure 2.8 reports the collected data on the major sources of difficulties with iteration. Teachers indicate the complexity of loop conditions (in terms of use of logical (Boolean) operators) and the treatment of the exit condition as problematic. However, they seem to have contrasting views as to the other aspects addressed. On the one hand, a number of teachers point out students’ misuse of iteration constructs, in particular *while* vs. *do-while* and the overuse of *for* loops in situations where it is not an appropriate choice.

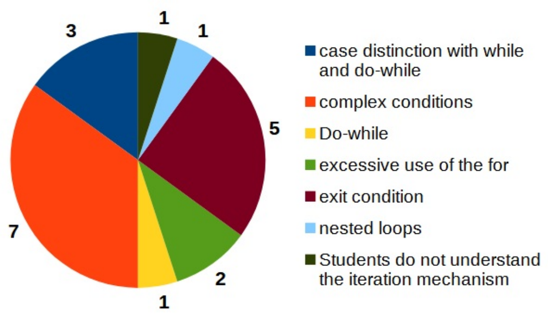


Figure 2.8: Major difficulty with iteration in teachers’ opinion.

A teacher pointed out *"students are not always doing well, they have difficulty with the exit condition of the loop, but above all with the complex conditions"*. Moreover, the nested iterations management, or iterations with nested flow control constructs, result in a certain difficulty, and this case is the most difficult to handle for students, perhaps because they cannot understand the functioning mechanism of nested loops one inside the other. One teacher said that *"some students were able to understand the mechanism of operation of the iteration once they learned the concept of an assembly jump"*.

Moreover, teachers think that students tend to avoid using Boolean operators to build efficient conditions. An interviewee argued that *"for loops are easier to students than while loops, since it is not necessary to figure out a suitable condition"*. To overcome these difficulties, several teachers insist on carrying out preliminary analysis steps based on flow-chart representations, so that students can clarify their ideas and understand the implied concepts in more depth. Also *"tracing the program execution with paper and pencil may be helpful to student, but similar tasks are only rarely done"*; indeed, the program code is *"less effective than a flow-chart"* to visualize what is going on when a program is run. Incidentally, although loop conditions and the related border computations (first and last repetition) play a crucial role in the understanding of iteration, in general the teachers take into consideration a varied range of factors to assess students' programs (unless the focus of the assignment is precisely *on* the loop condition), depending also on the connections with the examples worked out in class.

When asked about the examples they commonly presented in class to explain iteration, the teachers mentioned the tasks listed below, the most popular ones being those related to elementary Mathematics:

sum/average of a number sequence	power function
counting odd/even num. in a sequence	factorial function
min/max values of a sequence	Euclid's GCD algorithm
input data control (do-while)	math number sequence
first n multiples of a number	number base conversion
iteration over an array	pictures drawing with chars
nth element of a sequence	drawing polygons

Unexpectedly, although all the interviewees said that they present the main forms of loop constructs — *for*, *while*, *do-while* — and treat their similarities and differences by showing appropriate examples, few teachers are also explicit about the mappings between such control structures, e.g. how to transform a *for* or a *do-while* loop into a *while* and conversely. However teachers often propose as exercises the review of the proposed problem using a different iteration instruction. One teacher, however, attempts to emphasize the role and power of iteration by discussing the universality of three basic control structures, as captured by Böhm-Jacopini's theorem.

To facilitate understanding for the students, teachers present various cases, with different iterative instructions, by flow-chart to make easy students' comprehension. However, some teachers (explicitly two) argued that in this case students tend to learn by heart.

General educational issues

From the interviews, it emerges that the teachers try to motivate their students by proposing interesting real world problems or the implementation of computer games and other graphics applications that are meaningful to them. Often, a driving factor to increase students' engagement with learning is their teachers' enthusiasm, e.g. bringing to school the challenges faced in her/his professional experience. The educational objective is that students can make sense of the importance of mastering particular concepts and acquire particular skills.

It has emerged from the experience of some teachers that recovery activities can be useful to follow the student in difficulty step by step, see Figure 2.9, it can be effective to have the topics explained by others, for example the laboratory teacher or another student. For some interviewees it is important to make them understand that *"Informatics/CS is not just the use of the tool, the computer"*.

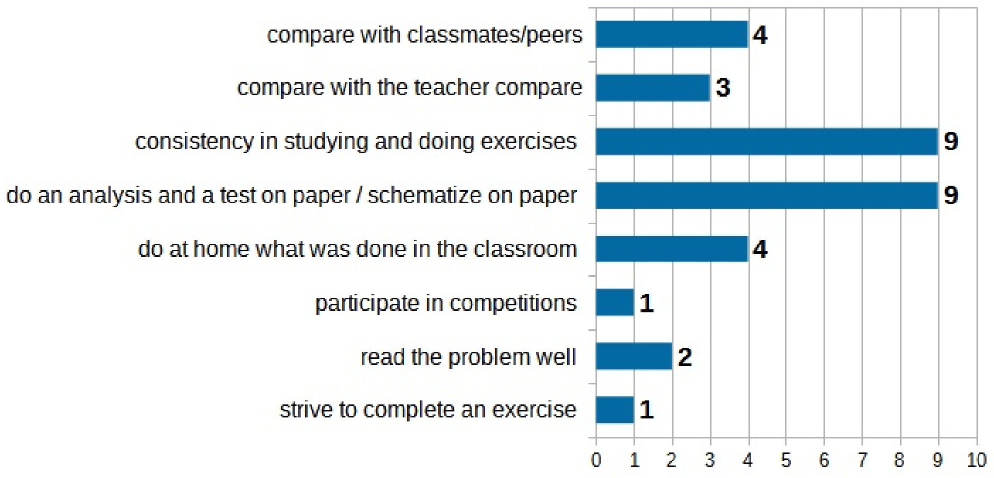


Figure 2.9: Teachers' suggestions to improve students' performance (more options were possible).

Other thoughts

As seen from the outline in Figure 2.1, every interviewee was asked for further possible ideas or suggestions they deemed important with regard to the teaching and learning of programming. Here are the main points raised by the teachers, that go far beyond the scope of our present work:

- *Lack of alignment between topics in Informatics and Mathematics.* A possible explanation of students' difficulties with the application of mathematical and logical concepts is that Informatics and Mathematics are not well integrated in the standard high-school curricula, recently subjected to reform. In other words, some of the mathematical topics may be covered either too early or too late to be effective when they are required to learn programming, in particular with regard to the logical aspects of Mathematics.

- *Robotics environments.* In order to enhance students' engagement, some schools have developed robotics laboratories. These usually successful experiences may also be proposed at earlier education levels, so that high-school students will be more familiar with Informatics and programming concepts.

- *Object-first approach.* Some teachers are considering whether the learning of programming could be improved by starting from the beginning with the object-oriented paradigm.

- *Late teaching.* Since this is a subject that requires reasoning, one should try to teach it earlier, more and better in the biennium of upper secondary schools, but also in the previous levels of teaching.

2.1.6 Discussion

In general, iteration is among the few most central concepts for Computer Science teachers, a claim on which all the interviewed teachers appear to agree, even though there are concepts that are considered more challenging, see Figure 2.4.

In the teachers' opinion students' major obstacles relate to abstraction and generalization, as shown in Figure 2.7. Furthermore, students' difficulties with iteration are identified in relation to complex condition and exit condition, see Figure 2.8. Difficulties about complex condition are evidently related to mathematical prerequisites and knowledge of Boolean algebra, as well as logical skills, as shown in Figure 2.6, which possibly were not well acquired. However, Logic has a particular importance in learning the fundamental concepts of Computer Science and coding, as reiterated by Wagner-Doebler [WD97], De Mol et al. [DMP15]. Blass [Bla16] also argued that "*mathematical logic provides tools for understanding and unifying topics in computer science*".

The lack of alignment between the Mathematics and Computer Science curricula is therefore a sore point, highlighted also previously when it came to the prerequisites, already Rich & Waters [RW88] and Quindeless [Qui14] noted the importance of the aspects of mathematical logic. The link between computer technology and high school Mathematics was investigated by Turskienė [Tur02], who argued that the integration of various computer technologies in teaching Mathematics would be appropriate.

The teachers have given greater emphasis to the problem of defining the condition of a loop, indicating that the condition itself is sometimes difficult to identify by a student, even more so when it is complex and must be used by operators logic.

However already both Knuth [Knu76] and Myers [Mye90] identified the central role of mathematical logic in Computer Science education. This critical issue is probably due, as some teachers say, to a misalignment of Computer Science and Mathematics courses. In Mathematics courses in fact the aspects of Logic, like those of set theory, are tackled in previous years, perhaps with inadequate in-depth analysis, which leads students to acquire those competencies that are then required for subsequent computer science studies. The aspects of Logic could also be scarcely assimilated due to teachers spending a short time over the topic, or due to the inadequate examples employed. It would be interesting to verify if in other countries these critical issues about misalignment of the programs between Mathematics and Computer Science is also present.

An aspect not to be underestimated, however, is that students, as stated by some teachers, tend to forget some previously seen concepts, often from other subjects, tending to work with airtight compartments and not grasping the interdisciplinary aspects of some topics. A hypothesis for the logical difficulties and also for the abstraction of students could be identified in the examples that are proposed to them, "trivial" from the logical point of view and too oriented to the use of the For-loop.

Moreover, most of the program examples teachers usually see in connection with iteration are quite straightforward (see the list in Section 2.1.5) and tend to induce the use of stereotypical patterns. Thus, to help students work with non-trivial loop conditions and neat combinations of flow-control constructs, it can be desirable to develop a catalog of significant examples presenting more varied and interesting structures. In a similar spirit, it may be helpful to investigate the role of iteration in the larger programming tasks in which the students engage (e.g. to solve "real world" problems, to implement computer games, etc.).

As recognized by several interviewees (see again Figure 2.4), it is likely that the major issues depend on students' difficulties to take a more abstract, comprehensive perspective when dealing with programs. A possible way, identified by the teachers, to induce students to develop their abstraction skills is to contrast their tendency to approach a task by trial-and-error and require them to analyze the problem with paper and pencil. Another possibility is to demand that students organize their programs into several functions and procedures to introduce meaningful levels of abstraction. In addition, it could be interesting to envisage and to explore the effectiveness of methodological tools inspired by the notion of loop invariant, see e.g. the pedagogical work in [Tam92; Gin03], suitably adapted to fit less formal learning styles [Ast91].

2.2 Student pilot survey

While information from teachers can be collected via carefully conducted face-to-face interviews, interviewing a large number of students can be onerous. Thus, we have designed an online survey, organized into multiple choice questions, open-ended questions and three small programming tasks. In particular, the three tasks aimed to gain some preliminary insights into a few aspects of the understanding of basic iteration constructs. The collected results indicate that most students seem to have developed a viable mental model of the basic workings of the underlying machine, but, on the other hand, dealing at a more abstract level with loop conditions and nested flow-control structures appears to be challenging.

2.2.1 Aims and scope of the pilot survey

Collecting students' opinion and verifying their performance via survey has become a popular way of data-collection, in particular in this pandemic period.

In our study, the survey is designed for high school students. Both in technical institutes and lyceums, we could collect their perceptions regarding programming in general and specifically concerning iteration. Furthermore, students' performance in three tasks aimed to gain some preliminary insights into a few aspects of the understanding of basic iteration constructs identified by teachers during interviews, as well as allowing to analyze how students' subjective perception of difficulty correlate with their effective performance.

The results of the survey allow us to identify students' effective difficulties, correlating perceptions with performance, allowing insight into different approaches to teach and learn iteration in high school.

Research questions underlying the design of the pilot survey were:

- What are students' major difficulties with iteration in their subjective perception?
- Are there differences between students' vs. teachers' perceptions of difficulties?
- To what extent do students' subjective perceptions of difficulty correlate with their actual achievements in the small proposed tasks?

2.2.2 Methodology

To obtain meaningful insights from students, we devised a short survey with sharp closed-ended questions, so that they did not get bored while answering them. Thus, to get feedback on their subjective perception of "learning difficulties", we decided to provide a few lists of concepts among which to choose (plus an open "other" field).

To identify small sets of basic programming-related concepts we have drawn from some "validating" work on Concept Inventories for Computer Science [Gol+10; Sch12] and introductory programming [Cac+16], as well as from the overview analysis [LR+17b].

The three tiny programming tasks address each of the learning dimensions introduced in [Mir12], namely the understanding of the computation model underlying iteration, the ability to establish relationships between the components of a loop and the statement of a problem, and the ability to see the program structures based on iteration at a more abstract level.

In addition, in light of the presumed role of meta-cognitive skills in effective learning [BRT05], we asked students two questions about their subjective perceptions of difficulty.

Even though from an educational researcher's viewpoint it would have been more insightful to cover a larger and more varied set of programming tasks, we decided to assign only three small tasklets in order to limit the risk that teenage students lose their concentration and provide scarcely meaningful answers.

Having set the general objectives and background of this main section, the rest of the chapter is organized as follows. Section 2.2.3 presents the survey organization, whilst Section 2.2.4 introduces the tasklets and the questions asked to students. Section 2.2.5 describes general data collection while Section 2.2.6 summarizes the results of the analysis. Finally, in Section 2.2.7 we discuss the findings and outline some future perspectives.

2.2.3 Instrument

The questions cover multiples points of view rather than having a single starting perspective, and include Concept Inventory, Taxonomy (Fuller et al. [Ful+07]), and students point of view. The choice of asking questions according to the students' point of view arises from our personal experience in the field, where students often complain about teaching methods and the time allocated to them for learning, as well as the objections that are increasingly present in the evaluation phase.

As a first step, we analyzed the style of interviews already conducted with teachers and Computer Science students, to understand the type of questions to ask and how to formulate them.

The outcomes obtained from the teachers' interviews allowed us to better define the objectives and the questions to ask the students.

The survey addressed to students includes 11 questions, three of which require to solve tiny problems by analyzing either small flow-charts or short code fragments based on iteration. The main features of the questionnaire are shown in Figure 2.10.

1. Course organization (3 questions) favorite programming languages, poor understanding of mathematical/logical prerequisites, accordance of the subject with personal expectations
2. Introductory programming in general (3 questions) Do you think more time would be needed on some programming concepts? Which ones? (range of options or open "other" field) Which kind of errors has been most penalizing for your grading? (open question) Are you usually successful in solving unfamiliar programming problems? (Likert scale of 4 levels)
3. Focus on iteration (1 question and 3 tiny problems) What do you find most difficult when trying to use a loop? (range of options) **Problem 3.1:** Given the statement of a simple problem being solved, choose the correct *loop condition* in a flow-chart. (4 options available) **Problem 3.2:** Given a while loop with a composed condition and a nested conditional, determine the number of iterations for a given input. (6 options) **Problem 3.3:** Given 5 code fragments involving nested construct with simple conditions, identify the functionally equivalent ones.
4. Other thoughts (1 question) Do you have any suggestions to make learning Computer Science more interesting?

Figure 2.10: Structure of the student survey.

The complete set of questions and tasks submitted to the students can be found in Appendix E.

2.2.4 Tasklets and questions

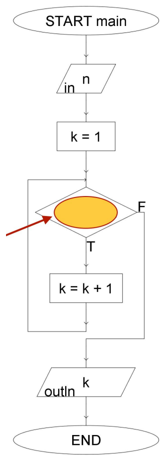
Of course, the three small tasks were not meant to assess students' mastery of iteration, but just to get some insight about the alignment between perceived and actual difficulties.

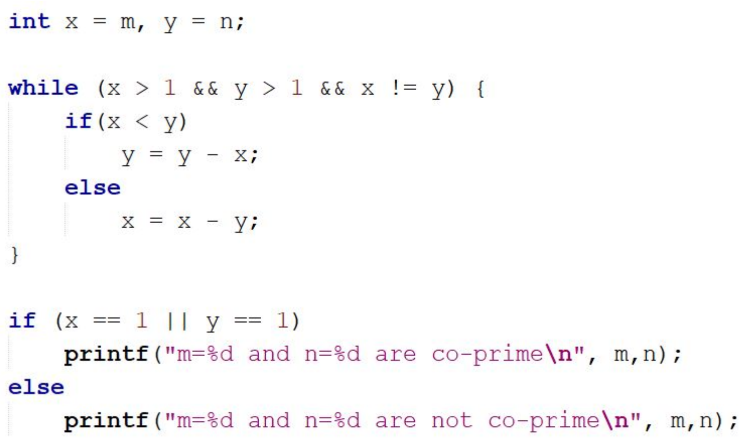
Tasklet 1: identifying the correct loop condition

Tasklet 1 was aimed to explore the ability to draw connections between a simple loop condition and the statement of a problem by reasoning on a flow-chart.

Problem statement: *The algorithm represented by the flow-chart in Figure 2.11 (left) computes the number of bits of the binary representation of a positive integer n, i.e. the smallest exponent k such that 2^k is greater than n. Choose the appropriate condition among the four listed below.*

The four available options were: $2^k = n$, $2^k \leq n$, $2^k < n$ and $2^k > n$. To achieve this — supposedly easy — task, students were expected to read carefully the statement above and, for each of the listed conditions, figure out the relationship between k and n after exiting the loop. The specific focus of this tasklet is suggested by the frequency of condition-related issues, see e.g. [CZP14].





```
int x = m, y = n;

while (x > 1 && y > 1 && x != y) {
    if(x < y)
        y = y - x;
    else
        x = x - y;
}

if (x == 1 || y == 1)
    printf("m=%d and n=%d are co-prime\n", m,n);
else
    printf("m=%d and n=%d are not co-prime\n", m,n);
```

Figure 2.11: Flow-chart of *tasklet 1* and code of *tasklet 2*.

Tasklet 2: ascertaining the number of iterations

Tasklet 2 addressed students' mastery of the "mechanics" of the execution of a loop controlled by a non-trivial condition and including a nested `if`.

Problem statement: *The program shown in Figure 2.11 (right) checks if two positive integers m and n are co-prime. If the input values are m=15 and n=44, how many times the* `while` *loop will repeat?*

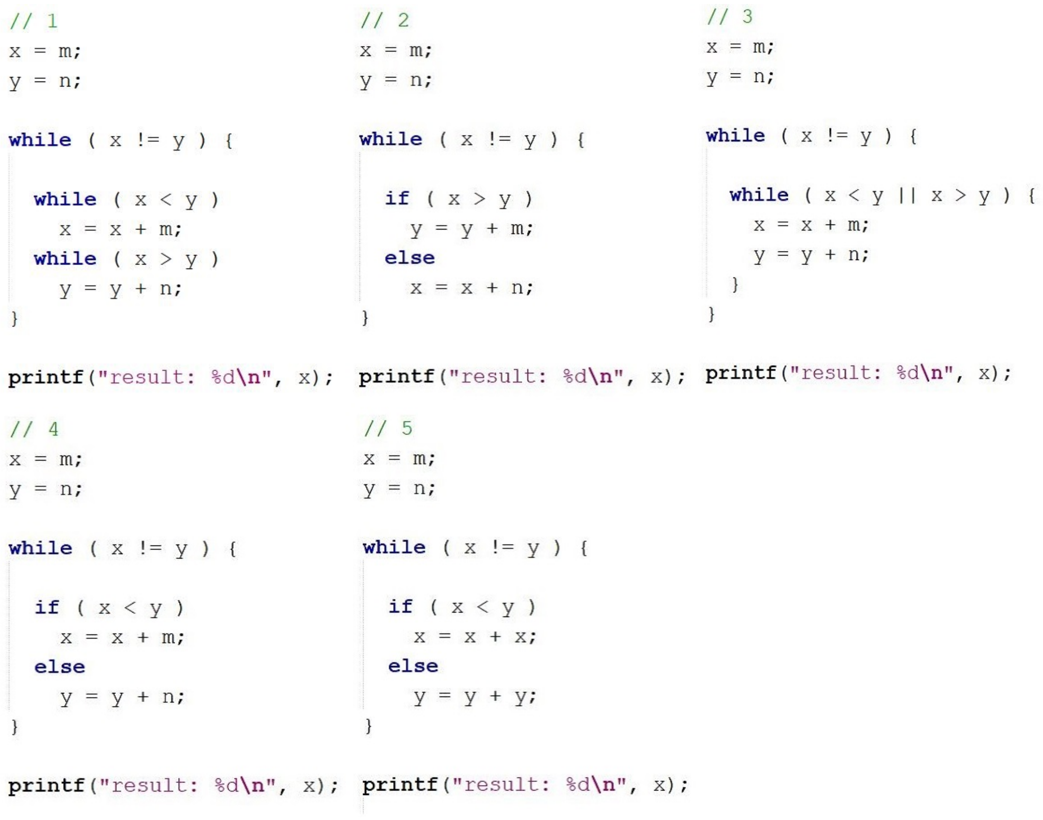
The above question could be answered by choosing among six options, namely: 0, 1, 2, 3, 4 *or more*, and *the loop never ends*. As we can see in Figure 2.11, the loop is characterized by a composite condition (using two `and`s) and a nested `if-else`. In order to identify the right option students were essentially required to trace the code execution carefully. Thus, this tasklet addresses tracing skills, which have been in the scope of several investigations, e.g. [Lop+08].

Tasklet 3: recognizing functionally equivalent programs

Tasklet 3, the most challenging one, asked to recognize equivalence between different programs in order to investigate the ability to grasp comprehensively nested combinations of conditionals and iteration constructs.

Problem statement: *Consider the five programs in Figure 2.12 and assume that the input values of m and n are always positive integers. Two such programs are equivalent if they compute and print the same output whenever they are run for the same input data. Identify the equivalent programs in Figure 2.12.*

To approach this problem on functional equivalence, students had to reason at a more abstract level. Each program involves nested constructs whose behavior must be grasped and dealt with comprehensively. The last tasklet is similar in structure as well as in spirit to that discussed in [IM20].



```
// 1
x = m;
y = n;

while ( x != y ) {

  while ( x < y )
    x = x + m;
  while ( x > y )
    y = y + n;
}

printf("result: %d\n", x);
```

```
// 2
x = m;
y = n;

while ( x != y ) {

  if ( x > y )
    y = y + m;
  else
    x = x + n;
}

printf("result: %d\n", x);
```

```
// 3
x = m;
y = n;

while ( x != y ) {

  while ( x < y || x > y ) {
    x = x + m;
    y = y + n;
  }
}

printf("result: %d\n", x);
```

```
// 4
x = m;
y = n;

while ( x != y ) {

  if ( x < y )
    x = x + m;
  else
    y = y + n;
}

printf("result: %d\n", x);
```

```
// 5
x = m;
y = n;

while ( x != y ) {

  if ( x < y )
    x = x + x;
  else
    y = y + y;
}

printf("result: %d\n", x);
```

Figure 2.12: The five programs to be compared in tasklet 3.

Subjective perception questions

Besides engaging in the three tasklets above, the students were asked two short questions about their subjective perception of difficulties. The first one was a multiple choice question: “*What do you find most difficult when you use loops*?" The five available options reported the difficulties that emerged as most significant from the teachers' interviews [SM19]:

i. To find the condition of a `while` or `do-while` loop;

ii. To define a complex condition including logical operators (AND, OR, NOT);

iii. To deal with nested loops;

iv. To understand, in general, when the loop should end;

v. To deal with the loop control variable.

It was important for us to verify whether or not the students' opinions matched those of the teachers.

The second question: "*What kind of mistakes affected your performance most significantly*?" was instead open, so the students could choose to indicate any source of error, either conceptual or of a different nature.

2.2.5 Data collection

The (anonymous) survey was administered to 164 students, most of whom were attending the second or third year (age 15–17) of scientific and technical high schools, i.e. when the basic flow-control constructs are introduced. In the lyceum of applied sciences and in technical high schools with specialization in CS, courses in Informatics are not electives. The survey was administered at school, under the supervision of a teacher, the students had about an hour to respond. As we have seen in Section 2.2.4, all three tasklets are numerical in nature, but this choice is due to the fact that so are most of the examples students are exposed to in class. Anyway, they are just based on simple arithmetic, familiar to students.

Figure 2.13 shows that the majority of the students interviewed (84.8%) are male; as proof of the still predominantly male character of the study of Computer Science.

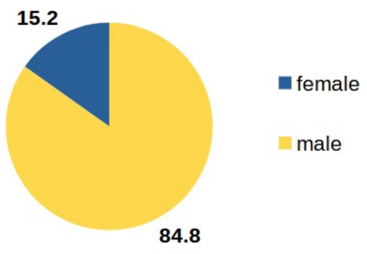


Figure 2.13: Gender of students in the sample.

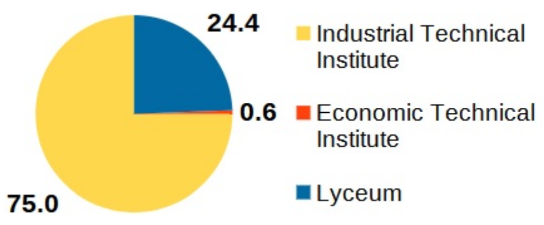


Figure 2.14: Students by type of school.

Students mainly attended the third year of upper secondary school (79.3%), see Figure 2.15.

Students' preferred languages are those they learn in school (C/C++, C#, Java[5]), although a good percentage (22.0%) say they prefer Python, demonstrating personal interests (Figure 2.16).

2.2.6 Results

We now present the main results of our investigation relative to the three tasklets and the two questions included in the survey. Overall, only about 8% of the students

[5]C/C++, C# and Java are currently the most used languages in Italian high schools.

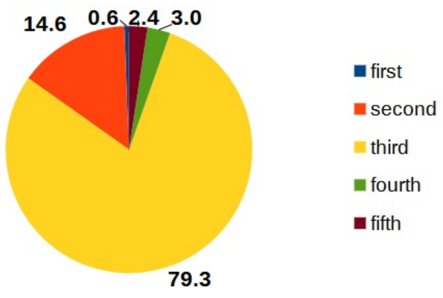


Figure 2.15: Students' year attendance.

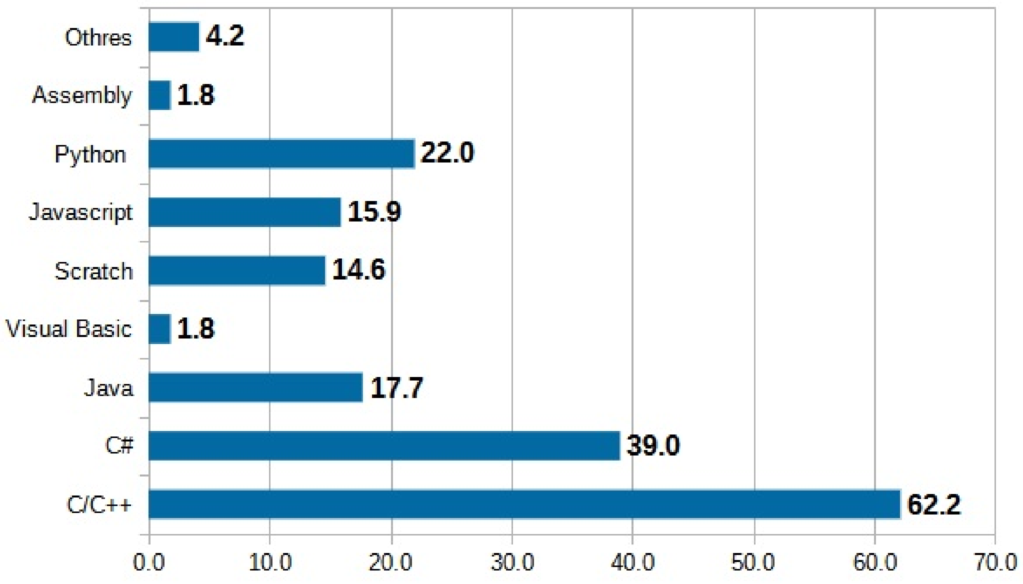


Figure 2.16: Students' favorite programming languages (more choices were possible).

solved all the three problems correctly, 27% provided two correct answers, 39% one correct answer, and 26% were wrong on all tasklets.

Tasklet 1: identifying the correct loop condition

Table 2.2 summarizes the results concerning tasklet 1. A little less than 40% of the students provided the correct answer, namely $2^k \leq n$, whereas about as many selected one of the two seriously wrong options, either $2^k = n$ or $2^k > n$. Although the flow-chart is rather simple, consisting of a very standard loop structure, and the problem specification is accurate, it turns out that students can easily be misled about the role or the interpretation of the loop condition.

Below (see Table 2.3) we compare the answers the students gave in the task and the answers they gave to the question regarding their difficulties when dealing with

Table 2.2: Rates of chosen options for tasklet 1.

Condition	**Percentage**	
$2^k = n$	3.7%	
$2^k \leq n$	38.4%	*correct option*
$2^k < n$	20.1%	
$2^k > n$	37.8%	

the iterations [6].

Table 2.3: Chosen options for tasklet 1 compared with students' difficulties about loops

Condition	**1**	**2**	**3**	**4**	**5**
$2^k = n$	0.00	0.00	1.83	1.22	0.61
$2^k \leq n$	4.27	9.15	17.68	5.49	1.83
$2^k < n$	4.88	4.88	7.32	2.44	0.61
$2^k > n$	6.10	9.15	14.63	3.66	4.27
	15.24	23.17	41.46	12.80	7.32

The data presented in Table 2.3 show that there is not a great correlation between incorrect answers to the problem, where the student had to indicate the correct condition of the loop, with the perception of the same, regarding difficulties in identifying the condition of the iteration.

Tasklet 2: ascertaining the number of iterations

As shown in Table 2.4, about 60% of the students chose the right option for tasklet 2, i.e. three iterations. It hence appears that a large majority of them is at ease with the functioning of iteration combined with a nested conditional, as well as with the interpretation of a composite (loop) condition including logical operators. It is conceivable that they identified the right option by tracing the code execution — which they probably did not try to do, on the other hand, to check their answer for tasklet 1.

As in the previous task, we compared (see Table 2.5) the answers to the problem with the answers the students gave in the task and the answers they gave to the question regarding their difficulties when dealing with the iterations. [6].

[6]For simplicity, in the table we have reported the number of the answer as presented in Fig.2.17: 1) Find the condition of the while loop or the do-while loop; 2) Define a complex condition, which uses logical operations (AND, OR, NOT, XOR); 3) Manage nested loops; 4) Understand, in general, when the loop has to stop; 5) Manage the variable that counts the loops.

Table 2.4: Rates of chosen options for tasklet 2.

Number of iterations	**Percentage**	
0	3.7%	
1	9.1%	
2	15.2%	
3	60.4%	*correct option*
4 or more	6.1%	
the loop never ends	5.5%	

Table 2.5: Chosen options for tasklet 2 compared with students' difficulties about loops

Number of iterations	**1**	**2**	**3**	**4**	**5**
0	1.22	0.00	1.22	1.22	0.00
1	2.44	0.61	2.44	1.83	1.83
2	3.05	4.27	6.71	0.61	0.61
3	7.93	15.24	25.61	7.32	4.27
4 or more	0.61	2.44	1.83	1.22	0.00
never ends	0.00	0.61	3.66	0.61	0.61
	15.24	23.17	41.46	12.80	7.32

In this case it is interesting to note the percentage (15.24%) of those who gave the correct answer and said that managing complex conditions was their greatest difficulty. It is worth noting that the difficulty was indicated by a low percentage of those students who answered incorrectly the task, which evidently means many students are unaware of their difficulties.

Tasklet 3: recognizing functionally equivalent programs

The rates of recurrent answers relative to tasklet 3 are listed in Table 2.6. It was clearly the hardest challenge and, as we can see, less than one fifth of the students were able to recognize that program 1 and program 4 are the equivalent ones. In addition, most of the answers grouped in the last row of Table 2.6 are meaningless, in that only one program was selected (about 30% of the whole sample), conceivably indicating that they just decided to skip this tasklet.

While such pairings as 1–3 or 4–5 (see Fig. 2.12) are likely to signal serious misconceptions, it is worth noting that regarding program 2 and program 4 as equivalent may be more simply ascribable to carelessness, i.e. not paying attention to the fact that the roles of `x` and `y` are swapped, but those of `m` and `n` are not. Here again, however, the frequency of incorrect answers indicates that students are not used to

Table 2.6: Rates of recurrent answers for tasklet 3.

Equivalent programs	**Percentage**	
Programs 1 and 4	18.9%	*correct answer*
Programs 4 and 5	13.4%	
Programs 2 and 4	11.0%	
Programs 1 and 3	7.3%	
Programs 1, 4, 5	3.7%	
Meaningless or isolated answers	45.7%	

test their conjectures by tracing code execution.

The comparison of the answers to the task and the answers students gave to the question regarding their difficulties when dealing with the iterations is presented in Table 2.7. [6].

Table 2.7: Answers for tasklet 3 compared with students' difficulties about loops

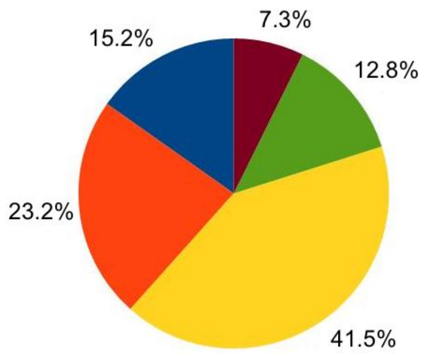
Equivalent programs	**1**	**2**	**3**	**4**	**5**
Programs 1 and 4	1.83	5.49	7.93	3.05	0.61
Programs 4 and 5	2.44	4.27	6.10	0.61	0.00
Programs 2 and 4	1.22	2.44	5.49	0.61	1.22
Programs 1 and 3	1.22	2.44	2.44	0.61	0.61
Programs 1,4,5	1.22	0.00	1.83	0.00	0.61
Programs 1,2,3,4,5	0.00	0.00	1.83	0.00	0.00
Others	12.20	8.54	12.80	6.71	3.66
	20.12	23.17	38.41	11.59	6.71

Again, there is no obvious correlation between the answers given to the question about the difficulties with the iterations and the answer to the task.

Subjective perception questions

The pie chart in Fig. 2.17 summarizes students' answers to the question: "*What do you find most difficult when you use loops?*" As we can see, nested loops are the source of issues reported most frequently (41.5%), followed by the definition of composite conditions (23.2%), the latter possibly due to insufficient familiarity with Boolean logic. Then the rates of the other options are, in decreasing order: figuring out a suitable loop condition (15.20%), understanding when an iteration should end (12.8%), and dealing with loop control variables (7.3%).

What emerges by comparing the subjective perception of difficulty (when dealing with iteration) to the actual performance in the three tasklets is that students may



- 🟦 To find the condition of a `while` or `do-while` loop (1);
- 🟥 To define a complex condition including logical operators (2);
- 🟨 To deal with nested loops (3);
- 🟩 To understand, in general, when the loop should end (4);
- 🟪 To deal with the loop control variable (5).

Figure 2.17: Major difficulty with iteration in students' perception.

underestimate their lack of mastery of loop conditions. If, on the one hand, more than 60% of them failed to choose the right option for tasklet 1 (in fact a straightforward condition), on the other only about 15% indicated the implied feature, i.e. identifying the loop condition, as a major source of difficulty.

Although the rate of choice of this feature is slightly higher (almost 18%) among those students who provided an incorrect answer for tasklet 1, there appears to be no statistically significant correlation between correct/incorrect answers to this tasklet and perceiving or not the related feature as a major source of difficulty, by cross-tabulating the corresponding counts (see Table 2.8) and subjecting them to a χ^2-test, we get a *p-value* of about 0.35, meaning that the data are fairly consistent with the assumption of independence of the two variables (*null* hypothesis). In addition, the limited awareness of difficulties with loop conditions is also signaled by the observation that just one out of the ten students who failed only on tasklet 1 seems to give prominence to the problem. More generally, as can be elicited "pictorially" from the partitioned bars in Figure 2.18, we cannot find any statistical evidence of correlation between poor performance in subsets of the tasklets and subjective perception of difficulty with specific concepts.

Table 2.8: Contingency table: correct/incorrect answers to tasklet 1 vs. perceived prominence or lack of prominence in difficulties with loop conditions.

	difficulties with loop conditions	**other difficulties**
incorrect answer to tasklet 1	18	83
correct answer to tasklet 1	7	56

To conclude the summary of the results of interest here, we consider the answers to the second question about the mistakes that have had more severe implications in the students' subjective perception. An inductive analysis [May14] of keywords

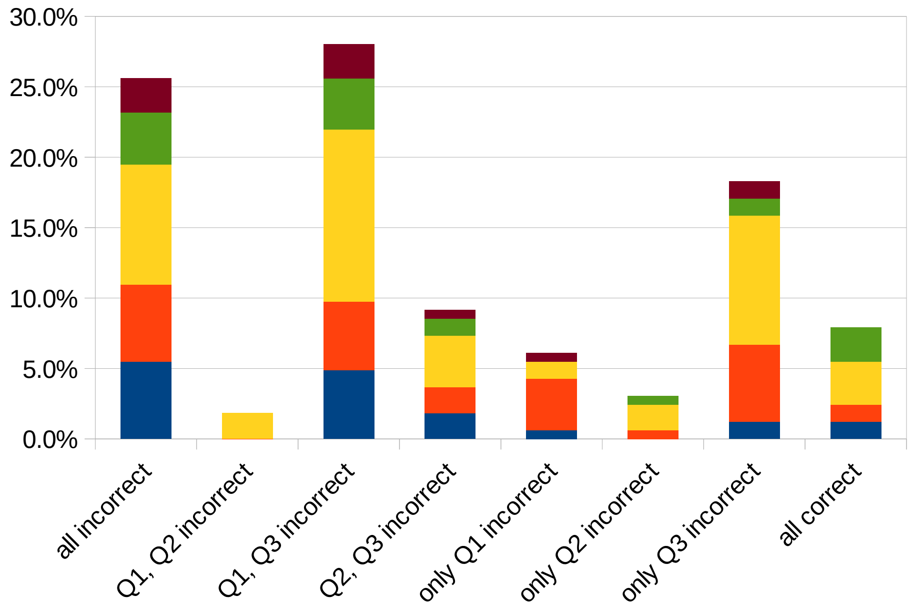


Figure 2.18: Distribution of students' perception of difficulty vs. performance; the interpretation of the colors is the same as in Figure 2.17.

occurring in the *open* answers gave rise to the categories summarized in Table 2.9. From the data in the right column it appears that out of 37.6% of students who identify some specific concept as a major source of mistakes, more than one quarter mention precisely iteration, confirming they are aware of the relevance of this topic to their learning of programming. Other reported concepts, with similar or lower frequency, refer to procedural and object abstractions, language syntax, mathematical and logical prerequisites.

Table 2.9: Major sources of mistakes in students' perception.

Sources of mistakes	**Percentage**
iteration, loops	10.4%
functions, subroutines	10.4%
syntax, instructions	7.3%
Mathematics and Logic	4.9%
objects, classes, methods	3.7%
general causes such as poor understanding of text, lack of time, insufficient practice, distraction	41.4%
elusive answers	11.6%
no answer provided	10.4%

Among the topics addressed, the students stated that it would have been prefer-able to spend more time on recursion (45.1%), see Figure 2.19, once again demonstrating the importance but also the difficulty in learning the specific topic.

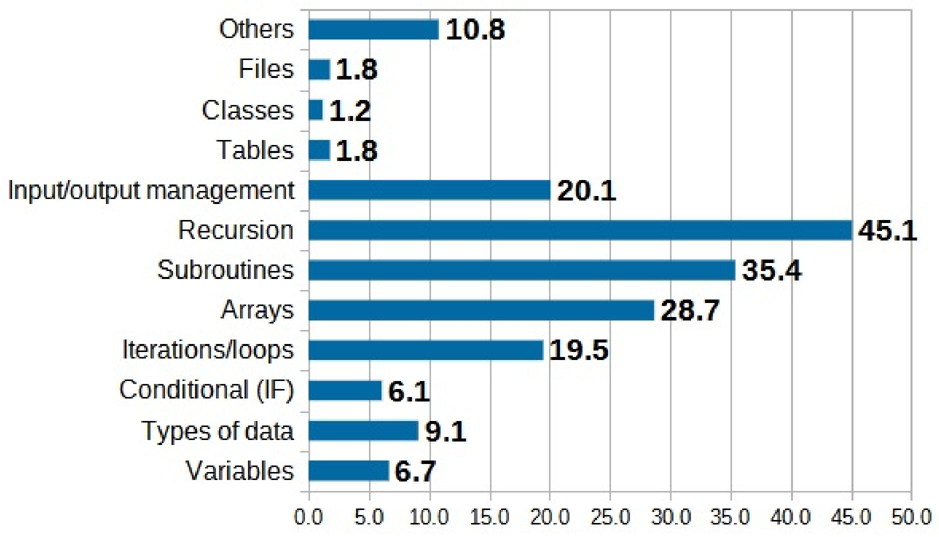


Figure 2.19: Topics to spend more time on (more choices were possible).

Figure 2.20 shows the topics that the students did not know adequately and therefore created difficulties in their programming activities. As we can see, the topics are mainly related to aspects of mathematical logic. Interestingly, more than one third of the students mentions specifically De Morgan's formulas as poorly understood.

2.2.7 Discussion

Based on the data we collected, as shown in the bars of Figure 2.18, almost two thirds of the students successfully completed no more than one of the three proposed tasklets. Mastery of iteration, even in relatively simple programs, is then to be considered a cognitively demanding learning objective for the considered age range. The results outlined in the previous section appear to corroborate, in the high school context, the findings of previous work addressing students' difficulties with conditionals and loops, e.g. [CZP14; Kac+10].

In a first task the students had to read carefully the statement of a very simple problem and identify the correct condition in a flow-chart by simply choosing among four options, the only difference being the relational operator in the loop condition: "$<$", "$\leq$", "$=$" and "$>$". Less than 40% of the students provided the correct answer ($\leq$), whereas about as many chose one of the two seriously wrong options ($=$ or $>$).

The second task asked to determine the number of iterations of a short code fragment for a given input. The loop was characterized by a composite condition (using two *and*s) and a nested *if-else*. In this case, about 60% of the students

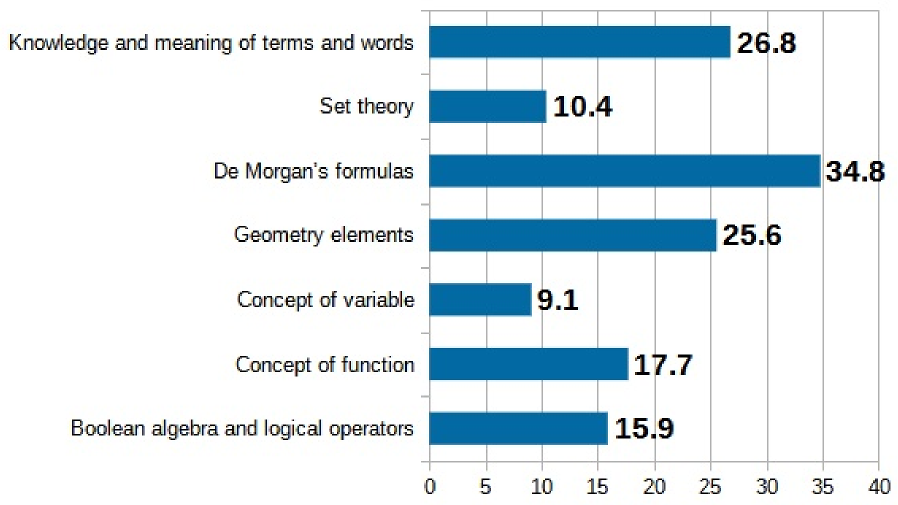


Figure 2.20: Topics that the students would prefer to understand more (more choices were possible).

identified the right answer (3 iterations). Finally, in the third task the students had to recognize functionally equivalent programs from a set of 5 items involving nested constructs (*if* and *while*) with simple conditions. Clearly, this task required a more comprehensive understanding of the effect of combining flow-control structures, and only less than 20% of the students were able to achieve it successfully. In particular, it appears that students' perception of difficulty with nested constructs is consistent with the actual state of affairs.

In particular, it turns out that dealing with nested flow-control structures and, perhaps to a minor extent, with loop conditions are especially challenging to novices, the former aspect also reflected in their subjective perception.

Here is a summary of our interpretation of the findings.

i. When required to trace the code execution, as in *tasklet 2*, a large majority of the students are able to determine the correct outcome, see Table 2.4. It is also conceivable that a number of the about 20% who opted for '2' or '4 *or more*' iterations made only minor computing mistakes. We can then presume that most high school students develop a viable and accurate enough mental model of the *notional machine* underlying code execution, including the functioning of nested constructs and the evaluation of relatively complex conditions.

ii. It is worth observing that 77% of the students who were correct in *tasklet 2* provided *seriously* wrong answers to *tasklet 1* or *tasklet 3*. Apparently, then, the students tend to not exploit their tracing abilities in order to test their conjectures about program behavior. This observation could be explained either by some general lazy attitude or, what is more relevant from a pedagogical perspective, by lack of method to approach programming tasks.

iii. As shown in Table 2.2, more than 40% of the students chose seriously incorrect options in *tasklet 1* (first and last option). A similar performance shows that, as a matter of fact, a large part of them are unable to master the relationships between loop condition and accurate specification in the application domain, even in a straightforward situation. This may possibly be ascribed to confusion about the role of the loop condition, meant as an 'exit' condition instead of a 'continue' condition, or to some more basic lack of problem-solving skills.

iv. Overall, the students seem to underestimate their difficulties in dealing with loop conditions — even simple conditions. On the one hand, they did not feel the need to check their solution to tasklet 1 by tracing the program execution for sample inputs (what could have been done very quickly). On the other hand, only a low percentage of those who made serious errors in *tasklet 1* and/or *tasklet 2* perceive their weakness in this respect as a major difficulty (see the chart in Figure 2.18).

v. By comparing students' performance in tasklets 2 and 3, it appears that their difficulties with nested constructs are not so much about the mechanics of code execution as about the ability of grasping code behavior at a more abstract level. So, the crucial point is how to develop students' abstraction skills, besides the understanding of the mechanical features of code (to illustrate which there are several widespread tools — see for example the paragraphs on program visualization in [LR+18]).

Implications for instructors

A few provisional implications for the instructional practice can be drawn from the points raised above. In particular, we point out three potential insights, which are worth further, more accurate investigation. By referring to a *competency framework* for computing education [Fre+18], the first two pertain to the *skills* area and the third one to the *dispositions* area:

- Firstly, more efforts should be made to develop a method to approach programming tasks, in particular to identify suitable test cases in order to confirm or refute working conjectures.
- Secondly, more careful attention should be paid to the role and treatment of loop conditions, especially in connection with the statement of a problem.
- Finally, at the meta level, students' attitude to think critically about their learning should be enhanced, for example by asking them to make explicit their degree of self-confidence in the achievement of a task or of a part of it.

Learning to program is however a slow and gradual process, as argued by Dijkstra in [Dij89], and therefore the teacher must grant adequate learning time to be spent on several effective examples.

Future work and perspectives

To begin with, in order to validate (or refute) our provisional interpretation of the findings discussed above, our next step has been to design a more comprehensive survey — described later in Chapter 4 —, to be administered to a larger sample of students.

We have also tried to envisage appropriate methodological approaches to the teaching and learning of iteration. In this respect, we thought that it would be helpful to collect a rich and varied set of examples, not limited to the stereotypical code patterns mentioned in the teachers' interviews [SM19], described in Appendix B. In particular, such examples should address "interesting" problems involving more complex loop conditions or (nested) combinations of flow-control structures.

As to the development of students' abstraction skills to interpret program behavior, a possible line of research could be based on De Raadt's and colleagues approach to explicitly teaching (and assessing) programming strategies [DRWT09a], which are relevant to a comprehensive understanding of nested constructs. Another potential source of inspiration in this respect may be the instructional work that elaborates on the concept of *loop invariant* [Tam92; Arn94; Gin03; FMV14], suitably adapted to fit less formal learning styles [Ast91].

As a further middle/long term objective, from a more general pedagogical standpoint, it may be interesting to explore the implications of the *productive failure* perspective [Kap16; LRR17] in a computing education context, especially in connection with the learner's self-confidence on the solution provided [Met17]. An investigation concerning the learner's self-confidence on the solution provided, in a set of tasks, it is been analyzed in the Chapter 4.

2.3 Teachers' vs. students' perception

Finally, teachers' opinion regarding students' difficulties are compared with students' perception concerning their own difficulties in programming.

As a further investigation, it is also interesting to compare students' perception and teachers' opinion of students' difficulties regarding learning Computer Science, and specifically concerning iteration (loop).

In the chart in Figure 2.21, where tightly related concepts have been aggregated, the percentage of teachers indicating concepts in a certain area is represented by the length of dark-red bars.

In the same chart, the "weights" of concepts are contrasted to their perceived difficulty for students and teachers, which pertain to the second section of the interviews and of the survey (see Figure 2.1).

Figure 2.22 shows a list of the languages used by the teachers, which are matched with a list of the languages preferred by the students; teachers could select only one language, students instead even more than one, hence the percentage difference.

By looking again at Figure 2.21, we can see the concepts that the students perceive (light-blue bars) and the teachers think of (orange bars) as serious learning obstacles. The chart should be self-explanatory. However, it can be observed that teachers are likely to underestimate the difficulties faced by some students with

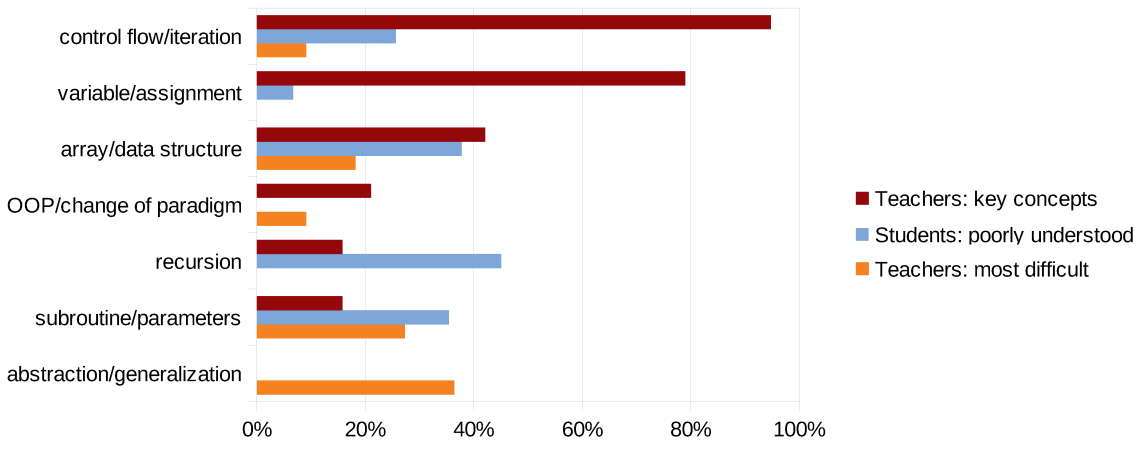


Figure 2.21: Key programming concepts for teachers and their difficulty in students' and teachers' perception. The absence of a visible bar means 0%.

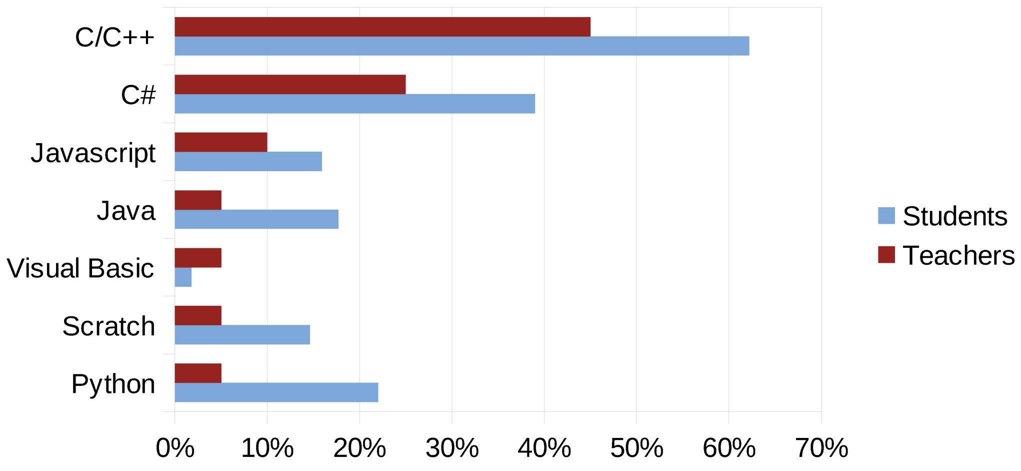


Figure 2.22: Favorite programming languages for teachers and students.

flow-control constructs. Moreover, they do not seem to pay much attention to the understanding of recursion, perhaps because they presume it is hardly within the grasp of most pupils. On the other hand, beside indicating a few of the key concepts taught, several interviewees emphasize the high-level thinking skills of abstraction and generalization (bottom bar).

Apart from recursion, according to the students, the hardest concepts are arrays, data structures and subroutines, i.e. precisely the last topics introduced by their teachers, shortly before the end of a school year. Students are often encouraged to compare their programs with those of their peers. Most of the teachers assign also unfamiliar tasks, when their students have reached a sufficient degree of mastery of programming basics — tasks which more than 60% of the students feel nevertheless quite confident to achieve and which can provide further motivation, particularly for high achieving students.

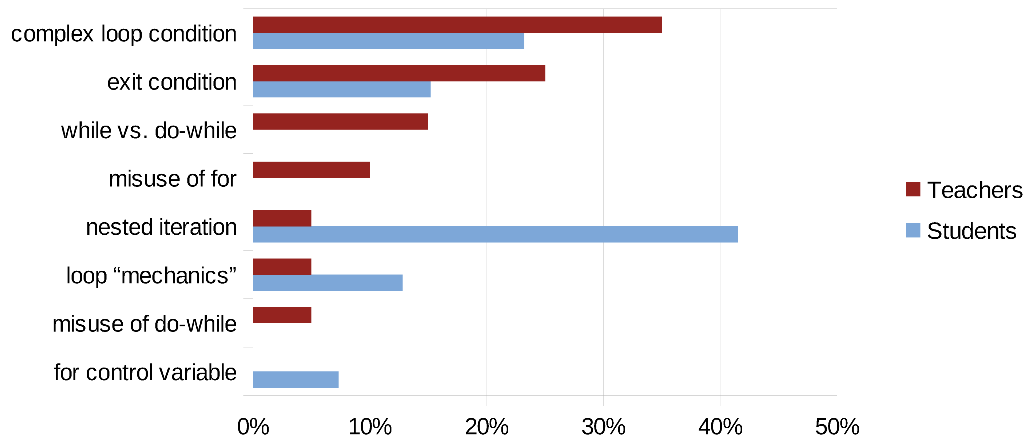


Figure 2.23: Major difficulty with iteration in teachers' and students' perception. Missing bars mean no available related option for students, 0% for teachers.

Focus on iteration

Figure 2.23 reports the collected data about the major sources of difficulties with iteration. To some extent, teachers and students agree on indicating the complexity of loop conditions (in terms of use of logical connectives) and the treatment of the exit condition as problematic. However, they seem to have contrasting views as to the other aspects addressed. On the one hand, a number of teachers point out students' misuse of iteration constructs, in particular *while* vs. *do-while* and the overuse of *for* loops in situations where it is not an appropriate choice. On the other hand, students give far more prominence to dealing with nested iterations — 42% of them report this being *the* major issue with iteration.

2.3.1 Comparison of teachers' vs. students' perceptions

Once established that iteration is among the few most central concepts for novice programmers, a claim with which all the interviewed teachers appear to agree, the first question to ask is if students' difficulties with loop constructs are *really* a major issue in the learning of programming. By considering this concept only from a general point of view, based on the teachers' and students' perception it is so only to some extent, whereas other programming concepts are seen as more challenging, see Figure 2.21. However, when we address the learning of specific features, such as the treatment of loop conditions or nested constructs, a different perspective emerges: students are not yet comfortable with these aspects.

As a matter of fact, students are aware of this, as the chart in Figure 2.23 shows, but their difficulties are also confirmed by the performance in the three programming-related tasks included in the survey, briefly outlined in the previous section. This is perhaps not too surprising, in light of the fact that most of the program examples they usually see in connection with iteration are quite straightforward (see the list in Section 2.1.5) and tend to induce the use of stereotypical patterns. Thus, to help students work with non-trivial loop conditions and neat combinations of flow-control constructs, it can be desirable to develop a catalog of significant examples presenting more varied and interesting structures.

As recognized by several interviewees (see again Figure 2.21), it is likely that the major issues depend on students' difficulties to take a more abstract, comprehensive perspective when dealing with programs. A possible way, identified by the teachers, to induce students to develop their abstraction skills is to contrast their tendency to approach a task by trial-and-error and require them to analyze the problem with paper and pencil.

A final issue, which emerged from the teachers' interviews and students' survey, concerns the learning of mathematical and logic prerequisites, which are part of the Mathematics syllabus, and are important in the learning of programming.

References

[Arn94] David Arnow. "Teaching Programming to Liberal Arts Students: Using Loop Invariants". In: *Proceedings of the 25th SIGCSE Symposium on Computer Science Education.* SIGCSE '94. New York, NY, USA: ACM, 1994, pp. 141–144.

[Ast91] Owen Astrachan. "Pictures As Invariants". In: *Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '91. New York, NY, USA: ACM, 1991, pp. 112–118.

[Bar+15] Erik Barendsen et al. "Concepts in K-9 Computer Science Education". In: *Proceedings of the 2015 ITiCSE on Working Group Reports.* ITICSE-WGR '15. New York, NY, USA: ACM, 2015, pp. 85–116. DOI: 10.1145/2858796.2858800.

[Bla16] Andreas Blass. "Symbioses between mathematical logic and computer science". In: *Annals of Pure and Applied Logic* 167.10 (2016). Logic Colloquium 2012, pp. 868 –878. DOI: https://doi.org/10.1016/j.apal.2014.04.018.

[BRT05] Susan Bergin, Ronan Reilly, and Desmond Traynor. "Examining the Role of Self-Regulated Learning on Introductory Programming Performance". In: *Proceedings of the 1st International Workshop on Computing Education Research.* ICER '05. New York, NY, USA: ACM, 2005, pp. 81–86.

[BSS13] Malte Buchholz, Mara Saeli, and Carsten Schulte. "PCK and reflection in computer science teacher education". In: *ACM International Conference Proceeding Series* (Nov. 2013). DOI: 10.1145/2532748.2532752.

[Cac+16] Ricardo Caceffo et al. “Developing a Computer Science Concept Inventory for Introductory Programming”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. New York, NY, USA: ACM, 2016, pp. 364–369. DOI: 10.1145/2839509.2844559.

[CZP14] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. “Identifying Challenging CS1 Concepts in a Large Problem Dataset”. In: *Proc. of the 45th ACM Tech. Symp. on Computer Science Education*. SIGCSE ’14. New York, NY, USA: ACM, 2014, pp. 695–700.

[Dij89] Edsger W. Dijkstra. “On the cruelty of really teaching computing science”. English. In: *Communications Of The Acm* 32.12 (1989), pp. 1398–1404.

[DMP15] Liesbeth De Mol and Giuseppe Primiero. “When logic meets engineering: introduction to logical issues in the history and philosophy of computer science”. eng. In: *History and Philosophy of Logic* 36.3 (2015), pp. 195–204.

[DRWT09a] Michael De Raadt, Richard Watson, and Mark Toleman. “Teaching and assessing programming strategies explicitly”. In: *Proceedings of the 11th Australasian Conference on Computing Education - Volume 95*. ACE ’09. Darlinghurst, Australia: Australian Computer Society, Inc., 2009, pp. 45–54.

[FMV14] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. “Loop invariants: Analysis, classification, and examples”. eng. In: *ACM Computing Surveys (CSUR)* 46.3 (2014), pp. 1–51.

[Fre+18] Stephen Frezza et al. “Modelling Competencies for Computing Education beyond 2020: A Research Based Approach to Defining Competencies in the Computing Disciplines”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 148–174.

[Ful+07] Ursula Fuller et al. “Developing a computer science-specific learning taxonomy”. eng. In: *ACM SIGCSE Bulletin* 39.4 (2007), pp. 152–170.

[Gin03] David Ginat. “Seeking or Skipping Regularities? Novice Tendencies and the Role of Invariants”. In: *Informatics in Education* 2 (2003), pp. 211–222.

[Gol+10] Ken Goldman et al. “Setting the Scope of Concept Inventories for Introductory Computing Subjects”. eng. In: *ACM Transactions on Computing Education* 10.2 (2010).

[HLR11] Orit Hazzan, Tami Lapidot, and Noa Ragonis. *Guide to Teaching Computer Science: An Activity-Based Approach*. 1st. Springer Publishing Company, Incorporated, 2011.

[IM20] Cruz Izu and Claudio Mirolo. "Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers". In: *Proc. of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '20. New York, NY, USA: ACM, 2020, pp. 466–472.

[Kac+10] Lisa C. Kaczmarczyk et al. "Identifying Student Misconceptions of Programming". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 107–111.

[Kap16] Manu Kapur. "Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning". In: *Educational Psychologist* 51 (Apr. 2016), pp. 1–11.

[Knu76] Donald E. Knuth. "Mathematics and Computer Science: Coping with Finiteness". eng. In: *Science* 194.4271 (1976), pp. 1235–1242.

[LMB08] John Loughran, Pamela Mulhall, and Amanda Berry. "Exploring Pedagogical Content Knowledge in Science Teacher Education". In: *International Journal of Science Education - INT J SCI EDUC* 30 (Aug. 2008), pp. 1301–1320. DOI: 10.1080/09500690802187009.

[Lop+08] Mike Lopez et al. "Relationships Between Reading, Tracing and Writing Skills in Introductory Programming". In: *Proc. 4th Int. Workshop on Comput. Educ. Research*. ICER '08. New York, USA: ACM, 2008, pp. 101–112.

[LR+17b] Andrew Luxton-Reilly et al. "Developing Assessments to Determine Mastery of Programming Fundamentals". In: *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. ITiCSE-WGR '17. New York, NY, USA: ACM, 2017, pp. 47–69. DOI: 10.1145/3174781.3174784.

[LR+18] Andrew Luxton-Reilly et al. "Introductory Programming: A Systematic Literature Review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 55–106.

[LRR17] Katharina Loibl, Ido Roll, and Nikol Rummel. "Towards a Theory of When and How Problem Solving Followed by Instruction Supports Learning". In: *Educational Psychology Review* 29.4 (Dec. 2017), pp. 693–715.

[May14] Philipp Mayring. *Qualitative Content Analysis: Theoretical Foundation, Basic Procedures and Software Solution.* Klagenfurt, 2014.

[Met17] Janet Metcalfe. “Learning from Errors”. In: *Annual Review of Psychology* 68 (Jan. 2017), pp. 465–489.

[Mir12] Claudio Mirolo. “Is Iteration Really Easier to Learn Than Recursion for CS1 Students?” In: *Proc. of the 9th Annual International Conference on International Computing Education Research.* ICER ’12. New York, NY, USA: ACM, 2012, pp. 99–104.

[MKB99] Shirley Magnusson, Joseph Krajcik, and Hilda Borko. “Nature, Sources, and Development of Pedagogical Content Knowledge for Science Teaching”. In: *Examining Pedagogical Content Knowledge.* Ed. by Julie Gess-Newsome and NormanG. Lederman. Vol. 6. Science & Technology Education Library. Springer Netherlands, 1999, pp. 95–132. DOI: 10.1007/0-306-47217-1_4.

[Mye90] J.Paul Myers. “The Central Role of Mathematical Logic in Computer Science”. In: *ACM SIGCSE Bulletin* 22.1 (1990), pp. 22–26.

[Qui14] Margareth Quindeless. “Logic in the curricula of Computer Science”. spa. In: *Revista AntioqueÃ+a de las Ciencias Computacionales y la IngenierÃ-a de Software (RACCIS)* 4.2 (2014), pp. 47–51.

[RW88] Charles Rich and Richard Waters. “The Programmer’s Apprentice Project: A Research Overview”. In: *Computer* 21 (Dec. 1988), pp. 10 –25. DOI: 10.1109/2.86782.

[Sae+11] Mara Saeli et al. “Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective”. In: *Informatics in Education* 10 (Apr. 2011), pp. 73–88.

[Sch12] Dane Schaffer. “An Analysis of Science Concept Inventories and Diagnostic Tests: Commonalities and Differences”. In: *Annual International Conference of the National Association for Research in Science Teaching.* Apr. 2012.

[Shu86] Lee S. Shulman. “Those Who Understand: Knowledge Growth in Teaching”. eng. In: *Educational Researcher* 15.2 (Feb. 1986), pp. 4–14.

[SM19] Emanuele Scapin and Claudio Mirolo. “An Exploration of Teachers’ Perspective About the Learning of Iteration-Control Constructs”. In: *Informatics in Schools. New Ideas in School Informatics.* Ed. by Sergei N. Pozdniakov and Valentina Dagienė. Cham: Springer, 2019, pp. 15–27.

[Tam92] Wing C. Tam. “Teaching Loop Invariants to Beginners by Examples”. In: *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’92. New York, NY, USA: ACM, 1992, pp. 92–96.

[Tur02] Sigita Turskienė. “Computer Technology and Teaching Mathematics in Secondary Schools”. eng. In: *Informatics in Education - An International Journal* 1.1 (2002), pp. 149–156.

[WD97] Roland Wagner-Dobler. “Science-Technology Coupling: The Case of Mathematical Logic and Computer Science.” eng. In: *Journal of the American Society for Information Science* 48.2 (1997), pp. 171–83.

[Wie89] Susan Wiedenbeck. “Learning iteration and recursion from examples”. In: *International Journal of Man-Machine Studies* 30.1 (1989), pp. 1 –22. DOI: `https://doi.org/10.1016/S0020-7373(89)80018-5`.

[ZH98] Rina Zazkis and Orit Hazzan. “Interviewing in mathematics education research: Choosing the questions”. eng. In: *Journal of Mathematical Behavior* 17.4 (1998), pp. 429–439.

Chapter 3

Teacher Survey

In this chapter we present the teacher survey and discuss the related findings. The survey, proposed by means of an online questionnaire to high school Computer Science teachers, is meant to get insights on their practice and instructional strategies to approach programming in general, and iteration in particular. After introducing the main aims, the scope of the investigation and the underlying methodology in the first two sections, in Section 3.3 we outline the organization of the instrument. The outcomes of the investigation are then summarized in Section 3.4 and discussed in Section: 3.5. Finally, we end this chapter with a few concluding remarks (Section 3.6).

3.1 Aims and scope of the teacher survey

The survey attempts to elicit some general features of teachers' PCK, in the context of programming, and focuses on the following research questions concerning specifically iteration:

- Which examples do high school teachers commonly use to introduce and explain the application of iteration constructs?
- What are their main instructional strategies to teach iteration?

3.2 Methodology

Interviewing teachers or asking them to complete a survey has become a popular way of data-collection in STEM fields [Her10; BS13; Fin+20].

The survey design has followed the principles elaborated in Chapter 2. The various questions formulated are not inspired by a single starting point, but we tried to identify questions from different points of view: PCK [Sae+11; BSS13], Content Representation [Bar+14], Concept Inventory, and Taxonomy [Ful+07]. Furthermore, questions have been formulated both considering outcomes obtained in previous teachers' interviews (see Chapter 2, Section 2.1) and — concerning students' difficulties and performance — previous students' survey (see Chapter 2, Section 2.2).

Regarding the survey design and analysis of the collected data, a useful inspiration came from the approach proposed by Rahimi at al. [RBH16]. The authors had focused on the elicitation and categorization of pedagogical content knowledge (PCK) versus the design of digital artifacts of Computer Science teachers. Their results suggested that teachers' PCK regarding design could be characterized in terms of two aspects: (i) teachers' knowledge about objectives and goals of designing digital artifacts by students; (ii) teachers' knowledge about ways to assess students' understanding and performance. As regards the first aspect (i), the authors distinguished an orientation towards more *conceptual* objectives and one towards more *practical* objectives. Regarding the second aspect (ii), they also found two types of teachers' knowledge, one focused on *process*-based evaluation and another on *product*-based evaluation.

In the analysis of the collected data that follows, orientations towards conceptual objectives, rather than practical objectives, have been distinguished. In addition, process-based assessment has been distinguished from product-based assessment.

This distinction made it possible to analyze the teachers' orientation, differentiating between conceptual objectives (abstraction, computational thinking, etc.) and practical objectives, more related to coding and software production.

3.3 Instrument

To begin with, the structure of the survey protocol, partly inspired by the approaches to elicit the PCK presented in the literature [Sae+11; VDB12; BSS13; Bar+14; RBH16], is outlined in Figure 3.1, where the most significant questions are reported verbatim. A set of questions (point 1) is aimed at collecting general information regarding teachers. Other questions (point 2) attempt to ascertain general context of introductory programming and the related prerequisites. Questions focused on iterations (point 3) aimed to investigate how central the learning of iteration is in the teacher's perspective, concerning teaching methodology, students' learning and strategies to foster learning. Then, the focus moves to a pedagogical approach (point 4) to enhance and stimulate students' learning. Assessment of iteration (point 5) is investigated regarding general aspects and specific solutions. Finally, the survey section concludes with students' aptitudes (6) regarding learning Computer Science in general and an open question (7) to collect further ideas, concerning teaching and learning iterations, not covered in the previous points.

The survey addressed to teachers includes 22 questions, and the first 5 questions deal with general information. Three questions concern learning programming in general, while six focus on iteration (teaching, learning and strategies to foster learning). Furthermore, two questions investigate learning pedagogical approaches, whilst four more concern students' learning assessment. Students' aptitudes for learning Computer Science are investigated with one question. Finally, there is one additional question concerning observations, regarding teaching and learning iterations, not highlighted in previously questions.

1. General information (5 questions)	
master degree, discipline taught mainly, type of school where they mainly teach, gender information, years of teaching Computer Science	
2. Learning programming in general (3 questions)	
2.1. teaching	How important do you think the following prerequisites are in order to understand the basic concepts of Computer Science?
	Are there any additional mathematical or scientific concepts that you consider important as prerequisites? Which ones?
2.2. learning	How would you rate the level of difficulties usually faced by students when learning each of the following programming concepts?
3. Focus on iteration (6 questions)	
3.1. teaching	Do you use any of the following examples to explain the iteration constructs? (More options were possible)
	Can you describe one or two examples you show in class to explain the WHILE loop?
	Can you describe one or two examples you show in class to explain the FOR loop?
	Can you describe one or two examples you show in class to illustrate the use of nested loops?
3.2. learning	In general, how would you rate students' mastery of iteration constructs?
3.3. strategies to foster learning	In your concrete teaching experience, how important do you consider the following instructional strategies to foster the learning of iteration constructs?
4. Pedagogical approach (2 questions)	
4.1. learning	In your concrete teaching experience, how important do you consider the following activities to enhance and consolidate learning of programming?
	To what extent do you consider the following "stimuli" important to enhance student motivation?
5. Assessment (4 questions)	
5.1. learning	In your concrete experience, how important do you consider the following aspects in order to assess students' learning of programming?
	How do you assess students' solutions that work but are structurally involved, i.e., significantly more complex than needed?
	How do you assess students' solutions that work but are computationally inefficient, i.e., poorly performing?
	In your practice, by what means do you usually assess the learning of iteration constructs? (More options were possible)
6. Students' aptitudes (1 question)	
Important students' dispositions for learning Computer Science	
7. Other thoughts (1 question)	
Any additional observations to consider in connection with the teaching and/or learning of iteration constructs	

Figure 3.1: Structure of the teacher survey protocol.

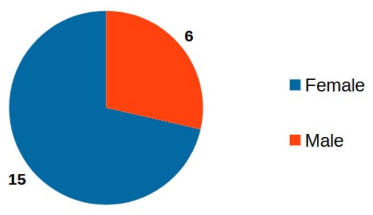
An English translation of the complete set of questions, submitted to the teachers, is available via this link `http://nid.dimi.uniud.it/additional_material/teacher_survey.html`.

3.4 Data collection and results

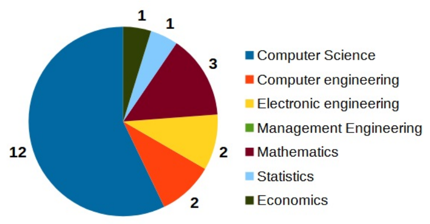
We conducted an anonymous online survey with 21 experienced high school Computer Science teachers, working in technical institutes and lyceums in Italy, probably several of the teachers previously interviewed responded to the survey. We had invited, via email, many CS teachers, from various Italian regions, and we therefore expected a greater number of contributions; probably the well-known difficulties encountered in managing the Covid-19 pandemic in schools discouraged participation.

3.4.1 General information

The teachers who responded to the survey are predominantly male (15 teachers), see Figure 3.2a, and are mostly Computer Science graduates (12 teachers), see Figure 3.2b.



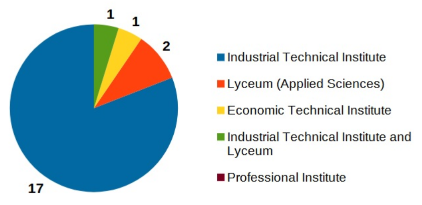
(a) Teachers by gender



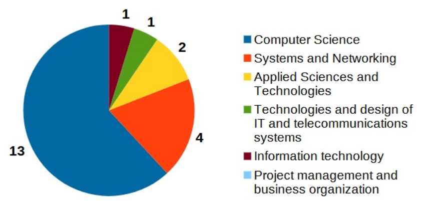
(b) Teachers' level of education

Figure 3.2: Teachers' gender and level of education.

The type of school and the subject taught are presented below. The teachers consulted teach mainly in the Industrial Technical Institute (17 teachers), see Figure 3.3a, and mainly they teach Computer Science (13 teachers), see Figure 3.3b.



(a) Type of school



(b) Subject taught

Figure 3.3: Teachers' service school and subject taught.

Computer Science teaching years are adequately distributed in the sample (Figure 3.4).

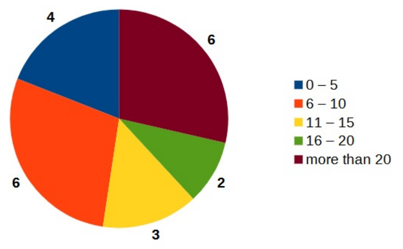


Figure 3.4: Teachers by years of service.

3.4.2 Learning programming in general

We asked to evaluate the level of difficulty that normally a student of an introductory course encounters when they learn and become familiar with some fundamental concepts, and what emerged is shown in Figure 3.5. Interesting indications emerge:

- *Variables and assignment* as well as *Elementary data types* are considered easy for students to learn;
- *Pointers and dynamic memory management*, *Handling of exceptions*, *Recursion*, *Event management*, *OOP: classes and objects*, *OOP: instance variables and encapsulation* and *OOP: constructors and methods* are indicated as concepts that are difficult to learn by students. In this case the prevailing indication is not to deal with them in the introductory courses;
- *Conditional (if)* is considered a relatively simple concept while *Iteration (loop)* has a higher degree of difficulty for the students;
- *Arrays and strings*, *Scoping and lifetime of variables*, *Functions and procedures*, *Passing parameters* and *File management* are considered relatively simple, although they can create difficulties for some students.

Once again it emerges that iterations are a concept that proves rather difficult to learn for students, as discussed in the previous chapters, see Chapter 1 and Chapter 2, and here investigated with both more structured and better targeted questions. In fact, the pilot interviews was quite heterogeneous, and furthermore literature focused on tertiary education.

Another general relevant issue concerns the extra-computing prerequisites, see Figure 3.6. A large number of teachers (12 very important, 8 quite important) refer to the mathematical/logic background (*Boolean algebra* and *Mathematical logic*) as well as to *text comprehension*[1], *Precision/formal rigor*, *General problem solving skills* and *Understanding of a formalism*[2], are all important extra-computing prerequisites.

Some teachers have also pointed to other mathematical or scientific concepts that they consider important as prerequisites:

[1]Reading comprehension is the most important prerequisite for 16 out of 21 teachers.

[2]Even the understanding of a formalism is probably referable to mathematical pre-requisites.

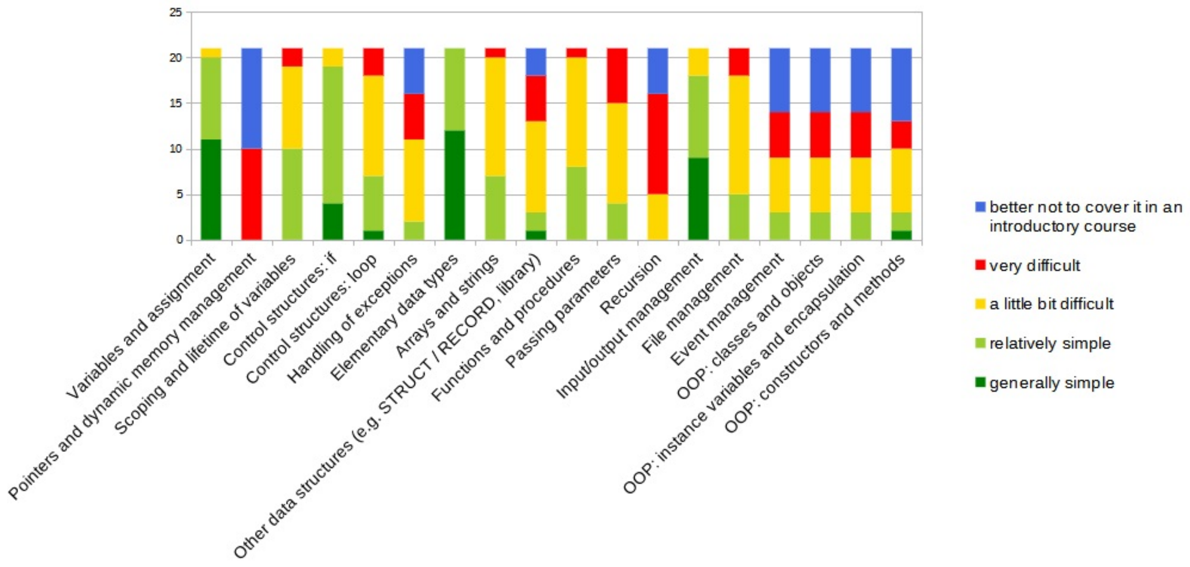


Figure 3.5: Main programming key concepts.

- expertise in formal proofs, being able to prove a theorem, algebraic or geometric, in an autonomous way;
- mastery of operations, symbols, equality, inequality;.
- good mathematical and logical bases in general;
- ability to look for the error and desire to solve it.

Concerning the ability to prove a theorem, it should be noted that as it is no longer required in upper secondary school Mathematics programs, the students' reduced ability to manage formal proofs independently is justified.

Furthermore, a single teacher stated that *"recently, students (of first year in upper secondary school) find it difficult to choose the correct relationships between natural numbers"*.

3.4.3 Focus on iteration

Teaching

Teachers use examples to explain the constructs for iteration, the most used are listed in Figure 3.7 (multiple options could be chosen).

The examples proposed are in line with what has already emerged in the interviews analyzed in Chapter 2, Section 2.1.5. It can be noted that most of the examples proposed are problems that can be solved through *For*-loop. In fact there are few that can be solved exclusively with *While*-loop or *Do-While*-loop, and just as few can be solved with *nested iterations*.

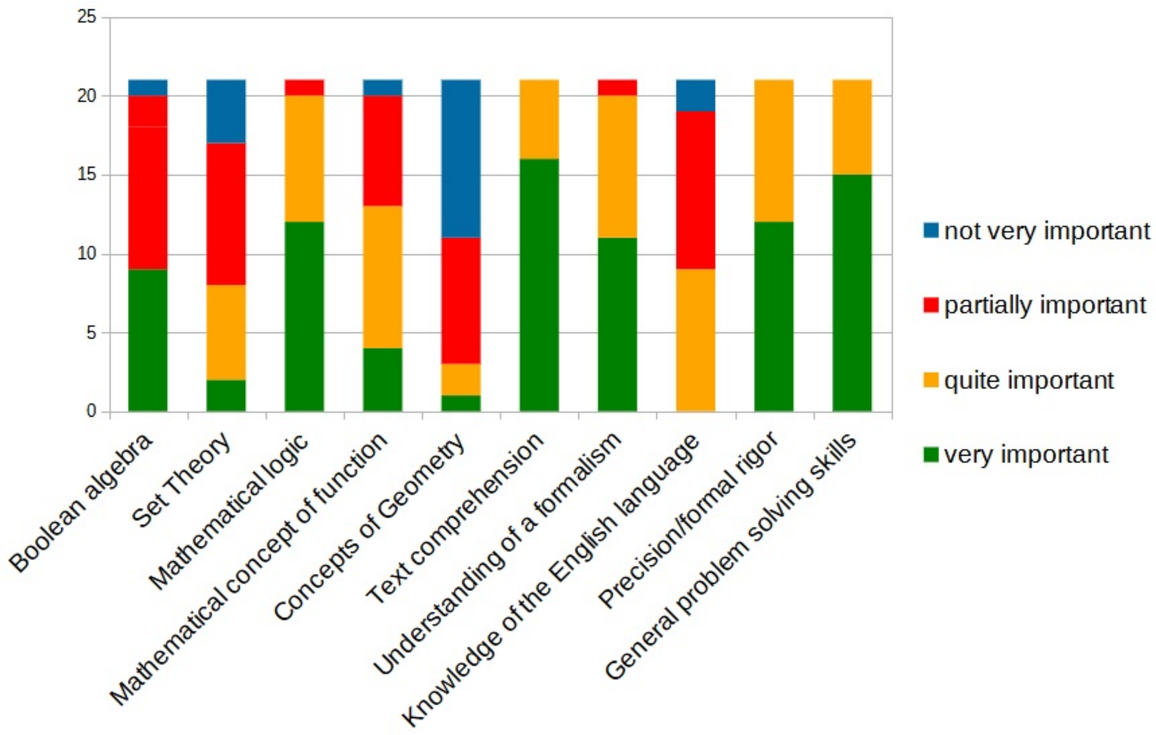


Figure 3.6: Extra-computing prerequisites.

The teachers have described one or two examples they propose in class to explain the While-loop.

- Given a number in input, print the sum of the first n numbers.
- Enter the measurement of the side of some geometric figure from the keyboard. Repeat the entry if the measurement is negative. It is not known how many times the user will make mistakes and therefore the data must be entered again as long as the user continues to make mistakes.
- Example 1: request an input datum and permanence in the cycle, requesting it again, as long as the entered datum is not adequate (for example a datum greater than or equal to zero or included in an interval).
  Example 2: request a series of input data and permanence in the cycle, requesting one more, as long as the entered data is not equal to a symbol associated with the termination (ask the user to provide a series of values as long as the user writes the symbol '0').
- Example 1: given a sequence of playing cards (only the values, terminated by 0), find the one with the highest value.
  Example 2: given a maximum value K, print the nursery rhyme *"an elephant swayed …"* up to K.
- Situations where the body of the cycle may not even be carried out (null sequences

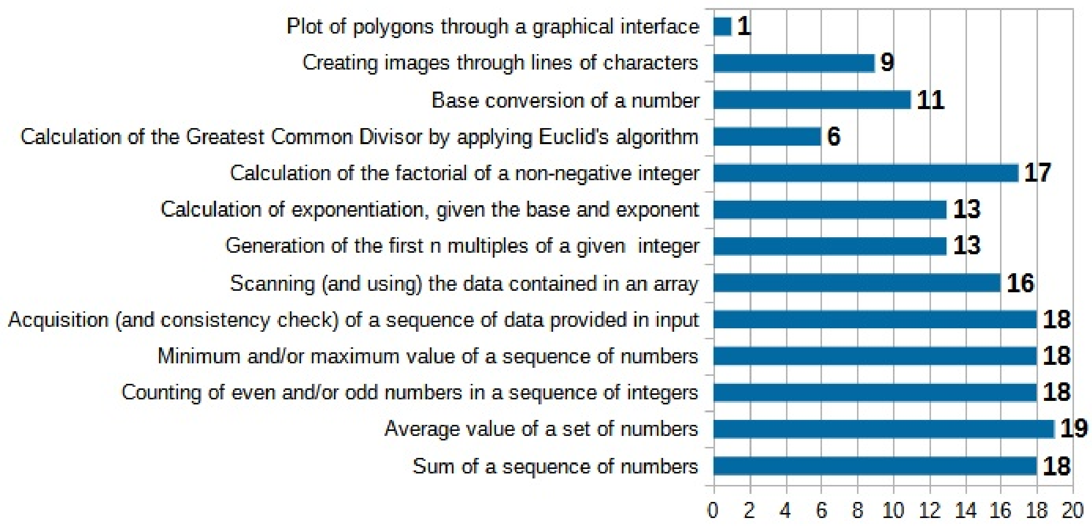


Figure 3.7: Examples to explain iteration, multiple options could be chosen.

or conditions that may not occur).

- Division, calculation of the remainder, by subtraction.
- Recipe phase: fill the pot with water, pour and repeat until you have exceeded the desired level.
- Rolling dice until a predetermined value is reached.
- Search for the first occurrence in a set of elements; in this case the iteration condition is not a simple condition.
- Ask for numbers and add them until a certain maximum value has been reached; type numbers until you guess a randomly generated number.
- Checking the input of one or more variables, checking a termination of a sequence of numbers.
- Sum of a sequence of numbers, primality test.
- Given a sequence of numbers ending with 0 calculate the number of elements inserted.
- The creation of a simple menu.
- *"In the second classes I introduce the While construct by proposing the sum of numbers. In the three years when I talk about preconditional iterative constructs I use the example of Fibonacci rabbits".*
- Reading a sequential file.
- Example 1: sum of n numbers read in input, where n is in turn read in input.
  Example 2: create a program that reads a positive real number in input, and requests it in input as long as the user enters a negative real number.
- Acquire a series of values until a particular value is entered; search for an element in a sequence.

- Reading a sequence of values with termination character.

Several of the examples proposed are tasks that can be easily solved using the For-loop. Moreover, many examples, if solved with While-loop, require simple termination conditions. In other words, the tasks can be solved with simple iterations, which make use of an iterator variable, knowing a priori the number of loops to execute.

Furthermore teachers have described one or two examples they propose in class to explain the For-loop.

- Display a greeting phrase on the screen N times, first with constant N, then with variable N, display N odd or even numbers on the screen.
- Example 1: request two numeric inputs and return of a message that compares the value of the pair of numbers (both strictly greater than zero, only one strictly greater than zero, none greater than zero, etc., etc. with further mixed cases and/or with equality).
  Example 2: request a numeric input corresponding to a month of the year and return of the number of days of the month entered.
- Average grades or students' heights in a class.
- Example 1: given a string, count the number of occurrences of a letter.
  Example 2: given the daily temperatures of a city over a week, calculate the average.
- Problems that make use of a counter (sequences with predefined length).
- Multiplication tables.
- Drawing of geometric figures with '*'.
- Example 1: management of N values introduced by the user.
  Example 2: array scan.
- *"Ask an operator for N numbers (with N defined) and calculate their sum, product, minimum, maximum. Explaining the For-loop first (which I believe to be simpler than the While-loop) typically offers good educational results".*
- Inserting and displaying elements of an array - Calculation of the power of a number.
- I resume the examples with the while loop by translating them with the for loop.
- I calculate the average of the grades of N students.
- Searching for data in an array.
- Average of N votes entered by the user, display of a Pythagorean table.
- Scan of the elements of a array.
- Calculate the perimeter of an irregular polygon; search for the maximum in a sequence of numbers.
- Iterative factorial.

Again, these are fairly simple problems, which do not require nesting.

Regarding the examples proposed in class to illustrate the use of nested iterations, the teachers described what is listed below.

- Matrix of multiplication tables.
- Scanning of a matrix with suitable termination conditions.
- Example 1: generate the first N lines of the multiplication table.
  Example 2: K friends play football M times a week: given as input the number of goals scored by each in each match, calculate who scored the most.
- The Console drawing of flat figures, e.g. flags, Christmas trees, houses, etc. with some particular characters (e.g. '*') and with different colors.
- Inserting and displaying the elements of a matrix - Sorting algorithms.
- Print all possible pairs or triples of a set of elements.
- For N numbers entered by an operator, calculate the factorial for each of them.
- Check if a string contains values from a sequence of characters. Validate a stack.
- Sorting an array.
- Creation of images composed of characters (e.g. full/empty square, pyramid, checkerboard, etc.).
- Search characters in strings.
- Multiplication tables, creation of geometric designs such as squares, rectangles, triangles etc.
- Example 1: request two numeric inputs and return of a message that compares the value of the pair of numbers (both strictly greater than zero, only one strictly greater than zero, none greater than zero, etc., with further mixed cases and/or with equality).
  Example 2: request a numeric input corresponding to a month of the year and return of the number of days of the month entered.
- Addition table.
- Given two arrays of integers, calculate how many numbers of the first array are also present in the second array.
- Given a class, build the scoreboard (the N marks of the subjects must be entered for each pupil of the class).

The proposed examples deal with nesting mainly with reference to the use of for-loop. There are only few examples where the while-loop, as well as the identification of the appropriate termination condition, are necessary.

The teachers gave a degree of importance to the following teaching strategies:

1. Computation analysis performed by an iterative program by tracing the values of the variables in sample cases
2. Insertion in the iterative program of instructions to print or display information on the processing status
3. Analysis of the computation performed by an iterative program in terms of a flowchart
4. Analysis of the execution conditions and the role of the first and/or last iteration of a loop
5. Analysis of the termination of an iterative program
6. Analysis of transformation schemes between functionally equivalent programs that apply different iterative constructs
7. Comparison of iteration and recursion
8. Documenting and applying style criteria to iteration-based code development
9. Analysis of the function performed by an iterative construct in terms of invariant properties of the loop
10. Analysis of the behavior of an iterative construct in terms of computational efficiency
11. Debugging of iteration-based programs for the identification and correction of artificially introduced errors for educational purposes
12. Choosing and applying appropriate test data to verify the functionality of an iteration-based program
13. Integration of user error management in the iterative program
14. Comparison (advantages / disadvantages) of alternative solutions, based on iteration, in terms of style, documentation, readability, efficiency ...
15. Learning the tools of a development environment (IDE) useful for the development of iterative code
16. Discussion of the generality of the iteration in the context of problems that admit algorithmic solutions

With practical teaching experience in mind, teachers consider the following teaching strategies (see Figure 3.8) to be important for learning the constructs for iteration.

- *Computation analysis performed by an iterative program by tracing the values of the variables in sample cases*,
- *Insertion in the iterative program of instructions to print or display information on the processing status*,
- *Analysis of the computation performed by an iterative program in terms of a flowchart*,
- *Analysis of the execution conditions and the role of the first and/or last iteration of a loop*,

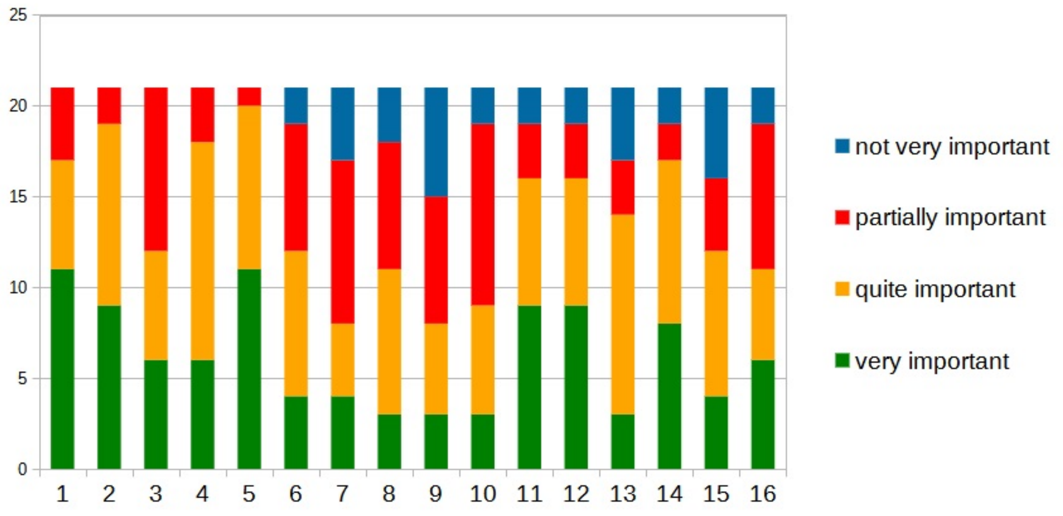


Figure 3.8: Teaching strategies for learning iteration.

- *Analysis of the termination of an iterative program.*

Less importance is given instead to the following didactic strategies:

- *Analysis of transformation schemes between functionally equivalent programs that apply different iterative constructs,*
- *Comparison of iteration and recursion,*
- *Documenting and applying style criteria to iteration-based code development,*
- *Analysis of the function performed by an iterative construct in terms of invariant properties of the loop,*
- *Analysis of the behavior of an iterative construct in terms of computational efficiency,*
- *Debugging of iteration-based programs for the identification and correction of artificially introduced errors for educational purposes,*
- *Choosing and applying appropriate test data to verify the functionality of an iteration-based program,*
- *Integration in the iterative program of instructions to handle user errors,*
- *Comparison (advantages / disadvantages) of alternative solutions, based on iteration, in terms of style, documentation, readability, efficiency etc.,*
- *Learning the tools of a development environment (IDE) useful for the development of iterative code,*
- *Discussion of the generality of the iteration in the context of problems that admit algorithmic solutions.*

Analyzing the data in the perspective of distinguishing two dimensions — taking inspiration from the work of Rahimi et at. [RBH16] —, which outline an orientation towards more conceptual objectives and one towards more practical objectives, some additional results can be identified.

Figure 3.9 highlights that teachers assign greater importance to practical objectives than conceptual objectives. In particular, great importance is assigned to the following strategies, which pursue practical objectives:

– Computation analysis performed by an iterative program by tracing the values of the variables in sample cases
– Insertion in the iterative program of instructions to print or display information on the processing status
– Debugging of iteration-based programs for the identification and correction of artificially introduced errors for educational purposes
– Choosing and applying appropriate test data to verify the functionality of an iteration-based program
– Comparison (advantages/disadvantages) of alternative solutions, based on iteration, in terms of style, documentation, readability, efficiency ...

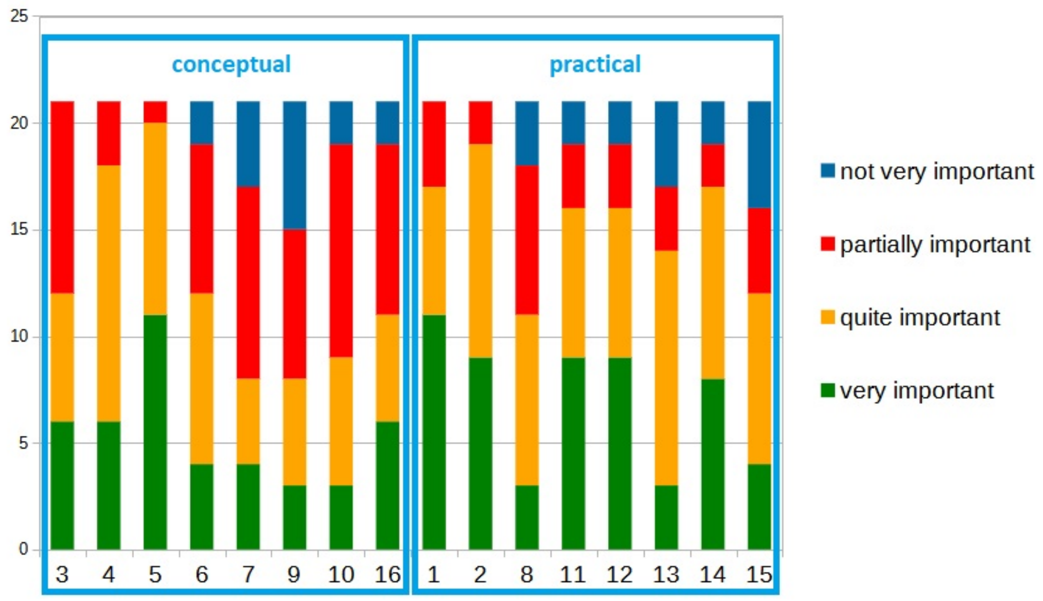


Figure 3.9: Two dimensions teaching strategies for learning iteration.

Among the strategies that pursue conceptual objectives, only the following is indicated as very important:

- Analysis of the termination of an iterative program

This finding could be justified by the fact that the majority of teachers teach in technical high school.

Learning

In general, teachers evaluate students' competence in iteration constructs in the way presented in Figure 3.10. From the diagram, it clearly emerges that students are more proficient with *For*-loop, while they have less mastery when they have to *"manage the first and/or last iteration of a While loop"* and in the *"management of the first and/or last iteration of a Do-while/Repeat loop"*. This is little surprising when most of the teachers' examples are also based on *For*-loops. Moreover, students are not adequately skillful in:

- *the understanding and handling of choice constructs (If) nested in loops,*
- *the understanding and handling of iterative constructs (While, For) nested in loops,*
- *the distinction between While and Do-while/Repeat loops,*
- *the choice of the type of iteration to use (While, Do-while/Repeat, For).*

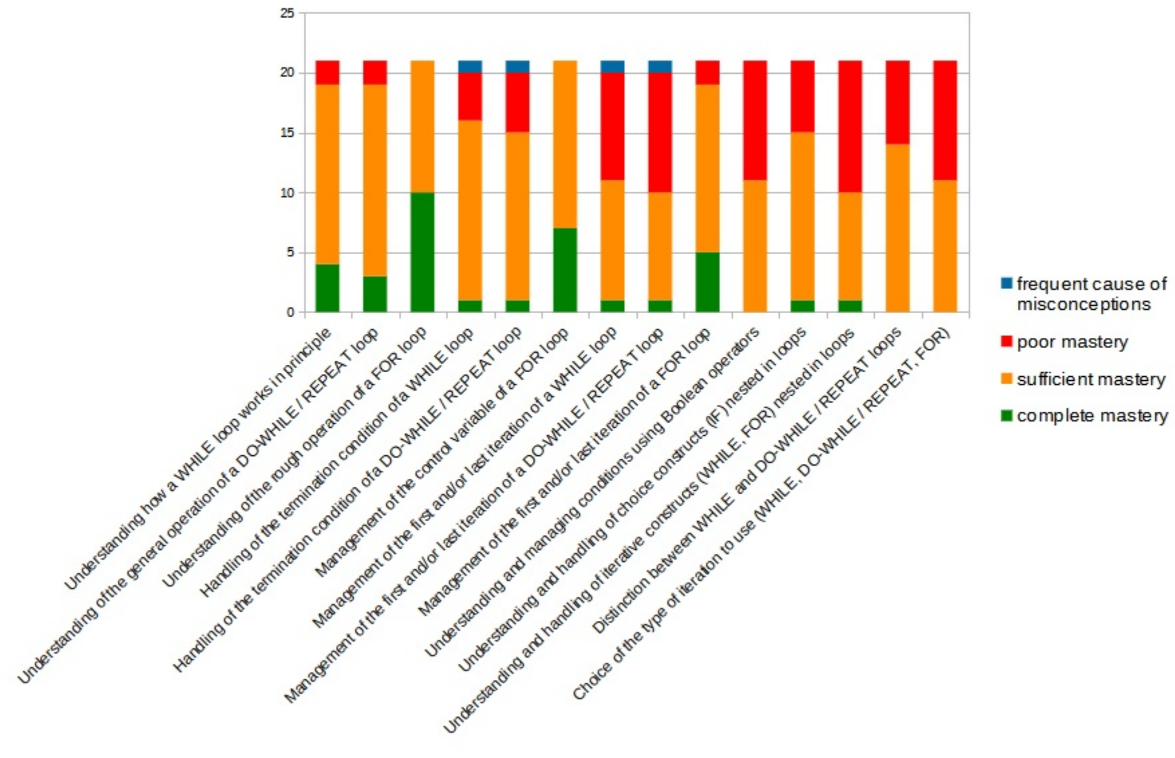


Figure 3.10: Students' mastery of iteration constructs.

3.4.4 Pedagogical approach

In the concrete teaching experience, activities aimed at improving and consolidating the learning of programming in students are considered important. A list of proposed activities, to improve and consolidate students' learning, is presented below.

1. Accurate preliminary analysis of the problem “with pen and paper”
2. Reading and understanding (possibly completion) of well written programs in terms of style and clarity
3. Illustration of algorithmic techniques through animations
4. Collective discussion of solutions with original characteristics
5. Regular assignment of homework
6. Regular laboratory activities
7. Laboratory activities in pairs
8. Collaboration between peers through group projects
9. Discussion with the teacher on possible solutions
10. Explanation by the students of the degree of confidence in the solutions they have adopted
11. Comparison between peers on possible solutions
12. Peer evaluation (e.g. based on a predefined grid)

Consolidation and improvement activities were deemed important, as shown in Figure 3.11.

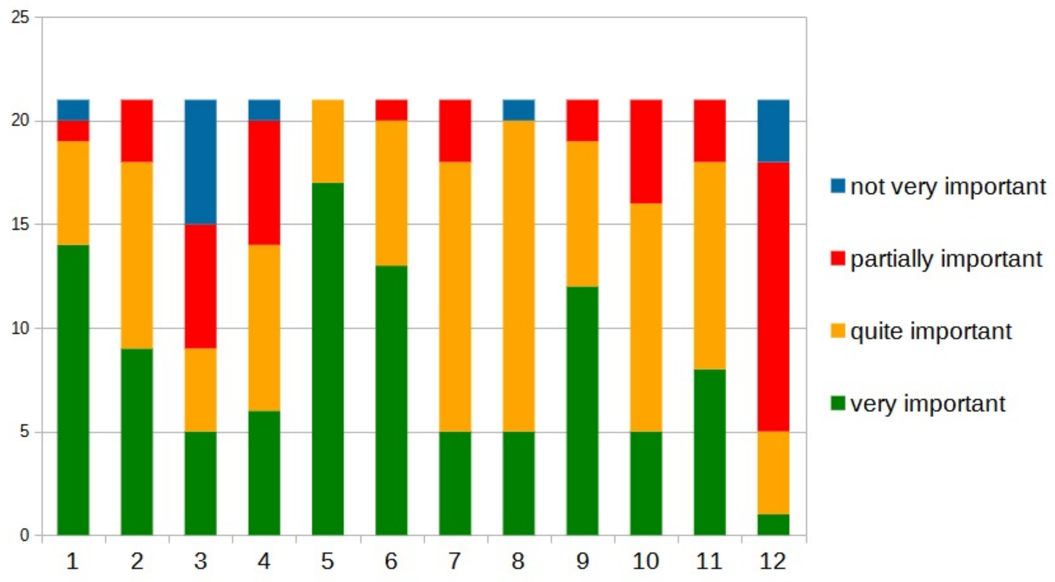


Figure 3.11: Activities aimed at improving and consolidating the learning of programming in students.

The following activities are, on average, important:

- *Accurate preliminary analysis of the problem “with pen and paper”*,

- *Reading and understanding (possibly completion) of well written programs in terms of style and clarity,*
- *Collective discussion of solutions with original characteristics,*
- *Regular assignment of homework,*
- *Systematic laboratory activities,*
- *Laboratory activities in pairs,*
- *Collaboration between peers through group projects,*
- *Discussion with the teacher on possible solutions,*
- *Explanation by the students of the degree of confidence in the solutions they have adopted,*
- *Comparison between peers on possible solutions.*

On the contrary, the following activities are considered of secondary importance: Illustration of algorithmic techniques through animations, Peer evaluation (e.g. based on a predefined grid).

Some additional results can be identified by analyzing the data while distinguishing two dimensions [RBH16], namely conceptual objectives and practical objectives. In terms of conceptual objectives, the activity *"Illustration of algorithmic techniques through animations"*, is identifiable, even though it is not considered very important (Figure 3.12). Instead, with regards to practical objectives the activity *"Collaboration between peers through group projects"* is identifiable, which is considered quite important.

In Figure 3.12 items 5,6,7 and 11 were not considered as they refer to students' learning styles in general, and they are not specific to Informatics.

In a further analysis that considers two other dimensions, one focused on *process*-based assessment and another on *product*-based assessment, three activities have been identified for each dimension, see Figure 3.12. With respect to the process objectives, the following activities were considered more important:

- *Accurate preliminary analysis of the problem "with pen and paper",*
- *Discussion with the teacher around possible solutions.*

Instead, with respect to the product objectives, the following activities were considered of primary importance:

- *Reading and understanding (possibly completion) of well written programs in terms of style and clarity,*
- *Collective discussion of solutions with original characteristics.*

It is interesting to note that the teachers did not consider *Peer evaluation* to be very important. This result is in contrast with what Hattie claims [Hat12], taking

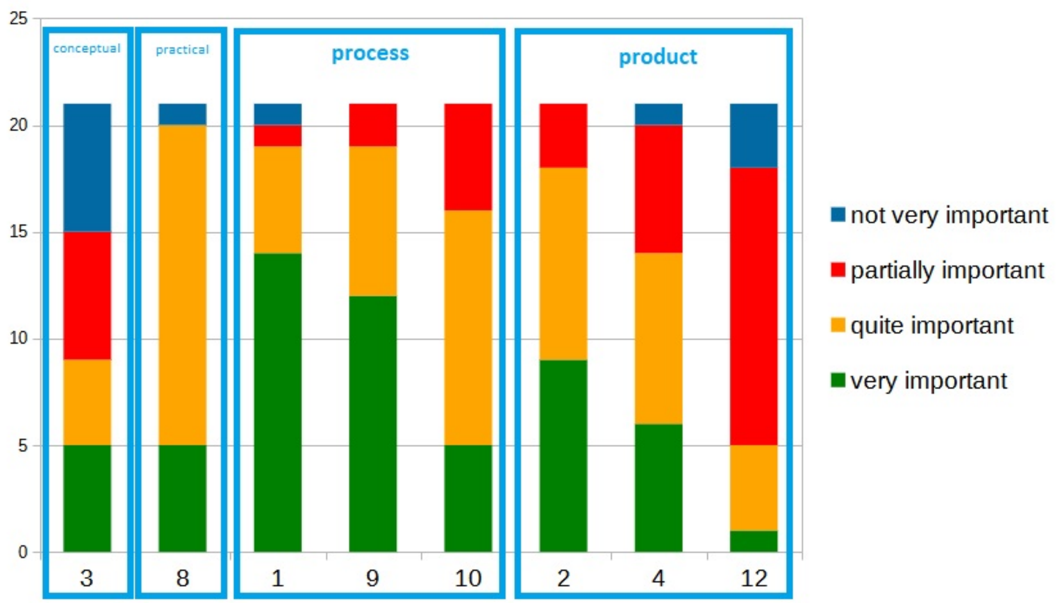


Figure 3.12: Activities, distinguishing dimensions, aimed at improving and consolidating the learning of programming in students.

up the studies by Nuthall [Nut07] and Sluijsmans et al. [SBGM02], regarding the importance of peer evaluation in learning processes.

Considering the stimuli to encourage students' motivation, the degree of importance attributed by teachers to the various options is shown in Figure 3.13.

A list of proposed stimuli, to encourage students' motivation, is presented below.

1. Feedback from the teacher during the intermediate steps of a program design
2. Game design and development
3. Design and development of graphic applications
4. Making applications that can be used by siblings, friends, etc.
5. Participation in competitions and games
6. Analysis and solution of concrete problems faced in the labor market
7. Collaboration in projects with companies
8. Study of solutions developed by professionals
9. Collaboration in peer projects in which the roles and responsibilities of each participant are clearly identified
10. Opportunity to illustrate the results of their work in the classroom through a multimedia presentation

The stimuli listed were suggested by what emerged during the teachers' interviews (see Chapter 2, Section 2.1).

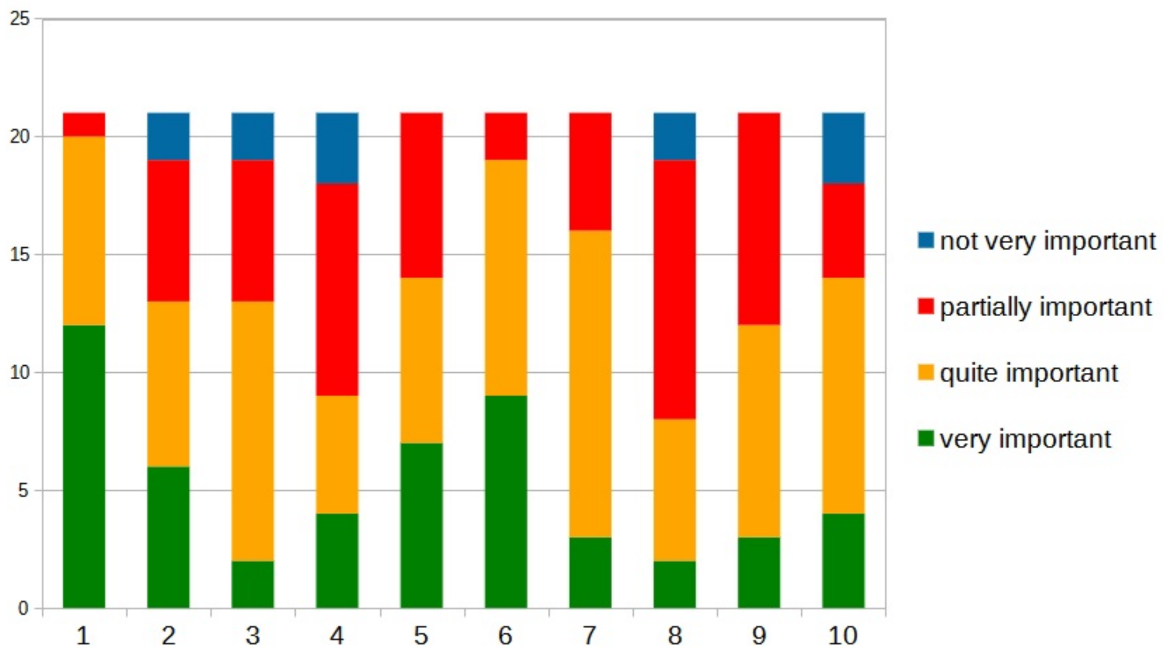


Figure 3.13: Stimuli to encourage students' motivation.

The most important stimulus to encourage and motivate students therefore seems to be *Feedback from the teacher during the intermediate steps of a program design*, as well as *Participation in competitions and games* and *Analysis and solution of concrete problems faced in the labor market.*

These stimuli are considered partially important:

- *Game design and development*,
- *Design and development of graphic applications*,
- *Making applications that can be used by siblings, friends, etc.*,
- *Collaboration in projects with companies*,
- *Study of solutions developed by professionals*,
- *Collaboration in peer projects in which the roles and responsibilities of each participant are clearly identified*,
- *Opportunity to illustrate the results of their work in the classroom through a multimedia presentation.*

In addition, with an analysis of the two other dimensions, one focused on *process*-based assessment and another on *product*-based assessment, three activities have been identified for each dimension, see Figure 3.14.

With respect to the process objectives, the following stimuli were considered more important than the others:

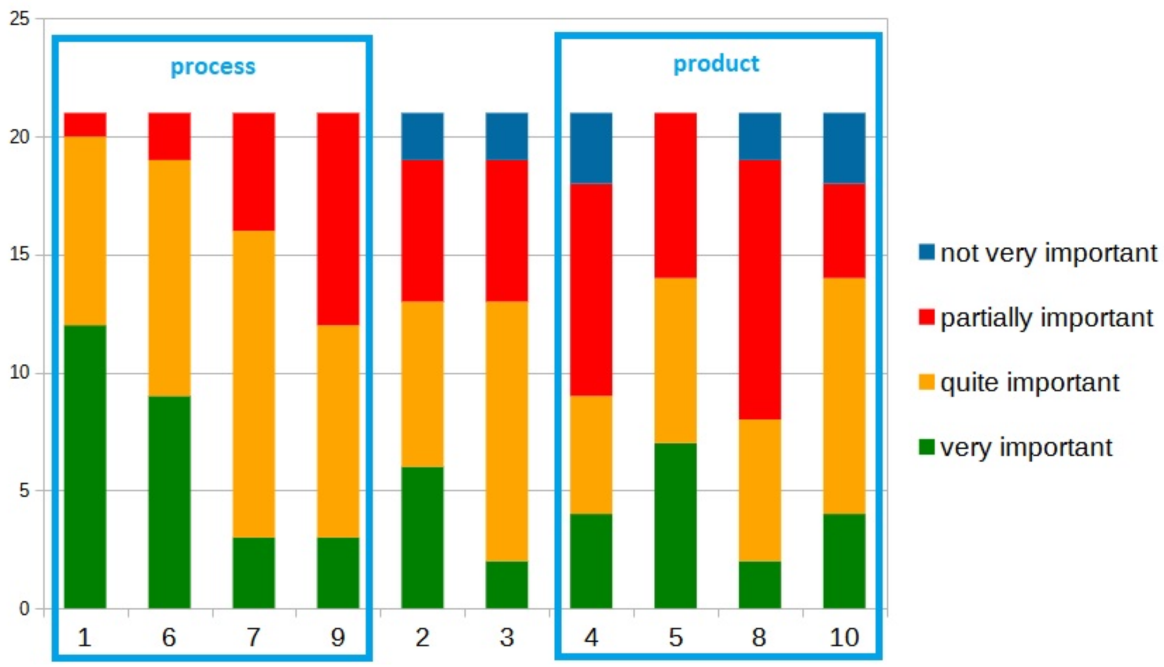


Figure 3.14: Stimuli, distinguishing dimensions, to encourage students' motivation.

- *Feedback from the teacher during the intermediate steps of a program design*;
- *Analysis and solution of concrete problems faced in the labor market*;
- *Collaboration in projects with companies.*

Instead, with respect to the product objectives, the following stimuli were considered more important than the others:

- *Participation in competitions and games*;
- *Opportunity to illustrate the results of their work in the classroom through a multimedia presentation.*

Overall, the teachers seem to prefer to evaluate stimuli inherent to process rather than product aspects.

3.4.5 Assessment

Concerning evaluation, on the basis of concrete experience, we asked the teachers how important they considered some methods of evaluating the learning of programming by students. The survey results are shown in Figure 3.15.

The list of methods to evaluate the learning in programming is as follows:

1. Ongoing observation of the design and technical choices by the students

2. Request to express orally (think-aloud) the reasoning made by students in setting up their own solutions
3. Request to explain the level of confidence in the various steps of the solutions adopted
4. Writing of a diary by students in which to note steps, decisions and choices in relation to the development of a program
5. Realization by the students of a presentation (slide) of the developed program
6. Drafting by the students of a technical document explaining how the program can be used
7. Verification of the functionality of the program created using a test battery not known to the students
8. Testing the program through peer use

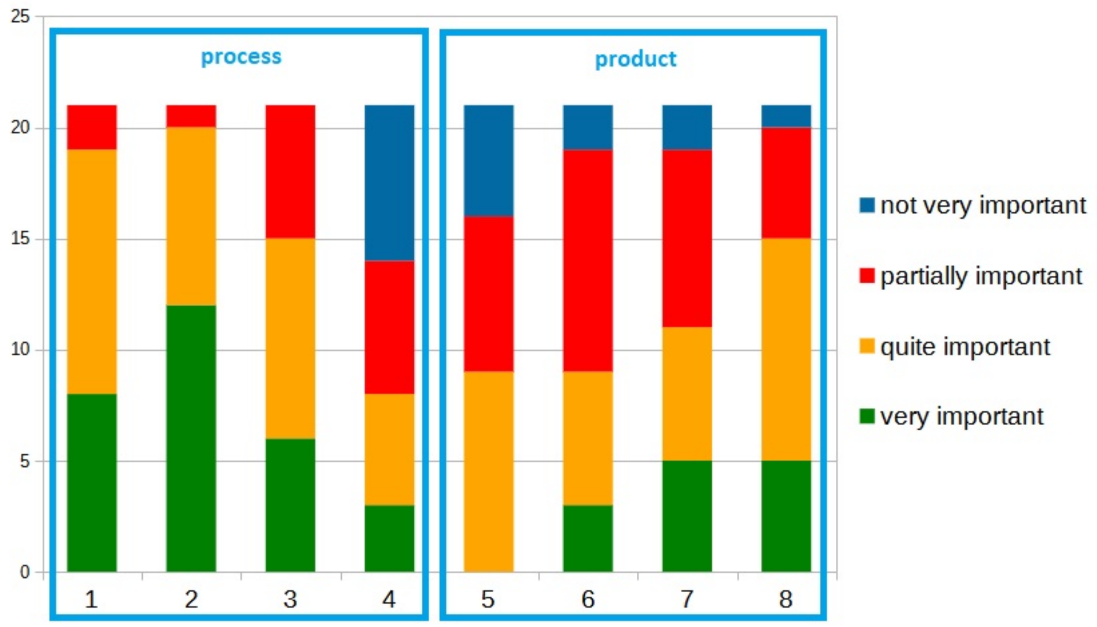


Figure 3.15: Important ways of evaluating the learning of programming, distinguished between process-based assessment and product-based assessment.

The following methods are considered important:

- *Ongoing observation of the design and technical choices by the students,*
- *Request to express orally (think-aloud) the reasoning made by students in setting up their own solutions,*
- *Request to explain the level of confidence in the various steps of the solutions adopted,*
- *Testing the program through peer use.*

On the other hand, the following methods are considered only partially important:

- *Drafting by the students of a technical document explaining how the program can be used,*
- *Verification of the functionality of the program created using a test battery not known to the students.*

Finally, teachers consider the following methods less important:

- *Writing of a diary by students in which to note steps, decisions and choices in relation to the development of a program,*
- *Realization by the students of a presentation (slide) of the developed program.*

Again, teachers place more emphasis on process-based assessment than on product-based assessment.

Teachers frequently encounter, especially in introductory courses, students who propose working solutions that are convoluted, that is significantly more complex than necessary. The teachers were asked how these functioning, but rather complex solutions, are evaluated (see Figure 3.16 internal chart). Figure 3.16 (external chart) shows the results of the assessment on working but computationally inefficient solutions.

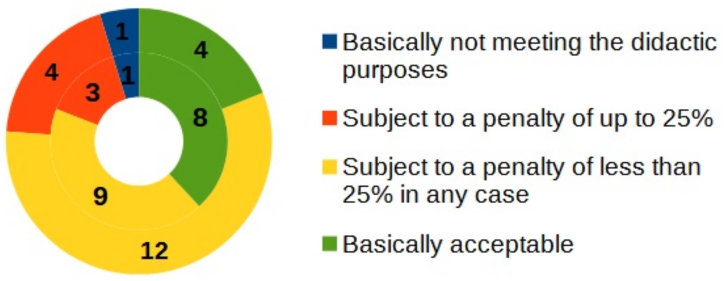


Figure 3.16: Evaluation of the solutions proposed by the students, complex solutions (internal chart) vs inefficient solutions (external chart).

In both cases it turns out that the teachers consider the solutions acceptable, or they apply a small penality (25% or less penality). It is interesting to note that most of the teachers tend to penalize more the computationally inefficient solutions rather than the solutions that are too complex or convoluted.

Focusing the inquiry into the teachers' practice, teachers were asked how they usually evaluate the learning of iteration.

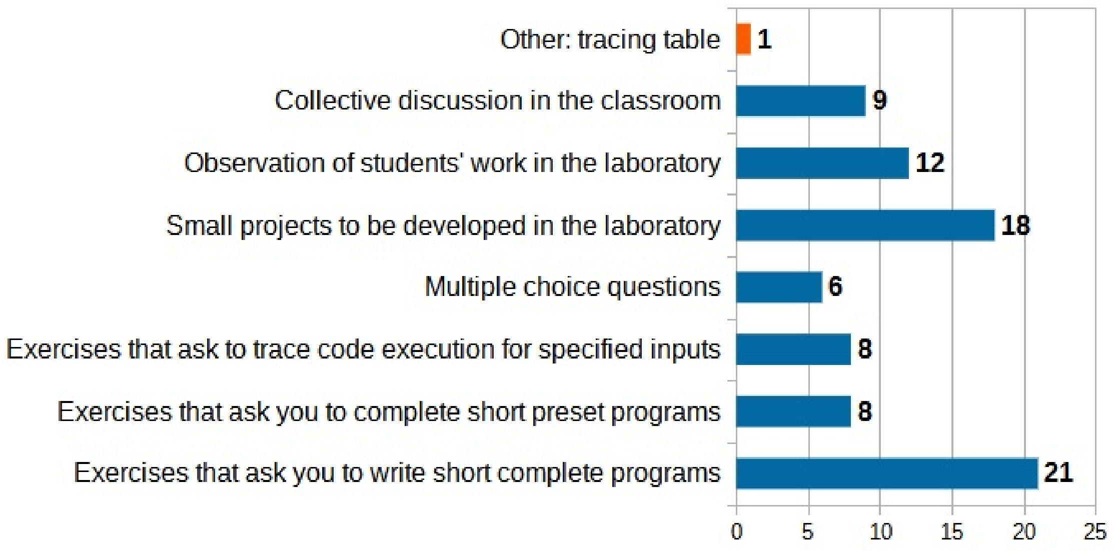


Figure 3.17: Assessment of the learning of iteration. More options were possible.

Figure 3.17 clearly shows that all teachers use *"Exercises that ask you to write short complete programs"* to evaluate their students' learning concerning iterations. Another widely used method is to propose to students *"Small projects to be developed in the laboratory"*. Surprisingly, only one teacher declared they use *"Tracing tables"* to evaluate students for iterations.

3.4.6 Students' aptitudes

Learning the basic principles of Computer Science is not easy for all students, and learning is often conditioned by the specific cognitive and learning styles of each individual student [Cor+18]. Teachers were asked to indicate which aptitudes they consider most important for learning Computer Science. Figure 3.18 shows the degree of importance assigned to a series of aptitudes.

The following are the most important students' aptitudes, as indicated by the teachers: *Spontaneous passion for programming*, *Interest in problem solving*, *Consistency and organization in self-study.*

On the other hand, less important aptitudes indicated are the following: *Precision and formal rigor*, *Imagination and creativity.*

The least important aptitude specified turned out to be the *Ability to relate to peers.*

3.4.7 Other suggestions

From the responses some additional observation emerged, which could be relevant in relation to teaching iteration.

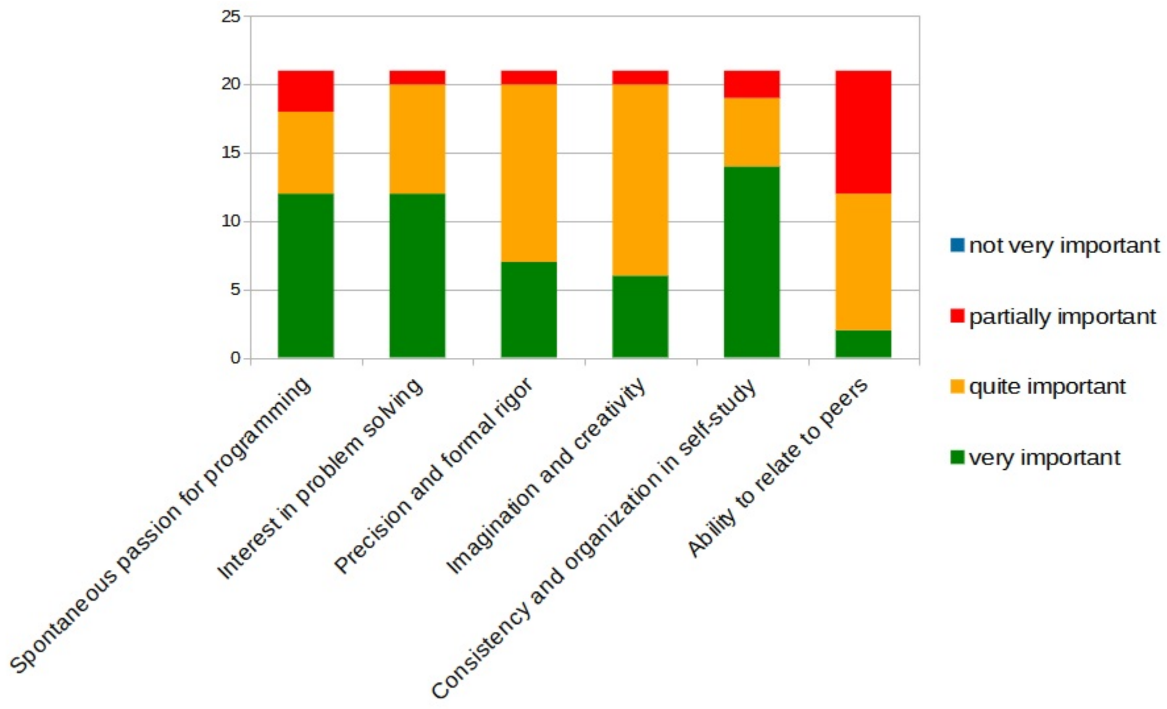


Figure 3.18: Importance assigned to a series of aptitudes for learning Computer Science.

A teacher stated that it is better *"to start from very simple programs that use the For-loop, increase the complexity little by little, it is important not to make the student lose touch with what the program is doing"*.

Another teacher claimed that *"to understand the reasoning, you don't have to write lines of codes straight away, but pupils must learn to formalize with pen and paper"*.

3.5 Discussion

The concept of iteration is a topic that can be addressed in upper secondary school introductory courses. Unlike other topics, where students encounter difficulties, loops are, together with recursion, among the most critical topics in these courses. The teachers indicated that conditionals (If statement) are a relatively simple topic for the students, while in their study of iterations the students encounter more difficulties (see Figure 3.5). In fact, teachers consider *Control structures (loop)* as presenting a higher degree of difficulty for their students. In addition, teachers indicate as difficult topics for students the following: pointers and dynamic memory allocation, recursion, any topic related to OOP (Object Oriented Programming), which should not be ideally explored in the introductory courses, as they are in fact topics suitable for advanced courses.

Some iteration constructs such as While-loop and Do-While-loop, having to manage the conditions, need some previous knowledge. In fact, from the study it emerges that knowledge and skills linked to Boolean algebra and concepts of Mathematical Logic are indicated by teachers as very important (see Figure 3.6). Other important prerequisites also emerged: text comprehension, precision/formal rigor, and general problem solving skills. These prerequisites, which do not strictly pertain to the discipline of Computer Science, will be dealt with in greater detail later in the discussion, as they are related to cognitive and learning aspects.

During the interviews carried out with a selected sample of teachers it emerged that mathematical prerequisites — in many aspects of Mathematics — and Logic are considered to be very important (see Chapter 2, Section 2.1); this result is obvious, in the light of important studies on the link between Mathematical Logic and Computer Science [Mye90; Dav95; WD97; BA01; Qui14; Bla16; Pie17].

An unexpected prerequisite was that of "text comprehension". Evidently more and more students are struggling to understand the texts of the problems that are proposed to them, or they do not understand formalism and coding.

Concerning Reading and Comprehension, Stein and Glenn [SG79] elaborated their "story grammar rules", where characters, places and time constitute the scenario or background in which the story unfolds. Knowing the typical structure of a certain type of text provides a reference framework that facilitates both the production and understanding of a text through top-down processing, creating expectations and facilitating inferences. The schematism involved in the understanding of stories can be related to other forms of use of schemes or rules. An example in this regard is that of the "five Ws", proposed by Anglo-Saxon journalists [Con94; Har96]:

- "Who?" = Who are the characters?
- "Where?" = where do the facts take place?
- "When?" = when?
- "What?" = what happened?
- "Why?" = why did it happen? (the cause and/or purpose).

When the passages faced by the pupils become more complex, it is not always easy to bring them back to a common structure. The identification of characters, places, times, and facts, however, remains an important prerequisite for carrying out further and more complex processes in understanding a text.

The thorough understanding of a text, in particular of a narrative, expository and scientific type, requires the reader to be able to identify the sequence of facts and their different types.

The text of a task or algorithm described in Computer Science formalism therefore requires greater comprehension skills than a narrative text. Not only are the standard characteristics of a narrative text difficult to identify, but the student is also faced with a new and unfamiliar formalism.

The comprehension that allows one to understand the content and meaning of a given text, also allows them to answer questions that can be asked about the content and meaning of what is read [CCDB07].

The process of comprehension is therefore not taken for granted, and it can be compared to a problem solving task in which the reader actively and strategically constructs the meaning of the text through the interaction between their previous knowledge and the information provided by the text [KD78; VDK83].

Understanding is therefore seen as a complex and articulated process. The different models that account for the understanding process substantially agree in describing it as a *construction* or *representation* mental activity of the reader who knows how to interact appropriately even with complex texts, and the more experienced the reader is, the easier the task becomes [Ger91]. Therefore, the process of understanding develops at various levels, and the construction of mental structures and their representation have an important value [DBP93].

Some studies [JL83] show that understanding depends not only on what is read but also on the information already possessed by the reader. Furthermore, Bransford and Johnson [BJ72] pointed out *"that, in the absence of the interaction between information present in the text and previous knowledge, the reader, despite being able to understand the meaning of the text on a superficial level, fails to grasp its global and profound meaning since he cannot construct what is technically defined as a coherent mental model or situational model"*.

Understanding, also based on notions already acquired, and therefore memorized by the reader, also bases its effectiveness on memory, in particular on *long-term memory* regarding information previously learned. Moreover, it exploits the *memory of work* [BH74] for the *"ability to maintain and simultaneously process the content of the text"*. In [DC80] it has been confirmed that the capacity of working memory is fundamental in understanding the text.

Therefore, understanding is based on what has already been learned, on the ability to develop inferences [3], the ability to make associations between what has been read, between parts of the text and what is already known, using working memory and long-term memory [Oak84; FM93; CO99].

In an introductory Computer Science course, students deal with topics of which they have yet to acquire the appropriate knowledge (memory activation) and are unable to establish the necessary inferences, finding themselves working with languages and formalisms different from those they were accustomed to. In CSE several models have been proposed to analyze program comprehension in terms of types of information implied [Pen87], mental representations [WR99], cognitive demand [DSL18],

[3]The ability to draw **inferences** is the ability to deduce information not explicit in the text, or to make connections between information within the text with the knowledge previously acquired by the reader.

or as a hypothesis-driven process [VVMS99]. The teacher can help the student by referring to previously acquired knowledge by the pupil, creating conceptual links.

3.5.1 Focus on iteration

Figure 3.7, and subsequent example lists, shows the tasks that teachers usually propose to their students to explain the iterations. Most of these problems, which they usually see in connection with the iteration, are quite simple and could induce stereotypes, as emerged in Chapter 2, Section 2.3. Furthermore, the proposed examples can be easily re-proposed using a For-loop. Most of the examples cited to explain the While-loop could be solved using a For-loop, avoiding the student the hassle of identifying the termination condition of the iteration. The tasks proposed to explain the While-loop partly overlap with those proposed to present the For-loop. This can in fact induce the student to prefer solution proposals based on For-loop, which however could generate misunderstandings and the acquisition of stereotyped forms in strategies to solve the problems posed.

The same thing is repeated when looking at the examples that are used in the explanations in class. In fact, several examples can be produced with the nested For-loop, such as: manipulating matrices or tables, sorting an array, drawing flat figures, manipulating strings. This is not surprising, and in fact the teachers clearly state that, regarding students' mastery of control constructs, students are more proficient with For-loop than While-loop (see Figure 3.10).

As far as the teaching strategies of the iterations are concerned, the pre-eminence of those directed towards practical objectives over those directed towards conceptual objectives clearly emerges. Among the strategies with conceptual objectives, the only one considered very important is correlated to the "analysis of the termination of an iterative program". The other strategies considered important are instead attributable to practical objectives, such as tracing, debugging, verifying functionality. This result could be due to the fact that the majority of teachers involved in the survey taught in technical upper secondary schools (see Figure 3.3a), where the practical and coding aspects may be prevalent.

3.5.2 Pedagogical approach

When analyzing the pedagogical aspects, such as strategies to improve and consolidate the learning of programming, it is clear that teachers favor strategies linked to practical objectives rather than conceptual objectives. Collaboration between peers in working groups is in fact considered important, and this confirms a practice considered positive for learning both by multidisciplinary studies (e.g. [Hat12]) and by specific works in Computer Science (e.g. [HLR11]). On the other hand, distinguishing between strategies aimed at process-based assessment compared to product-based ones, the data collected show a clear pre-eminence of the first category over the second. This data is interesting, as teachers, while favoring practical objectives, still tend to propose strategies where they value more the process-based assessment.

Regarding stimuli that teachers adopt to encourage and motivate students, it emerges that a process-based approach is considered more important. In particular, the stimulus "Feedback from the teacher during the intermediate steps of a program design" is considered very important, and this result reflects the importance of feedback in Hattie's work [Hat12]. In fact, the feedback aims to reduce the distance between the point where the student is and the point where they should arrive in the learning [Sad89; Sad10]. It can also provide the student with information on concepts that have been misunderstood, and can motivate the student to invest more effort in the assigned task [HT07].

Considering the work of Rahimi et al. [RBH16, p. 75], we can therefore conclude that teachers favor a practical-process-based approach, in fact they argued that "addressing more practical objectives and understanding students' development of soft and design skills through mainly process-based assessment approaches".

Denning & Tedre [DT21] point out that "[t]he preponderance of public discourse on CT[4] is not the abstract algorithm but the executable computer program". Also they argue that there are methods to teach Computer Science without a computer, concerning a more abstract perspective. On the one hand, coding is important in an introductory CS course, but agreeing with Denning and Tedre [DT19], who argued that "we hope that all teachers of computing bring their students a good sense of the richness and beauty of the many dimensions of computation", reasoning and process-based strategies could improve students' comprehension and performance.

3.5.3 Assessment

Concerning evaluation, it clearly emerges that teachers give priority to process-based assessment rather than a product-based assessment. Teachers bestow importance to observing students' solution techniques, as well as understanding the reasoning behind the proposed solutions, but also requesting an evaluation directly from the student on the solutions that they have adopted.

The only product-based assessment that has some importance turns out to be "testing the program through peer use", confirming the usefulness of the comparison between peers.

Furthermore, teachers tend to evaluate with more indulgence, and apply fewer penalties to ineffective solutions rather than unnecessarily complex and convoluted solutions. This choice can be justified by what emerged above, namely that teachers prefer practical objectives and process-based strategies. Moreover, to evaluate the students, teachers prefer to propose exercises in which they require writing short complete programs, or small projects to be developed in the laboratory. It is interesting to note that the practice of tracing is used by a certain number of teachers, and

[4]CT: Computational Thinking.

there are in fact works that confirm its importance as a methodology ,e.g. [Lis+04; VS07; Lop+08]. After all, in technical upper secondary schools the practice in laboratory is relevant, and this probably induces to favor practical objectives.

The practice of having collective discussions in the classroom is also considered quite important, Hazzan et al. [HLR11] consider the class discussion a valid assessment method. On the other hand, no other practices that were considered important emerged, such as peer assessment or individual feedback [MSABA10]. The practice of individual feedback can encourage the student to reflect on their work, stimulating the student's self-confidence [MS68; BFL99; Zim00; SPV20].

3.5.4 Implications for instructors

In Chapter 2, dealing with the findings of a series of interviews with Computer Science teachers, some critical factors had been identified, and the main points raised are as follows:

- The application of mathematical and logical concepts is very important in Informatics, with regard to iterations in particular, having to manage the conditions. However Informatics and Mathematics are not well integrated in the standard high-school curricula, recently subjected to reform.
- Some teachers had considering whether the learning of programming could be improved by starting from the beginning with the object-oriented paradigm, but it seems that this topic is too difficult to be taught in the introductory courses, so it is better dealt with later.

Here, in the light of the analysis made on the data linked by the teachers' survey, further critical factors can be identified, which are listed below.

- Text comprehension by the students of the last generation is common to all disciplines. This is probably above all a job that the literature teacher must carry out. However, it would be interesting to activate interdisciplinary paths, which involve students together with teachers of Computer Science and Literature, to analyze literary texts that present computational concepts. This type of work has already been explored in Mathematics (e.g. [Odi00; SB07; ZR08; LP15; Bis15; CPC13]), while few attempts have been made for Computer Science (e.g. [Mea91; CADM05; Dou04]).
- The understanding of text related to Informatics, even if not proposed through coding, could be more familiar to students if some concepts of computational thinking were presented before high school [Nar20, Chapter I].
- In teaching the iterations it would be desirable to propose more examples and exercises based on the While-loop or Do-While-loop, in order to force students to identify the appropriate conditions, based on the knowledge and skills acquired in Mathematics, and more specifically in Boolean algebra and mathematical logic. This would

discourage students to create concepts and stereotypes based on the For-loop patterns, as De Raadt[DR08, p. 107] suggested when argued that it is best to *"[r]efer to programming strategies rather than underlying syntax where possible. For instance, one could say 'use a for loop to achieve that' when a more strategic instruction would be 'use a counter-controlled loop to achieve that'."*

- As students reported having issues with nested iterations, teachers could enrich the catalog of examples proposed by introducing more tasks, not only with nested For-loop, but also more with nested While-loop and Do-While-loop, as well as with nesting other control-flow constructs.
- Reiterate to the students both the importance of analyzing the problems posed, even with paper and pencil, as well as the importance of verifying the proposed solutions through tracing.
- Give feedback as frequently as possible, so that the student understands their mistakes and keeps higher levels of motivation. Strategies to activate peer assessment could be attempted.
- Conceptual objectives are more easily achievable in tertiary education. However, high school teachers could still try paths pertaining to perspective too, even if only limited to some topics, to evaluate CS concepts rather than developing code, shifting their teaching strategies towards more process-based objectives.

3.6 Concluding remarks

In this chapter, we presented a teachers' online survey protocol aiming to investigate teaching methodologies and strategies regarding iteration, but also their assessment criteria. The results of the survey substantially confirm what emerged from the teachers' pilot interviews. Furthermore, the examples and tasks that teachers usually propose regarding iteration have been cataloged, in particular tasks in connection with *While*-loop, *For*-loop and nested iterations. The most popular examples presented in class are those related to elementary Mathematics, but they are also quite straightforward and tend to induce the use of stereotypical patterns. Hence teachers could enrich the catalog of examples proposed by introducing more tasks, and also induce students to develop their abstraction skills offering them more challenging assignments.

As far as tasks are concerned, it has been useful to design and develop a catalog (Appendix B) of significant examples presenting more varied and interesting structures. We suggest building a catalog by collecting program comprehension (code reading) tasks which require to trace, explain, or evaluate small programs. Each task could cover multiple topics (exit condition, complex condition, nested iterations, nested flow-control constructs, numeric and not numeric data, strings and arrays). A catalog can inspire teachers to introduce more varied and significant sets of tasks in connection with iteration. In addition, a catalog could help students work with non trivial loop conditions and neat combinations of flow-control constructs.

The tasklets discussed in the next Chapter 4 could be a basis to be extended in collaboration with teachers.

Moreover, outcomes highlight that teachers focus orientation towards more practical objectives rather then conceptual objectives. In addition, teachers prefer assess students' understanding and performance through process-based assessment rather then product-based assessment.

Finally, the teachers confirm that the application of mathematical and logical concepts is very important in Computer Science, especially with regard to iterations in particular.

References

[BA01] Mordechai Ben-Ari. *Mathematical logic for computer science.* eng. 2nd ed. London: Springer, 2001.

[Bar+14] Erik Barendsen et al. "Eliciting computer science teachers' PCK using the Content Representation format: Experiences and future directions". In: *Proceedings of the 6th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP'14) – Teaching and Learning Perspectives.* Ed. by Yasemin Gülbahar, Erinç Karataş, and Müge Adnan. Vol. 8730. Istanbul, Turkey: Ankara University Press, Sept. 2014, pp. 71–82.

[BFL99] Albert Bandura, WH Freeman, and Richard Lightsey. *Self-efficacy: The exercise of control.* 1999.

[BH74] Alan D. Baddeley and Graham Hitch. "Working memory". In: *Psychology of learning and motivation.* Vol. 8. Elsevier, 1974, pp. 47–89.

[Bis15] Gian Italo Bischi. *Matematica e Letteratura. Dalla Divina Commedia al Noir.* EGEA, Milano, 2015, p. 160.

[BJ72] John D. Bransford and Marcia K. Johnson. "Contextual prerequisites for understanding: Some investigations of comprehension and recall". In: *Journal of Verbal Learning and Verbal Behavior* 11.6 (1972), pp. 717 –726. DOI: `https://doi.org/10.1016/S0022-5371(72)80006-9`.

[Bla16] Andreas Blass. "Symbioses between mathematical logic and computer science". In: *Annals of Pure and Applied Logic* 167.10 (2016). Logic Colloquium 2012, pp. 868 –878. DOI: `https://doi.org/10.1016/j.apal.2014.04.018`.

[BS13] Teresa Busjahn and Carsten Schulte. “The Use of Code Reading in Teaching Programming”. In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. Koli Calling ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 3–11. DOI: 10.1145/2526968.2526969.

[BSS13] Malte Buchholz, Mara Saeli, and Carsten Schulte. “PCK and reflection in computer science teacher education”. In: *ACM International Conference Proceeding Series* (Nov. 2013). DOI: 10.1145/2532748.2532752.

[CADM05] Romeo Crapiz, Franca Alborini, and Mirka De Marchi. “Letteratura e Informatica Un’esperinza didattica al Liceo Copernico di Udine”. it. In: *Didamatica 2005: Didattica Informatica*. Potenza, Italy: AICA, Oct. 2005, pp. 994–1002.

[CCDB07] Barbara Carretti, Cesare Cornoldi, and Rossana De Beni. *Il disturbo di comprensione del testo*. 2007.

[CO99] Kate Cain and Jane V. Oakhill. “Inference making ability and its relation to comprehension failure in young children”. In: *Reading and Writing* 11 (1999), pp. 489–503.

[Con94] Bradford R. Connatser. “Setting the Context for Understanding”. In: *Technical Communication* 41.2 (1994), pp. 287–291.

[Cor+18] Cesare Cornoldi et al. *Processi cognitivi, motivazione e apprendimento*. Bologna: il Mulino, 2018.

[CPC13] Donna Christy, Christine Payson, and Patricia Carnevale. “The Bridge to Mathematics and Literature”. In: *Mathematics Teaching in the Middle School* 18.9 (2013), pp. 572–577.

[Dav95] Martin Davis. “The Universal Turing Machine (2Nd Ed.)” In: ed. by Rolf Herken. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1995. Chap. Influences of Mathematical Logic on Computer Science, pp. 289–299.

[DBP93] Rossana De Beni and Francesca Pazzaglia. *Lettura e metacognizione. Attività didattiche per la comprensione del testo*. Guide per l’educazione speciale. Centro Studi Erickson, 1993.

[DC80] Meredyth Daneman and Patricia A Carpenter. “Individual differences in working memory and reading”. In: *Journal of verbal learning and verbal behavior* 19.4 (1980), pp. 450–466.

[Dou04] Mark Dougherty. “What Has Literature to Offer Computer Science?” In: *Humanit* 7.1 (2004), pp. 74–91.

[DR08] Michael De Raadt. “Teaching programming strategies explicitly to novice programmers”. PhD thesis. University of Southern Queensland, 2008.

[DSL18] Rodrigo Duran, Juha Sorva, and Sofia Leite. “Towards an Analysis of Program Complexity From a Cognitive Perspective”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 21–30. DOI: 10.1145/3230977.3230986.

[DT19] Peter J. Denning and Matti Tedre. *Computational thinking*. Mit Press, 2019.

[DT21] Peter J. Denning and Matti Tedre. “Computational Thinking: A disciplinary perspective”. In: *Informatics in Education* 20.3 (2021), pp. 361–390.

[Fin+20] Sally Fincher et al. “Notional Machines in Computing Education: The Education of Attention”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 21–50. DOI: 10.1145/3437800.3439202.

[FM93] Luciana Ferraboschi and Nadia Meini. *Strategie semplici di lettura. Esercizi guida per la comprensione del testo*. Materiali di recupero e sostegno. Centro Studi Erickson, 1993.

[Ful+07] Ursula Fuller et al. “Developing a computer science-specific learning taxonomy”. eng. In: *ACM SIGCSE Bulletin* 39.4 (2007), pp. 152–170.

[Ger91] Morton Ann Gernsbacher. “Cognitive processes and mechanisms in language comprehension : the structure building framework”. In: *Psychology of Learning and Motivation* 27 (1991), pp. 217–263.

[Har96] Geoff Hart. “The Five W’s: An Old Tool for the New Task of Audience Analysis.” In: *Technical Communication: Journal of the Society for Technical Communication* 43 (1996).

[Hat12] John Hattie. *Visible learning for teachers: Maximizing impact on learning*. Routledge, 2012.

[Her10] Matthew Hertz. “What Do "CS1" and "CS2" Mean? Investigating Differences in the Early Courses”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 199–203. DOI: 10.1145/1734263.1734335.

[HLR11] Orit Hazzan, Tami Lapidot, and Noa Ragonis. *Guide to Teaching Computer Science: An Activity-Based Approach*. 1st. Springer Publishing Company, Incorporated, 2011.

[HT07] John Hattie and Helen Timperley. “The power of feedback”. In: *Review of educational research* 77.1 (2007), pp. 81–112.

[JL83] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness*. 6. Harvard University Press, 1983.

[KD78] Walter Kintsch and Teun A. van Dijk. “Toward a model of text comprehension and production.” en. In: *Psychological Review* 85.5 (1978), pp. 363–394. DOI: `10.1037/0033-295X.85.5.363`.

[Lis+04] Raymond Lister et al. “A Multi-national Study of Reading and Tracing Skills in Novice Programmers”. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’04. New York, NY, USA: ACM, 2004, pp. 119–150. DOI: `10.1145/1044550.1041673`.

[Lop+08] Mike Lopez et al. “Relationships Between Reading, Tracing and Writing Skills in Introductory Programming”. In: *Proc. 4th Int. Workshop on Comput. Educ. Research*. ICER ’08. New York, USA: ACM, 2008, pp. 101–112.

[LP15] Sally I. Lipsey and Bernard S. Pasternack. “Mathematics in Literature”. In: (2015).

[Mea91] H. Willis Means. “Using Literature in a Computer Science Service Course: Improving Abstract/Critical Thinking Skills”. In: *J. Comput. Sci. Coll.* 6.5 (Apr. 1991), pp. 30–34.

[MS68] H. Edward Massengill and Emir H. Shuford. *The effect of ’Degree of Confidence’ in student testing*. eng. Tech. rep. 1968.

[MSABA10] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. “Learning Computer Science Concepts with Scratch”. In: *Proceedings of the Sixth International Workshop on Computing Education Research*. ICER ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 69–76. DOI: `10.1145/1839594.1839607`.

[Mye90] J.Paul Myers. “The Central Role of Mathematical Logic in Computer Science”. In: *ACM SIGCSE Bulletin* 22.1 (1990), pp. 22–26.

[Nar20] Enrico Nardelli. *Coding e oltre. L’informatica nella scuola*. Liscianilibri, 2020.

[Nut07] Graham Nuthall. *The Hidden Lives of Learners*. NZCER Press, 2007.

[Oak84] Jane V. Oakhill. “Inferential And Memory Skills In Children’s Comprehension Of Stories”. In: *British Journal of Educational Psychology* 54 (1984), pp. 31–39.

[Odi00] Piergiorgio Odifreddi. “Metodi Matematici Della Letteratura”. In: *Nuova Civiltà Delle Macchine* 18.3 (2000), pp. 116–133.

[Pen87] Nancy Pennington. “Comprehension Strategies in Programming”. In: *Empirical Studies of Programmers: Second Workshop.* USA: Ablex Publishing Corp., 1987, pp. 100–113.

[Pie17] Aleksander Piecuch. “Zaniedbana algebra a nauczanie informatyki”. eng. In: *Edukacja - Technika - Informatyka* VIII.3 (2017), pp. 288–295.

[Qui14] Margareth Quindeless. “Logic in the curricula of Computer Science”. spa. In: *Revista AntioqueÃ+a de las Ciencias Computacionales y la IngenierÃ-a de Software (RACCIS)* 4.2 (2014), pp. 47–51.

[RBH16] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. “Typifying informatics teachers’ PCK of designing digital artefacts in Dutch upper secondary education”. In: *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives.* Springer. 2016, pp. 65–77.

[Sad10] D. Royce Sadler. “Beyond feedback: Developing student capability in complex appraisal”. In: *Assessment & evaluation in higher education* 35.5 (2010), pp. 535–550.

[Sad89] D. Royce Sadler. “Formative assessment and the design of instructional systems”. In: *Instructional science* 18.2 (1989), pp. 119–144.

[Sae+11] Mara Saeli et al. “Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective”. In: *Informatics in Education* 10 (Apr. 2011), pp. 73–88.

[SB07] Bharath Sriraman and Astrid Beckmann. *Mathematics and Literature: Perspectives for interdisciplinary classroom pedagogy.* Jan. 2007.

[SBGM02] Dominique MA Sluijsmans, Saskia Brand-Gruwel, and Jeroen JG van Merriënboer. “Peer assessment training in teacher education: Effects on performance and perceptions”. In: *Assessment & Evaluation in Higher Education* 27.5 (2002), pp. 443–454.

[SG79] Nancy Stein and Christine Glenn. “An Analysis of Story Comprehension in Elementary School Children”. In: *New Directions in Discourse Processing* 2 (Jan. 1979).

[SPV20] Phil Steinhorst, Andrew Petersen, and Jan Vahrenhold. “Revisiting Self-Efficacy in Introductory Programming”. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research.* 2020, pp. 158–169.

[VDB12] Jan H. Van Driel and Amanda Berry. “Teacher Professional Development Focusing on Pedagogical Content Knowledge”. eng. In: *Educational Researcher* 41.1 (2012), pp. 26–28.

[VDK83] Teun Adrianus Van Dijk and Walter Kintsch. *Strategies of discourse comprehension.* 1983.

[VS07] Vesa Vainio and Jorma Sajaniemi. “Factors in novice programmers’ poor tracing skills”. In: *SIGCSE Bull.* 39.3 (2007), pp. 236–240. DOI: `10.1145/1269900.1268853`.

[VVMS99] A. Marie Vans, Anneliese Von Mayrhauser, and Gabriel Somlo. “Program understanding behavior during corrective maintenance of large-scale software”. In: *International Journal of Human-Computer Studies* 51.1 (1999), pp. 31–70. DOI: `https://doi.org/10.1006/ijhc.1999.0268`.

[WD97] Roland Wagner-Dobler. “Science-Technology Coupling: The Case of Mathematical Logic and Computer Science.” eng. In: *Journal of the American Society for Information Science* 48.2 (1997), pp. 171–83.

[WR99] Susan Wiedenbeck and Vennila Ramalingam. “Novice Comprehension of Small Programs Written in the Procedural and Object-Oriented Styles”. In: *Int. J. Hum.-Comput. Stud.* 51.1 (July 1999), pp. 71–87. DOI: `10.1006/ijhc.1999.0269`.

[Zim00] Barry J. Zimmerman. “Self-efficacy: An essential motive to learn”. In: *Contemporary educational psychology* 25.1 (2000), pp. 82–91.

[ZR08] Christopher Zaleta and Kim Ruebel. “Exploring Mathematical Concepts in Literature”. In: *Middle School Journal* 40 (2008), pp. 36–42.

Chapter 4

Student Survey

In order to test the provisional hypotheses suggested by the pilot studies discussed in Chapter 2, as well as to address other related issues, we designed and developed an instrument to investigate in more depth high-school students' understanding of iteration. After setting the research questions in Section 4.1, we outline the methodology in Section 4.2. Then, Section 4.3 presents the second survey for students, covering a set of tasklets. In particular, we will summarize the motivations underlying the choice of the tasks and the overall structure of the instrument. Moreover, in Section 4.5, we will analyze the data collected from a sample of 225 high school students. Finally, based on the obtained results, in Section 4.6 we attempt to formulate some suggestions for teachers and in Section 4.7 we end with a few concluding remarks.

4.1 Aims and scope of the student survey

Our current endeavor is based on two previous works: (i) a students' survey that included three small tasks involving basic looping constructs, as well as two questions on their subjective perception of difficulty (Chapter 2), and (ii) a teachers' survey that aimed to ascertain their beliefs about major sources of issues for basic programming concepts and their approach to the teaching and learning of iteration constructs (Chapter 3).

Based on students' performance in the three small programming tasks included in the pilot survey, discussed in Section 2.2, it is evident that several of them do not master loop conditions and nested constructs. Nested loops and complex loop conditions are also challenging in students' subjective perception.

Students' pilot survey outcomes suggests that, on the one hand, most students seem to have developed a viable mental model of the basic workings of the underlying machine. On the other hand, dealing at a more abstract level with loop conditions and nested flow-control structures appears to be challenging.

The aim of this instrument is to verify, from various points of view, high school students' mastery of iteration constructs. By mastery we here mean *conceptual* mastery of code structures, focusing on program *comprehension* rather than construction [Izu+19]. The perspective is therefore different from that suggested in [CLM20], not evaluating program-writing skills but instead reading, understanding, tracing and abstraction skills.

To sum up, we expect that the outcomes of the survey should allow us to identify major areas of students' difficulties and whether they match with students' perception of their own self-confidence. More specifically, this part of the study attempts to address the following research questions:

Q1 To what extent are students at ease with a range of "technical" features implied by iteration? (E.g., structure and role of loop conditions, loop-control variables, nesting of flow-control constructs, looping over arrays.)

Q2 Does the effort to trace code facilitate thinking of the overall computation at a higher abstraction level?

Q3 Are students' answers more accurate when using flow-chart or textual code representations of programs?

Q4 To what extent are students self-confident about their comprehension of a program's overall computation and purpose?

4.2 Methodology

In this work, we tried to understand novices' difficulties related to iteration through a survey submitted to Computer Science high school students. Interviewing students or administering a survey to students is becoming a popular way of data-collection in Computer Science education [Haz99].

Measuring students' learning in Computer Science needs designing programming tasks [Gin18; Gro+17]. Differently from the pilot survey, this one is made up of tasks only. This choice allows to verify the real performances of the students, through tasks that consider understanding of iterative programs from multiple points of view. Furthermore, this survey considers students' levels of self-confidence.

Tasks have been designed taking a cue from those proposed in the catalog (see Appendix B) and extend the investigation on students' performance, regarding iterations, considered in the previous survey, analyzed in Chapter 2, where only three tasks were submitted. Concerning high-order thinking skills tasks, the investigation of connections between iteration and abstraction, presented in Chapter 1, was useful in identifying areas to explore.

Mannila et al. [Man+20] claim that "the self-efficacy-questions provided information on how students experience the programming process and what they may find difficult — or uninteresting", as they recall Zimmerman's previous work [Zim00]. For instance, the authors explore primary school students' performance through simple tasks and match it with novices' self-efficacy.

Smetsers Weeda & Smetsers [SWS17] argue that flow-charts supports novice programmers in facilitating planning and understanding. Furthermore, Chetty et al. [CW15] realized that "students perceived the design to be beneficial to their learning", and further that "success rates in the course improved dramatically".

In the design of our investigation instrument we have included both a code only version and a version which also includes a flow-chart, as this choice allows us to compare students' performance and their accuracy.

4.3 Characterization of the instrument

Beyond the suggestions drawn from the literature [Lis+06; TL14c; RBH17; IM20; Man+20; MIS20], our study is based on the insights gained from our two previous studies:

(i) A first survey asking high-school teachers about the role of iteration in their practice and their perception of students' difficulties (Chapter 2, Section 2.1);

(ii) A second survey addressed to students, again asking them about their subjective perception of difficulties, but also including three small tasks involving basic iteration constructs (Chapter 2, Section 2.2).

Our main contribution is to offer some key findings related to the investigation of students' mastery of iteration.

- We plan to deepen students' mastery of nested iterations, sometimes not covered with adequate examples.
- Iterations with complex conditions are rarely presented, and only few tutorials deal with the topic effectively.
- With regard to reading and tracing skills that are usually acquired, we want to analyze these skills and compare them with thinking and abstraction skills.
- Relate reading and tracing skills with reversibility and the ability to master it.
- Studies carried out with students of Mathematics and of Computer Science have confirmed that these students have a spatial cognitive style [HK99; WLB09; Ati+20]. We want to understand if this cognitive style actually has a value in the reading of a flow-chart algorithm with respect to coding.
- How much self-confidence do the students display regarding the iterations and how consistent is it with their actual level of mastery of the subject?

Data from teachers and students had confirmed that iteration is a central concept in the introductory courses and the data allowed to identify the treatment of loop conditions and of nested constructs as major sources of students' difficulties with loop constructs.

In particular, we pointed out three potential insights, which are worth further investigation. By referring to a *competency framework* for computing education [Fre+18], the first two pertain to the *skills* area and the third one to the *dispositions* area:

- Firstly, identify suitable test cases in order to confirm or refute working conjectures regarding students' difficulties on iterations.
- Secondly, investigate treatment of loop conditions, especially in connection with the statement of a problem.
- Finally, at the meta level, examine students' attitude to thinking critically about their learning, asking them to make explicit their degree of self-confidence in the achievement of a task or of a part of it.

In other words, the question arises as to whether a representation with a flow-chart allows the use of the spatial cognitive style of reference to facilitate understanding, or whether understanding is made easier by reading the code. In addition, we ask whether spatial styles act at different levels.

In the previous survey [SM20] — where the focus aimed to program comprehension (code reading), not construction (code writing) — only three small problems were included, referred to as *tasklets*, to address each of the learning dimensions introduced in [Mir12], namely the understanding of the computation model underlying iteration, the ability to establish relations between the components of a loop and the statement of a problem, and the ability to interpret the program structures based on iteration.

Here the objective is to design a survey based on problems and exercises (tasklets), to verify, from various points of view, the high school students' mastery of iteration constructs. By mastery we mean *conceptual* mastery of code structures, focusing on program *comprehension* rather than construction [Izu+19], the same approach used in the students' pilot survey.

We have devised two tasksets, both consisting of 6 tasklets with a balanced distribution over the areas and topics of Table 4.1, arranged in such a way that students can be expected to complete the test in about one hour. The chosen tasklets have been conceived in order to attempt to answer questions Q1–4 listed above. Each tasklet asks one or two multiple-choice questions. In addition, the programs of tasklets may be presented by code or flow-charts. The survey has then been made accessible online, the four resulting versions (2 tasksets × code/flow-chart modes) being assigned randomly. An English translation is available via this link: `http://nid.dimi.uniud.it/additional_material/iteration_survey.html`

In Table 4.1, each area is labeled with the investigation question (tag) it pertains to. Any single tasklet usually involves topics in more than one area. While the technical

Table 4.1: Areas and topics addressed by the tasklets.

A. Tasklets addressing higher-order thinking skills (Q2, Q4)
   1. Abstraction on the computational model
      a. Equivalence (nested constructs, `for`/`while`, `do-while`/`while` ...)
      b. Reversibility
   2. Relationships with the application domain
      a. Completion (of condition, expression, statement ...)
      b. Functional purpose

B. Tasklets addressing code features (Q1, Q3)
   1. Structural features
      a. Plain loop
      b. Nested conditional
      c. Nested loop
   2. Processing plan
      a. Exit condition
      b. Loop control variable
      c. Downward `for` loop

B. (continued)
   3. Conditions
      a. Simple condition
      b. Composite condition
      c. Boolean expression/variable

C. Tasklets addressing code execution, conceivably via tracing (Q1, Q2)
   1. Output/final state
   2. Number of iterations

D. Tasklets addressing data types (Q1)
   1. Numerical data (only)
   2. Non-numerical data
   3. Array data

SC. Perception of self-confidence (Q4)

FC. Flow-chart versus code (Q3)

code features (Q1) are intrinsically related to any given tasklet, the correlation between tracing effort and abstraction (Q2) is addressed by asking two subsequent questions in the same task. On the other hand, the role of flow-charts (Q3) is expected to result from comparing the outcomes for two randomly assigned versions of the same task, where the program is presented as flow-chart vs. code. Finally, the indication of the subjective perception of self-confidence (Q4), in a 4-grade Likert scale, is required for each question concerning high-level thinking skills.

As to the last point (Q4), one of the insights emerging from the analysis in [SM20] — where we argued that students seem to underestimate their difficulties when dealing with loop conditions — suggests we should investigate also students' attitude when thinking critically about their learning, i.e. at the meta level, in particular by asking them to make explicit their degree of self-confidence in the achievement of a task or a part of it.

Most items in Table 4.1 should be self-explanatory, but in the following subsections we will elaborate a little on each of the four reported areas.[1]

A. Tasklets addressing higher-order thinking skills

This area covers two broad categories, concerning abstraction over the computation structure and functional abstraction in connection with some problem domain.

Cetin [CD17] stated that *"the connection between loops and reflective abstraction lies in the genetic decomposition of loops"*, identifying four cases: pre-action conception of loops, action conception of loops, process conception of loops, object conception of loops.

Functional abstraction has also been investigated here through reversible tasks. Reversibility is a property of a program or function that indicates it could be brought back to its original state, but it is a topic that only few scholars have explored, and only recently have papers been presented on the subject. Ginat & Armoni [GA06] consider the central role of "reverse thinking" in Computer Science. Teague & Lister [TL14b] investigated to what degree novice programmers manifest the ability to work with this concept of reversibility. They started from the assumption that Piaget had identified reversibility as an indicator of the ability to reason at a concrete operational level. Their results suggested "that many students remain at the sensorimotor and pre-operational levels because all the instruction they receive is at the concrete operational level". They conclude the analysis arguing that when students work with concepts like reversibility they could reason abstractly.

Izu et al. [IPW17] argued that "reasoning about reversibility requires students to have a mental model of the state, thus they should reason about program behavior as a whole, compared with reasoning about concrete cases using testing and tracing", but students often fail to correctly reason about reversibility. Moreover, Izu et al. [IMW18] suggested that the concept of reversibility could be a useful resource for educators to assess and develop students' understanding of program behavior. Mirolo et al. [MI19; MIS20] analyzed reversibility tasks and exercises under the lens of the SOLO taxonomy [BFC82; Lis+06; Ful+07; MHD09].Their analysis has provided insights into novices' mastery of conditional constructs:

- students do not seem to be careful enough while dealing with border computations;
- the lack of an explicit *Else* branch turns out to affect students' analysis of the code behavior;
- a significant number of students appear to face problems in order to master the correlation between conditions and operations in the reversing code.

To test students' abilities in the former category we use *equivalence* [IM20] and *reversibility* [TL14b; MIS20] tasks, as both kinds of assignments are inspired by recent literature. For instance, a sample equivalence task is shown in Figure 4.1.

[1]A link to a public repository with an English versions of the whole set of tasklets is available via the link `http://nid.dimi.uniud.it/additional_material/iteration_survey.html`.

The role of reversibility in learning, on the other hand, dates back to Piaget's work on cognitive development, where it is considered as an indicator of achievement of the concrete operational stage. So, both equivalence and reversibility tasks require students to reason about program behavior comprehensively, generalizing what could be ascertained by tracing code execution for some specific input data.

```
int x = m;
int y = n;

while ( x != y ) {

  while ( x < y ) {
    x = x + m;
  }
  while ( x > y ) {
    y = y + n;
  }
}
output( "x = " + x );
```

reference program

```
int x = m;
int y = n;

while ( x != y ) {

  while ( x < y ) {
    x = x + x;
  }
  while ( x > y ) {
    y = y + y;
  }
}
output( "x = " + x );
```

option 1

```
int x = m;
int y = n;

while ( x != y ) {

  do {
    x = x + m;
  } while ( x < y );
  do {
    y = y + n;
  } while ( x > y );
}
output( "x = " + x );
```

option 2

```
int x = m;
int y = n;

while ( x != y ) {

  if ( (x < y) || (x > y) ) {
    x = x + m;
    y = y + n;
  }
}
output( "x = " + x );
```

option 3

```
int x = m;
int y = n;

while ( x != y ) {

  if ( x < y ) {
    x = x + m;
  } else {
    y = y + n;
  }
}
output( "x = " + x );
```

option 4

Figure 4.1: Equivalence tasklet: *Which option is equivalent to the reference program?*

The tasklets in the second broad category, addressing functional abstraction, include more common types of questions which ask to choose the appropriate condition/expression/statement to complete a program intended to achieve a given purpose. We can see in Figure 4.2 an example of functional purpose tasklet.

```
// input: char[] v  s.t. v.length > 0
int n = v.length;  // lunghezza array v
char x = v[n-1];
for ( int i=n-2; i>=0; i=i-1 ) {
  v[i+1] = v[i];
}
v[0] = x;
output( "v = " + v );
```

Figure 4.2: Functional purpose tasklet: *What could be the purpose of the program?*

Equivalently, it is asked to identify the purpose of a given program as a further variation on the "Explain in Plain English" theme [Lis+06]. The completion tasks are also motivated by the unexpected (to a similar extent) difficulties found in [SM20] to fit the loop condition with the specification of a straightforward problem.

B. Tasklets addressing code features

Much of the structure of area B is based on the data collected in our two pilot surveys, answered by teachers [SM19] and by students [SM20], which suggest that loop conditions and nested constructs can be identified as major sources of difficulties. Nested loops, in particular, appear to be a significant challenge also in the learners' subjective perception. Other researchers have identified widespread issues and misconceptions regarding precisely the nested constructs [Gin04; MBŽ18; Cet+20], and in [Cet15], for instance, a theoretical "action-process-object-schema" framework is suggested to analyze student's cognitive obstacles in this respect.

Nested loops are not explicitly mentioned in Luxton-Reilly et al. [LR+18] review, but there are some works on this topic, while for-loop, while-loop and nested loops are investigated by [TG10]. In his work, Ginat [Gin04] showed that students have difficulties with nested loops. Students from different grade levels show a cognitive difficulty in understanding nested loops [IWP16; MBŽ18]. In this context, Yarmish [YK07] suggested that "when teaching nested loops teachers should focus on student recognition of problems where nested loops should be used".

Cetin [Cet15] explored students' understanding of loops and nested loops concepts. In particular, his results showed that the n-level nested loop requires an understanding of the iteration status; so student should develop mental constructions. In a later work Cetin [Cet+20] identifies two typical misconceptions: simultaneous nested loops misconception and low level understanding of loops concept.

Koppelman and Van Dijk [KD10] stated that "nested while-loops generate a lot of mistakes", that "novices did not usually recognize the need for a nested loop", and proposed some test tasks. Furthermore, Cerny et al. [CHR13] conclude that nested loops have a hierarchical structure and this opens the possibility of using hierarchical segment-based abstractions to analyze them.

Moreover, Grover & Pea [GP18b] consider that Logic and logical thinking can be considered as a case of computational thinking, with conditions and complex conditions, as well as to construct Boolean expressions, are important topics in introductory courses. They also imply knowledge of logical and mathematical prerequisites.

With regard to identifying various patterns related to iterations, several works report patterns mainly with for-loops [Lis11; Kes19; IPW19; Izu+19; EHR20]. Even when patterns related to nested iterations are considered, they usually take nested *for*-loop considerations [Lop+08; Cet15; LDC20]. We believe that the *while*-loop is significant, as it is a more general iterative control construct, which forces the student to identify a condition as well as to master relations and Boolean operators.

A couple of additional topics pertaining to this area concern the understanding of the treatment of loop-control variables, see e.g. [DB86], and the different level of mastery of downward (or down-counting) loops vs. more stereotypical upward (up-counting) loops, pointed out e.g. in [KD03].

C. Tasklets addressing code execution

Small problems that can be solved at low levels of abstraction by tracing the code execution for specific input data are certainly among the most common programming tasks in which novices are required to engage. Ordinarily, tracing is deemed to be a basic ability "to build [...] higher-level comprehension skills upon" [Lis+04], even though it is not a sufficient prerequisite in this respect [TL14a].

Our main purpose, however, is to address the investigation question Q2, namely, whether code tracing can, to some extent, support higher-order thinking *in the task at hand*. In [SM20] we indeed found some cues suggesting that students' performance on more abstract issues implied by the task may improve when they are actually led to engage in some careful tracing, but that they tend to elude this effort to check their conjectures about program behavior. The importance of tracing in novice programmers' learning was already analyzed by Perkins et al. in 1986 [Per+86].

More recently, Lister et al. [Lis+04] argued "that many students have a fragile grasp of both basic programming principles and the ability to systematically carry out routine programming tasks, such as tracing through code", an important requirement for evaluating and interpreting the code. The authors suggested that "it may be appropriate to first teach systematic tracing as a base skill, then allow students to build these higher-level comprehension skills upon that base".

Furthermore, the authors take up what had already been analyzed by Philpott et al. [PRW07], who had indicated that "high tracing scores clearly establishes a link between well developed tracing skills and the ability to think relationally". In fact Lopez et al. [Lop+08] considered both non-iterative and iterative tasks, and found that many students struggle with tracing loops, in particular *while*-loop. They tested the hypothesis that there might be a positive correlation between program writing skill and tracing skill, "particularly when the tracing involved loops".

Venables et al. [VTL09] confirmed Lopez et al.'s results, and furthermore proposed new types of question requiring students to trace iterative code, while the other type required students to explain what a piece of code did. However, they did find that some aspects of their model are sensitive to the particular exam questions used. In other terms they "found that student performance on explaining code was hard to characterize, and the strength of the relationship between explaining and code writing is particularly sensitive to the specific questions asked".

Moreover, Teague & Lister [TL14a] presented evidence that some novice programmers have the ability to hand execute – make a trace – pieces of code and yet are not able to explain what that code does. This result is consistent with neo-Piagetian stage theory of programming.

Thus, the idea is to assign similar tasks pertaining to the "abstract" area A, either including or not a first question that can be answered via tracing. More specifically, such questions ask about the outcome or the overall number of iterations for the given input data.

D. Tasklets addressing data types

The covered data types are essentially *numbers*, *booleans*, *strings*, and *arrays.* There are manifold reasons to include a range of data types. On the one hand, it is desirable that the set of tasklets is not perceived as entailing "just mathematical stuff."

The importance for students to master arrays has been a well-known concept for some time [SS88]. Thompson et al. [Tho+06] argue that in the SOLO Advanced Multistructural (AM) category, that also has an element of a relational understanding, the students recognized that the way the loop is constructed affects the direction in which the array is processed, and the majority of the students could identify valid criteria to classify the code segments. Other works, e.g. Yarmish and Kopec [YK07] investigated students' errors that related to two-dimensions arrays, and Alzahrani et al. [Alz+18] described errors related to vector (one-dimension arrays) index.

On the other, we want to test how students deal with Boolean data and arrays. The indexed access to arrays, in particular, can be problematic for novices, especially in connection with iteration — see e.g. some recent work [RDLR20; MS21].

SC. Perception of self-Confidence

In the design of the survey we deliberately included the students' self-confidence analysis, a type of investigation still used very little in CSE at the moment [SPV20]. Students' self-confidence [MS68; HH82; Ban00] constitutes an attitude about their skills and abilities. It means they accept and trust themselves and have a sense of control over their activities. A low level of self-confidence might make a student feel full of self-doubt. Having high or low self-confidence is rarely related to actual abilities, and mostly based on perceptions. Self-confidence, when measured, offers predictive advantages when a task is familiar [Zim00].

There are various ways in which self-confidence beliefs contribute to the development of cognitive skills, and we want to explore students' beliefs about their self-efficacy in mastering iterations, in various forms.

Each tasklet requires to reason about a given program by asking at least one question in area A, and after answering this question students also have to indicate their perceived level of self-confidence in a Likert scale ranging from 1 (not confident at all) to 4 (fully confident). Apart from the potential pedagogical implications [SPV20], we decided to include this feature because our previous survey showed that the subjective perception of difficulty is not always aligned with the actual performance in a task [SM20].

FC. Flow-chart vs. code

Already in 1983 Ramsey et al. [RAVD83] performed an experiment to assess the relative merits of program design languages (PDLs) and flow-charts as techniques for the development and documentation of detailed designs for computer programs. They figured out that "the use of a PDL by a software designer, for the development and description of a detailed program design, produced better results than did the use of flowcharts". Their results described that "the designs appeared to be of significantly better quality, involving more algorithmic or procedural detail, than those produced using flowcharts", and "flowchart designs exhibited considerably more abbreviation and other space-saving practices than did PDL designs".

However, Smetsers Weeda & Smetsers [SWS17] argued that "flowcharts support novice programmers to keep track of where they are and give guidance to what they need to do next, similar to a road-map". Furthermore "they facilitate planning, understanding and decomposing the problem, communicating ideas in an early stage, step-wise implementation and evaluating and reflecting on the solution (and approach) as a whole".

Moreover, Rahimi et al. [RBH17] described a flowchart-based approach to identifying secondary school students' misconceptions on basic algorithm. Their "results suggest that, given their abstract and language-independent nature, flowcharts can be considered as an effective tool for revealing students' difficulties in understanding algorithmic concepts", so they suggested "the usage of flowcharts as a formative assessment tool" to Informatics teachers.

Cetin [Cet+20] shared the implicit importance of the spatial cognitive style and he argued that "visualization is one of the ways to help students improve their understanding". He figured out that "visualization based instruction helped pre-service teachers improve their understanding of loops concept", we can hypothesize that the same benefit can also be identified for students.

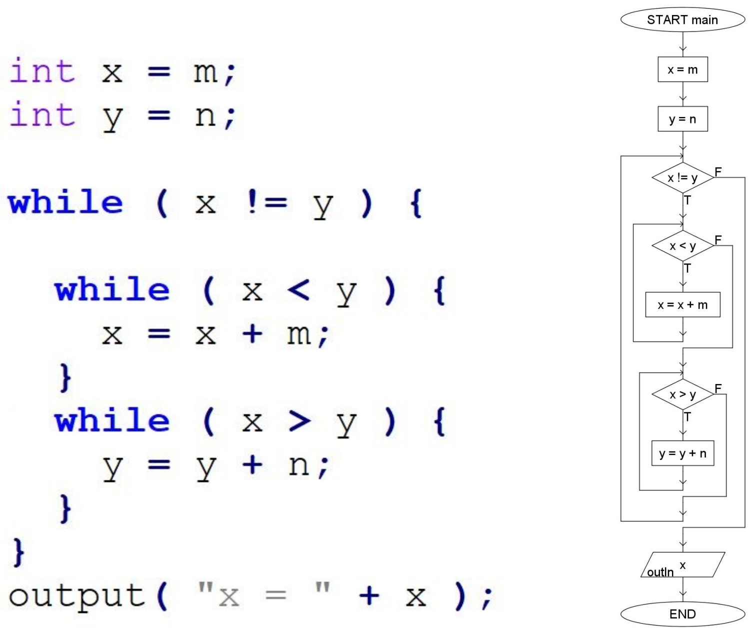
Moreover, Fincher et al. [Fin+20] studied that graphical presentation provided insight into misconceptions about the semantics of for-loops.

For instance, a task in code and flow-chart versions is shown in Figure 4.3.

Putting the pieces together

Overall, we have defined 10 tasklets, labeled T1–T10 in Table 4.2, which show their distribution in terms of areas/topics, as well as relative to the two versions of the survey intended to be assigned to students, either version consisting of 6 tasklets. The first version is connected to "tracing-based" questions, whilst the second one to "more abstract" questions. For the sake of comparison, T3 and T7 are presented in two related variants, distinguished by additional label suffixes a/b. Moreover, the programs of three tasklets may be represented either as code or as flow-charts (line FC in the table). Of course, a single tasklet can address more topics, usually pertaining to different areas.

The criteria underlying the structure of the two versions of the test include: use of one or two multiple-choice questions for each tasklet; balanced distribution, relative to the two versions, among areas and topics (see Table 4.1); reasonable size to complete the test in about one hour (based on our previous experience). This instrument has then been made accessible online, the version and the code/flow-chart format of some tasklets being assigned randomly.



```
int x = m;
int y = n;

while ( x != y ) {

  while ( x < y ) {
    x = x + m;
  }
  while ( x > y ) {
    y = y + n;
  }
}
output( "x = " + x );
```

Figure 4.3: Equivalence by code and by flow-chart.

4.4 Tasklets

Survey 1 tasklets

Tasklet T1

Tasklet T1 aimed to explore the ability to draw connections between a simple loop condition and the statement of a problem by reasoning on a code fragment.
Problem statement: *The following program calculates the remainder and the quotient of the integer division* m/n*, where* $m \geq 0$ *and* $n > 0$*. Of the conditions below, which one is correct for the while loop?.*

```
// input: int m ≥ 0, int n > 0
int x = m;
int y = 0;
while (   ??   ) {
  x = x - n;
  y = y + 1;
}
output( "x = " + x + ", y = " + y );
```

Figure 4.4: Tasklet T1.

The six available options were: $x > 0$, $x < n$, $x \geq n$, $x > n$, $x < m$, $y \leq n$.

Table 4.2: Classification of tasklets across areas/topics and test versions.

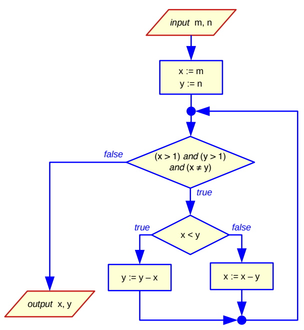
Ref. to Table 4.1	**Version 1**	**Version 2**
A.1.a	T4, T10	T5, T9
A.1.b	T2, T7b	T7a, T8
A.2.a	T1	T3b, T6
A.2.b	T3a, T7b	T7a
B.1.a	T1, T3a, T7b	T7a, T8, T9
B.1.b	T2	T3b, T6
B.1.c	T4, T10	T5
B.2.a	T1	T3b
B.2.b	T7b, T10	T5, T6, T7a
B.2.c	T7b	T8
B.3.a	T1, T4, T7b, T10	T5, T6, T7a, T9
B.3.b	T2, T3a	T3b, T8
B.3.c	T3a	T3b
C.1	T4, T10	T5
C.2	T2, T3a	T8, T9
D.1	T1, T2, T4	T5
D.2	T3a, T7b, T10	T3b, T6, T7a, T8, T9
D.3	T7b, T10	T5, T7a
FC	T2, T3a, T4	T3b, T8, T9
SC	T1, T2, T3a, T4, T7b, T10	T3b, T5, T6, T7a, T8, T9

Tasklet T2

Tasklet T2/i addressed students' mastery of the "mechanics" of the execution of a loop controlled by a non-trivial condition and included a nested if.
Problem statement: *With reference to the following program, if the input values are* $m = 15$ *and* $n = 44$*, how many iterations of the while loop will be performed?.*

```
// input: int m > 0, int n > 0
int x = m;
int y = n;
while ( (x > 1) && (y > 1) && (x != y) ) {
  if ( x < y ) {
    y = y - x;
  } else {
    x = x - y;
  }
}
output( "x = " + x + ", y = " + y );
```

(a) Code version



(b) Flow-chart version

Figure 4.5: Tasklet T2.

The six available options were: no iteration, 1, 2, 3, 4 or more, the loop never ends.

Tasklet T2/ii addressed students' mastery of reversibility regarding execution of a loop controlled by a non-trivial condition and included a nested if.
Problem statement: *If at the end of the program execution the values reported in*

the output are $x = 1$ and $y = 14$, what were the input values of m and n?.

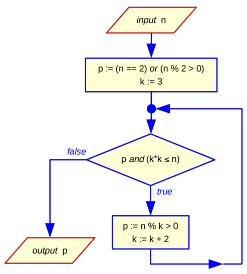
The six available options were: x = 1 and y =14 they can never be the output of the program, m = 7 and n = 28, m = 15 and n = 44, m = 44 and n = 1, m = 162 and n = 10, we cannot know the input values of m and n because there are multiple possibilities.

Tasklet T3a

Tasklet T3a/i addressed students' mastery of the "mechanics" regarding execution of a loop controlled by a non-trivial condition.
Problem statement: *With reference to the following program, if the input value is $n = 23$, how many iterations of the while loop will be executed? (The % operation denotes the remainder of the integer division.).*

```
// input: int n > 1
boolean p = (n == 2) || (n % 2 > 0);
int k = 3;
while ( p && (k*k <= n) ) {
  p = n % k > 0;
  k = k + 2;
}
output( "p = " + p );
```

(a) Code version



(b) Flow-chart version

Figure 4.6: Tasklet T3a.

The six available options were: no iteration, 1, 2, 3, 4 or more, the loop never ends.

Tasklet T3a/ii addressed the students' ability to identify the functional purpose. Problem statement: *And could you identify the purpose of the program, for any input value $n > 1$? (The program is the same as in the previous question.).*

The six available options were: calculate the smallest divisor of n, calculate the greatest divisor of n, check if n is a prime number, check if k is a divisor of n, calculate the integer part of the square root of n, starting at 3 and until p is true count how many times 2 can be added before reaching the square root of n.

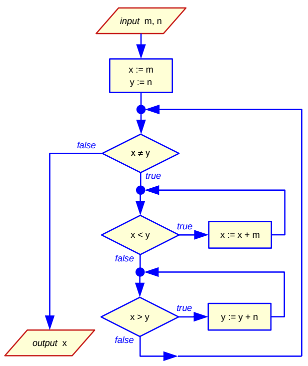
Tasklet T4

Tasklet T4/i addressed the students' ability to grasp comprehensively nested combinations of conditionals and iteration constructs.

Problem statement: *With reference to the following program, if the input values are* $m = 6$ *and* $n = 21$*, what will be the output value of* x*?*

```
// input: int m > 0, int n > 0
int x = m;
int y = n;
while ( x != y ) {
  while ( x < y ) {
    x = x + m;
  }
  while ( x > y ) {
    y = y + n;
  }
}
output( "x = " + x );
```

(a) Code version



(b) Flow-chart version

Figure 4.7: Tasklet T4.

The six available options were: x = 24, x = 27, x = 42, x = 84, x = 126, the loops never end.

Tasklet T4/ii addressed the students' ability to identify functional equivalence. Problem statement: *The program of the previous question, also reported here, is applied for integer input values* $m > 0$ *and* $n > 0$*. Which of Programs 1–4 is equivalent to it? (Two programs are equivalent if the final states, at the end of their respective executions, are always the same when the initial states are the same, and provided that the initial states satisfy the input requirements.).*

```
int x = m;
int y = n;
while ( x != y ) {
  while ( x < y ) {
    x = x + x;
  }
  while ( x > y ) {
    y = y + y;
  }
}
output( "x = " + x );
```

(a) Program 1

```
int x = m;
int y = n;
while ( x != y ) {
  do {
    x = x + m;
  } while ( x < y );
  do {
    y = y + n;
  } while ( x > y );
}
output( "x = " + x );
```

(b) Program 2

```
int x = m;
int y = n;
while ( x != y ) {
  if ( (x < y) || (x > y) ) {
    x = x + m;
    y = y + n;
  }
}
output( "x = " + x );
```

(c) Program 3

```
int x = m;
int y = n;
while ( x != y ) {
  if ( x < y ) {
    x = x + m;
  } else {
    y = y + n;
  }
}
output( "x = " + x );
```

(d) Program 4

Figure 4.8: Tasklet T4: the four programs to be compared.

Tasklet T7b

Tasklet T7b/i addressed students' mastery of reversibility regarding execution of a loop controlled concerning with array manipulation.
Problem statement: *With reference to the following program, if the final state of the*

character array is v = {'s', 't', 'o', 'p'}, what was the initial state of v before the execution of the program? (The integer v.length represents the number of components of the array v.).

```
// input: char[] v  s.t. v.length > 0
int n = v.length;  // lunghezza array v
char x = v[n-1];
for ( int i=n-2; i>=0; i=i-1 ) {
  v[i+1] = v[i];
}
v[0] = x;
output( "v = " + v );
```

Figure 4.9: Tasklet T7b.

The six available options were: v = { 's', 't', 'o', 'p' }, v = { 't', 'o', 'p', 'p' }, v = { 't', 'o', 'p', 's' }, v = { 'p', 's', 't', 'o' }, v = { 'p', 'o', 't', 's' }, v = { 't', 's', 'p', 'o' }.

Tasklet T7b/ii addressed the students' ability to identify the functional purpose. Problem statement: *What could be the purpose of the program?.*

The six available options were:

1. running the program, the state of the array v does not change;
2. move the elements of v one place to the right, losing the rightmost element;
3. rotate the elements of v one place to the right, inserting the rightmost one at the beginning;
4. rotate the elements of v one place to the left, inserting the leftmost one at the end;
5. reverse the order of the elements of v;
6. swap all subsequent pairs of components of v.

Tasklet T10

Tasklet T10/i addressed the students' ability to grasp comprehensively nested combinations of conditionals and iteration constructs.
Problem statement: *With reference to the following program, if the initial state, in input, of the string array is v = "one", "two", "ten", "ten", "ten", "two", "two", "one", "two", what will be the output values of v[k] and n? (The integer v.length represents the number of components of the array v; the equals method compares two strings and returns true if they are equal, false if they are different.).*

```
// input: String[] v  s.t. v.length > 0
int k = 0;
int n = 0;
for ( int i=0; i<v.length; i=i+1 ) {
  int c = 1;
  for ( int j=i+1; j<v.length; j=j+1 ) {
    if ( v[j].equals(v[i]) ) {
      c = c + 1;
    }
  }
  if ( c > n ) {
    k = i;
    n = c;
  }
}
output( "v[k] = " + v[k] + ", n = " + n );
```

Figure 4.10: Tasklet T10.

The six available options were: v[k] = "one", n = 2; v[k] = "two", n = 1; v[k] = "two", n = 2; v[k] = "two", n = 4; v[k] = "two", n = 9; v[k] = "ten", n = 3.

Tasklet T10/ii addressed the students' ability to identify functional equivalence. Problem statement: *The program of the previous question, also reported here, is applied when the string array v contains at least one element. Which of Programs 1–4 is equivalent to it? (Two programs are equivalent if the final states, at the end of their respective executions, are always the same with the same initial states that satisfy the input requirements.).*

```
int k = 0;
int n = 0;
int i = 0;
do {
  i = i + 1;
  int c = 1;
  for ( int j=i+1; j<v.length; j=j+1 ) {
    if ( v[j].equals(v[i]) ) {
      c = c + 1;
    }
  }
  if ( c > n ) {
    k = i;
    n = c;
  }
} while ( i < v.length );
output( "v[k] = " + v[k] + ", n = " + n );
```

(a) Program 1

```
int k = 0;
int n = 0;
int i = 0;
do {
  int c = 1;
  for ( int j=i+1; j<v.length; j=j+1 ) {
    if ( v[j].equals(v[i]) ) {
      c = c + 1;
    }
  }
  if ( c > n ) {
    k = i;
    n = c;
  }
  i = i + 1;
} while ( i < v.length );
output( "v[k] = " + v[k] + ", n = " + n );
```

(b) Program 2

```
int k = 0;
int n = 0;
int i = 0;
do {
  int c = 1;
  for ( int j=i+1; j<v.length; j=j+1 ) {
    if ( v[j].equals(v[i]) ) {
      c = c + 1;
    }
  }
  if ( c > n ) {
    k = i;
    n = c;
  }
  i = i + 1;
} while ( i == v.length );
output( "v[k] = " + v[k] + ", n = " + n );
```

(c) Program 3

```
int k = 0;
int n = 0;
int i = 0;
do {
  int c = 1;
  for ( int j=i+1; j<v.length; j=j+1 ) {
    if ( v[j].equals(v[i]) ) {
      c = c + 1;
    }
  }
  if ( c > n ) {
    k = i;
    n = c;
  }
  i = i + 1;
} while ( i >= v.length );
output( "v[k] = " + v[k] + ", n = " + n );
```

(d) Program 4

Figure 4.11: Tasklet T10: the four programs to be compared.

Survey 2 tasklets

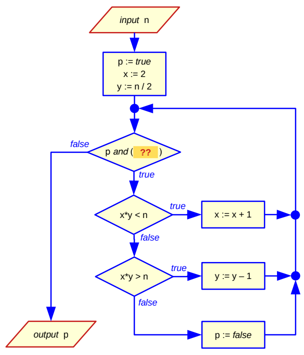
Tasklet T3b

Tasklet T3b addressed students' mastery of the "mechanics" of loop controlled execution by a non-trivial condition and included a nested if.

Problem statement: *The following program checks if the input value $n \geq 2$ is a prime number. Of the conditions below, which one correctly completes the compound condition of the while loop?.*

```
// input: int n > 1
boolean p = true;
int x = 2;
int y = n / 2;
while ( p && (  ??  ) ) {
  if ( x*y < n ) {
    x = x + 1;
  } else if ( x*y > n ) {
    y = y - 1;
  } else {
    p = false;
  }
}
output( "p = " + p );
```

(a) Code version



(b) Flow-chart version

Figure 4.12: Tasklet T3b.

The six available options were: $y \leq n$, $y > n$, $x < y$, $x \leq y$, $n \ \% \ x$, $n \ \% \ y$.

Tasklet T5

Tasklet T5/i addressed the students' ability to grasp comprehensively nested combinations of conditionals and iteration constructs.
Problem statement: *With reference to the following program, if the input value is $n = 4$, what will be the output value of b$[n/2]$ = b$[2]$? (The first statement creates an array of integers b with $n+1$ elements.).*

```
// input: int n >= 0
int[] b = new int[ n+1 ];
b[0] = 1;
for ( int i=1; i<=n; i=i+1 ) {
  b[i] = 1;
  for ( int j=i-1; j>0; j=j-1 ) {
    b[j] = b[j-1] + b[j];
  }
}
output( "b[" + (n/2) + "] = " + b[n/2] );
```

Figure 4.13: Tasklet T5.

The six available options were: b$[2] = 1$, b$[2] = 3$, b$[2] = 4$, b$[2] = 6$, b$[2] = 10$, none of the previous answers.

Tasklet T5/ii addressed students' ability to evaluate functional equivalence of nested loops concerning with array manipulation.
Problem statement: *The program of the previous question, also reported here, is applied for input values of $n \geq 0$. Which of Programs 1–4 is equivalent to it? (Two programs are equivalent if the final states, at the end of their respective executions, are always the same when the input value is the same and satisfies the indicated requirements.).*

```
int[] b = new int[ n+1 ];
b[0] = 1;
for ( int i=1; i<=n; i=i+1 ) {
  b[i] = 1;
  int j = i - 1;
  while ( j > 0 ) {
    j = j - 1;
    b[j] = b[j-1] + b[j];
  }
}
output( "b["+(n/2)+"] = "+b[n/2] );
```

(a) Program 1

```
int[] b = new int[ n+1 ];
b[0] = 1;
for ( int i=1; i<=n; i=i+1 ) {
  b[i] = 1;
  int j = i - 1;
  while ( j > 0 ) {
    b[j] = b[j-1] + b[j];
    j = j - 1;
  }
}
output( "b["+(n/2)+"] = "+b[n/2] );
```

(b) Program 2

```
int[] b = new int[ n+1 ];
b[0] = 1;
for ( int i=1; i<=n; i=i+1 ) {
  b[i] = 1;
  int j = i;
  while ( j > 0 ) {
    j = j - 1;
    b[j] = b[j-1] + b[j];
  }
}
output( "b["+(n/2)+"] = "+b[n/2] );
```

(c) Program 3

```
int[] b = new int[ n+1 ];
b[0] = 1;
for ( int i=1; i<=n; i=i+1 ) {
  b[i] = 1;
  int j = i - 1;
  do {
    j = j - 1;
    b[j] = b[j-1] + b[j];
  } while ( j > 0 );
}
output( "b["+(n/2)+"] = "+b[n/2] );
```

(d) Program 4

Figure 4.14: Tasklet T5: the four programs to be compared.

Tasklet T6

Tasklet T6 aimed to explore the ability to identify connections between a simple loop condition and the statement of a problem by reasoning on a code fragment.
Problem statement: *The following program determines how many times a non-empty pattern p, for example a word, occurs in a text t, where both p and t are character strings. Which of the conditions below on the for loop control variable is correct? (The length method returns the length of a string; equals check if two strings are equal; substring(i, i + k) extracts a substring of k characters starting from that of index i.).*

```
// input: String t, String p  s.t. p.length() > 0
int n = t.length();
int k = p.length();
int c = 0;
for ( int i=0;  ?? ; i=i+1 ) {
  if ( t.substring(i,i+k).equals(p) ) {
    c = c + 1;
  }
}
output( "c = " + c );
```

Figure 4.15: Tasklet T6.

The six available options were: $i \leq n$, $i < n$, $i < n-1$, $i \leq n-k$, $i < n+k$, $i < k$.

Tasklet T7a

Tasklet T7a/i addressed students' mastery of reversibility regarding execution of a loop controlled by a non-trivial condition and included a nested if.
Problem statement: *With reference to the following program, if the final state of the character array is v = {'s', 't', 'o', 'p'}, what was the initial state of v before the execution of the program? (The integer v.length represents the number of components of the array v.).*

```
// input: char[] v  s.t. v.length > 0
int n = v.length;
char x = v[0];
for ( int i=1; i<n; i=i+1 ) {
  v[i-1] = v[i];
}
v[n-1] = x;
output( "v = " + v );
```

Figure 4.16: Tasklet T7a.

The six available options were: v = { 's', 't', 'o', 'p' }, v = { 's', 's', 't', 'o' }, v = { 't', 'o', 'p', 's' }, v = { 'p', 's', 't', 'o' }, v = { 'p', 'o', 't', 's' }, v = { 't', 's', 'p', 'o' }.

Tasklet T7a/ii addressed the students' ability to identify the functional purpose.
Problem statement: *What could be the purpose of the program?.*

The six available options were:

1. running the program, the state of the array v does not change;
2. move the elements of v one place to the left, losing the leftmost element;
3. rotate the elements of v one place to the right, inserting the rightmost one at the beginning;
4. rotate the elements of v one place to the left, inserting the leftmost one at the end;
5. reverse the order of the elements of v;
6. swap all subsequent pairs of components of v.

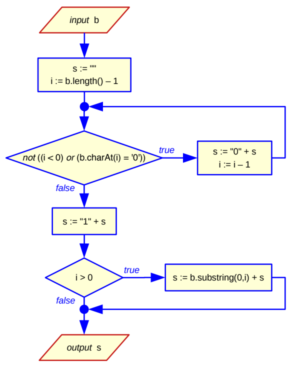
Tasklet T8

Tasklet T8/i addressed students' mastery of the "mechanics" of loop controlled execution regarding string manipulation.
Problem statement: *With reference to the following program, if the binary input string is b = "10110111", how many iterations of the while loop will be performed? (The length method returns the length of a string; + concatenates two strings; charAt*

returns the character at a given position in the string; substring (0, i) extracts a substring of characters starting from the initial one of index 0.).

```
// input: String b  of char '0'/'1'
String s = "";
int i = b.length() - 1;
while ( ! ((i < 0) || (b.charAt(i) == '0')) ) {
  s = "0" + s;
  i = i - 1;
}
s = "1" + s;
if ( i > 0 ) {
  s = b.substring(0,i) + s;
}
output( "s = " + s );
```

(a) Code version



(b) Flow-chart version

Figure 4.17: Tasklet T8.

The six available options were: no iteration, 1, 2, 3, 4 or more, the loop never ends.

Tasklet T8/ii addressed students' mastery of reversibility regarding execution of a loop controlled with string manipulation.

Problem statement: *If at the end of the program execution the output string is s = "11010000", what was the value of the input string b?.*

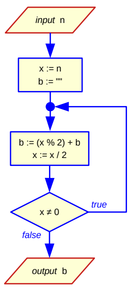
The six available options were: s = "11010000" can never be the program output, b = "11011111", b = "11000001", b = "11001111", b = "11010101", we cannot know the input string b because there are multiple possibilities.

Tasklet T9

Tasklet T9/i addressed students' mastery of the "mechanics" of loop controlled execution concerning string manipulation.
Problem statement: *With reference to the following program, if the input value is* $n = 45$*, how many iterations of the do-while loop will be performed? (The / and % operations calculate the quotient and the remainder of the integer division, respectively; + represents the string-number concatenation.).*

```
// input: int n >= 0
int x = n;
String b = "";
do {
  b = ( x % 2 ) + b;
  x = x / 2;
} while ( x != 0 );
output( "b = " + b );
```

(a) Code version



(b) Flow-chart version

Figure 4.18: Tasklet T9.

The six available options were: 4 loops, 5 loops, 6 loops, 7 loops, 8 loops, none of the previous answers.

Tasklet T9/ii addressed the students' ability to identify functional equivalence. Problem statement: *The program of the previous question, also reported here, is applied for input values $n \geq 0$. Which of Programs 1–4 is equivalent to it? (Two programs are equivalent if the final states, at the end of their respective executions, are always the same with the same input that satisfies the indicated requirements.).*

```
int x = n;
String b = "";
while ( x == 0 ) {
  b = ( x % 2 ) + b;
  x = x / 2;
}
output( "b = " + b );
```

(a) Program 1

```
int x = n;
String b = "";
while ( x != 0 ) {
  b = ( x % 2 ) + b;
  x = x / 2;
}
output( "b = " + b );
```

(b) Program 2

```
int x = n / 2;
String b = ( n % 2 ) + "";
while ( x != 0 ) {
  b = ( x % 2 ) + b;
  x = x / 2;
}
output( "b = " + b );
```

(c) Program 3

```
int x = n;
String b = "";
while ( x != 0 ) {
  b = ( x % 2 ) + b;
  x = x / 2;
}
b = ( x % 2 ) + b;
x = x / 2;
output( "b = " + b );
```

(d) Program 4

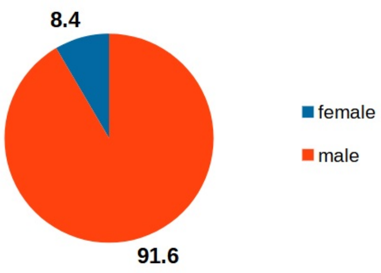
Figure 4.19: Tasklet T9: the four programs to be compared.

4.5 Data collection and results

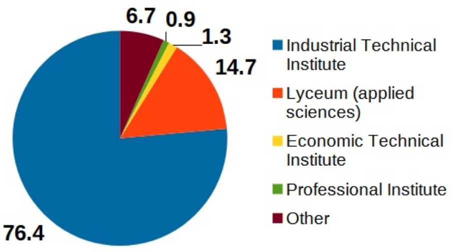
The (anonymous) survey was administered to 225 students attending the second, third and fourth year (age 15–18) of scientific and technical high schools, i.e. when the basic flow-control constructs are introduced and/or used extensively in programming activities.

4.5.1 General information

The students who responded to the survey are predominantly male (91.6%), see Figure 4.20a, and mostly attend a technical high school (76.4%), see Figure 4.20b. The percentage of female is small for one comparative analysis.



(a) Students by gender



(b) Distribution of students by type of high school

Figure 4.20: Students' general information.

The distribution of students according to they year level is shown in Figure 4.21, where we can see that most of the students attend the third year (53.8%), followed by the fourth year (28.4%) and the second year (17.8%).

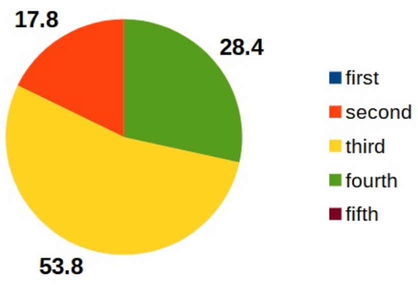


Figure 4.21: Students' year of attendance.

The languages that students know are mainly those used in school courses, see Figure 4.22, with a clear prevalence of C/C++ (35.0%), Scratch (19.0%), Java (10.8%).

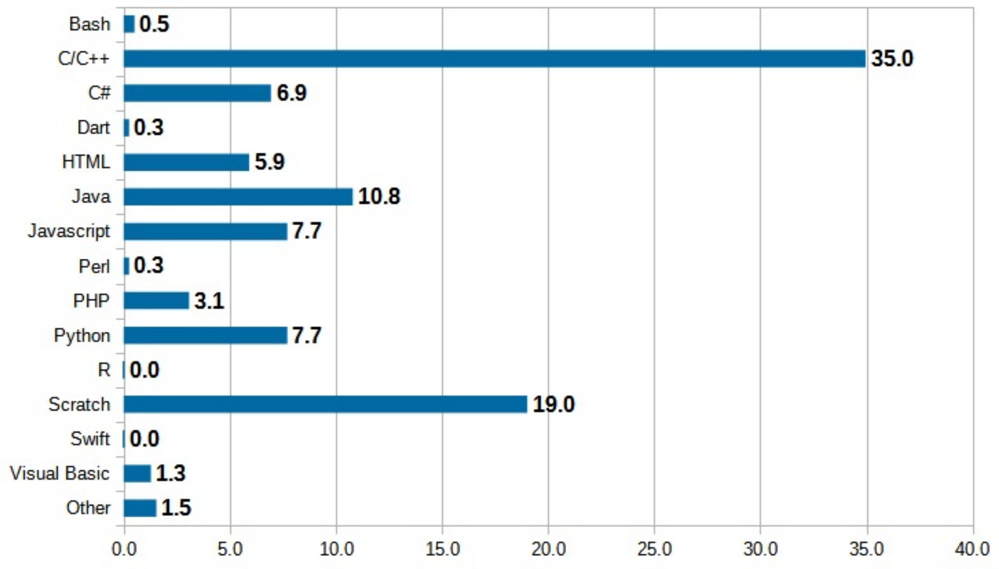


Figure 4.22: Languages known by students (more options were possible).

4.5.2 Tasklets

In this section, survey 1 and survey 2 taskset are presented. For each tasklet, the problem statement, main topic, areas and other topics to which it can refer are indicated. For each tasklet a summary table of the answers given by the students is presented. In the case of a tasklet with both code and flow-chart version, the answers are shown as total in the % column, while those specific to each version are listed in the columns % code and % flow-chart. For the more meaningful tasklets, which require more commitment on the part of the students, data on the students' self-confidence with respect to the answers given are presented.

Survey 1 tasklets

Tasklet T1

Areas and topics: A.2.a, B.1.a, B.2.a, B.3.a, D.1, SC.
Main topic: correct loop condition.

Table 4.3: Rates of chosen options for tasklet T1 (correct loop condition).

Option	%	
$x > 0$	19.1%	
$x < n$	4.6%	
$x \geq n$	42.0%	*correct option*
$x > n$	19.1%	
$x < m$	12.2%	
$y \leq n$	3.1%	

Table 4.4: Students' self-confidence for tasklet T1 (correct loop condition).

Option	%	
1	7.6%	not confident at all
2	26.7%	
3	45.8%	
4	19.8%	fully confident

Tasklet T2

Areas and topics: A.1.b, B.1.b, B.3.b, C.2, D.1, FC, SC.
T2/i main topic: number of iterations.

Table 4.5: Rates of chosen options for tasklet T2/i (number of iterations).

Option	%		% code	% flowchart
no iteration	3.8%		1.4%	6.8%
1 iteration	7.6%		8.3%	6.8%
2 iterations	11.5%		8.3%	15.3%
3 iterations	54.2%	*correct option*	61.1%	45.8%
4 or more iterations	16.8%		13.9%	20.3%
the loop never ends	6.1%		6.9%	5.1%

T2/ii main topics: reversibility.

Table 4.6: Rates of chosen options for tasklet T2/ii (reversibility).

Option	%		% code	% flowchart
x = 1 and y =14 can never be the output of the program	13.7%		18.1%	8.5%
m = 7 and n = 28	4.6%		2.8%	6.8%
m = 15 and n = 44	49.6%		56.9%	40.7%
m = 44 and n = 15	6.1%		2.8%	10.2%
m = 162 and n = 103	3.8%		2.8%	5.1%
we cannot know the input values of m, n because there are multiple possibilities	22.1%	*correct option*	16.7%	28.8%

Table 4.7: Students' self-confidence for tasklet T2/ii (reversibility).

Option	%		% code	% flowchart
1	6.9%	not confident at all	9.7%	3.4%
2	22.9%		25.0%	20.3%
3	38.9%		34.7%	44.1%
4	31.3%	fully confident	30.6%	32.2%

Tasklet T3a

Areas and topics: A.2.b, B.1.a, B.3.b, B.3.c, C.2, D.2, FC, SC.
T3a/i main topic: number of iterations.

Table 4.8: Rates of chosen options for tasklet T3a/i (number of iterations).

Option	**%**		**% code**	**% flowchart**
no iteration	12.2%		11.1%	13.6%
1 iteration	44.3%	*correct option*	45.8%	42.4%
2 iterations	14.5%		18.1%	10.2%
3 iterations	13.0%		9.7%	16.9%
4 or more iterations	10.7%		9.7%	11.9%
the loop never ends	5.3%		5.6%	5.1%

T3a/ii main topics: functional purpose.

Table 4.9: Rates of chosen options for tasklet T3a/ii (functional purpose).

Option	**%**		**% code**	**% flowchart**
to compute the smallest divisor of n	4.6%		2.8%	6.8%
to compute the greatest divisor of n	10.7%		11.1%	10.2%
to check if n is a prime number	34.4%	*correct option*	34.7%	33.9%
to check if k is a divisor of n	25.2%		26.4%	23.7%
to compute the integer part of the square root of n	6.1%		6.9%	5.1%
starting at 3 and until p is true, to count how many times 2 can be added before reaching the square root of n	19.1%		18.1%	20.3%

Table 4.10: Students' self-confidence for tasklet T3a/ii (functional purpose).

Option	**%**		**% code**	**% flowchart**
1	24.4%	not confident at all	20.8%	28.8%
2	37.4%		37.5%	37.3%
3	26.7%		30.6%	22.0%
4	11.5%	fully confident	11.1%	11.9%

Tasklet T4

Areas and topics: A.1.a, B.1.c, B.3.a, C.1, D.1, FC, SC.
T4/i main topic: output state.

Table 4.11: Rates of chosen options for tasklet T4/i (output state).

Option	%		% code	% flowchart
x = 24	14.5%		22.2%	5.1%
x = 27	9.9%		9.7%	10.2%
x = 42	45.0%	*correct option*	34.7%	57.6%
x = 84	4.6%		2.8%	6.8%
x = 126	3.1%		5.6%	0.0%
the loops never end	22.9%		25.0%	20.3%

T4/ii main topics: equivalence.

Table 4.12: Rates of chosen options for tasklet T4/ii (equivalence).

Option	%		% code	% flowchart
Program 1	18.3%		22.2%	13.6%
Program 2	26.0%		19.4%	33.9%
Program 3	11.5%		13.9%	8.5%
Program 4	44.3%	*correct option*	44.4%	44.1%

Table 4.13: Students' self-confidence for tasklet T4/ii (equivalence).

Option	%		% code	% flowchart
1	9.2%	not confident at all	9.7%	8.5%
2	19.8%		20.8%	18.6%
3	36.6%		40.3%	32.2%
4	34.4%	fully confident	29.2%	40.7%

Tasklet T7b

Areas and topics: A.1.b, A.2.b, B.1.a, B.2.b, B.2.c, B.3.a, D.2, SC.
T7b/i main topic: reversibility.

Table 4.14: Rates of chosen options for tasklet T7b/i (reversibility).

Option	%	
v = 's', 't', 'o', 'p'	9.2%	
v = 't', 'o', 'p', 'p'	3.8%	
v = 't', 'o', 'p', 's'	42.7%	*correct option*
v = 'p', 's', 't', 'o'	21.4%	
v = 'p', 'o', 't', 's'	19.1%	
v = 't', 's', 'p', 'o'	3.8%	

Table 4.15: Students' self-confidence for tasklet T7b/i (reversibility).

Option	%	
1	17.6%	not confident at all
2	35.9%	
3	22.9%	
4	23.7%	fully confident

T7b/ii main topic: functional purpose.

Table 4.16: Rates of chosen options for tasklet T7b/ii (functional purpose).

Option	%	
by running the program, the state of the array v does not change	8.4%	
to move the elements of v one place to the right, losing the rightmost element	7.6%	
to rotate the elements of v one place to the right, inserting the rightmost one at the beginning	37.4%	*correct option*
to rotate the elements of v one place to the left, inserting the leftmost one at the end	28.2%	
to reverse the order of the elements of v	16.0%	
to swap all subsequent pairs of elements of v	2.3%	

Table 4.17: Students' self-confidence for tasklet T7b/ii (functional purpose).

Option	%	
1	15.3%	not confident at all
2	28.2%	
3	30.5%	
4	26.0%	fully confident

Tasklet T10

Areas and topics: A.1.a, B.1.c, B.2.b, B.3.a, C.1, D.2, D.3, SC.
T10/i main topic: output state.

Table 4.18: Rates of chosen options for tasklet T10/i (output state).

Option	%	
v[k] = "one", n = 2	9.2%	
v[k] = "two", n = 1	8.4%	
v[k] = "two", n = 2	16.0%	
v[k] = "two", n = 4	42.7%	*correct option*
v[k] = "two", n = 9	13.0%	
v[k] = "ten", n = 3	10.7%	

T10/ii main topics: equivalence.

Table 4.19: Rates of chosen options for tasklet T10/ii (equivalence).

Option	%	
Program 1	20.6%	
Program 2	36.6%	*correct option*
Program 3	24.4%	
Program 4	18.3%	

Table 4.20: Students' self-confidence for tasklet T10/ii (equivalence).

Option	%	
1	23.7%	not confident at all
2	34.4%	
3	29.0%	
4	13.0%	fully confident

Survey 2 tasklets

Tasklet T3b

Areas and topics: A.2.a, B.1.b, B.2.a, B.3.b, B.3.c, D.2, FC, SC.
T3b main topic: correct loop condition.

Table 4.21: Rates of chosen options for tasklet T3b (correct loop condition).

Option	%		% code	% flowchart
$y \leq n$	14.9%		14.0%	15.7%
$y > n$	6.4%		9.3%	3.9%
$x < y$	17.0%		18.6%	15.7%
$x \leq y$	24.5%	*correct option*	25.6%	23.5%
$n \% x$	22.3%		16.3%	27.5%
$n \% y$	14.9%		16.3%	13.7%

Table 4.22: Students' self-confidence for tasklet T3b (correct loop condition).

Option	%		% code	% flowchart
1	13.8%	not confident at all	18.6%	9.8%
2	44.7%		39.5%	49.0%
3	34.0%		32.6%	35.3%
4	7.4%	fully confident	9.3%	5.9%

Tasklet T5

Areas and topics: A.1.a, B.1.c, B.2.b, B.3.a, C.1, D.1, D.3, SC.
T5/i main topic: output state.

Table 4.23: Rates of chosen options for tasklet T5/i (output state).

Option	%	
b[2] = 1	20.2%	
b[2] = 3	20.2%	
b[2] = 4	16.0%	
b[2] = 6	17.0%	*correct option*
b[2] = 10	4.3%	
none of the previous answers	22.3%	

T5/ii main topics: equivalence.

Table 4.24: Rates of chosen options for tasklet T5/ii (equivalence).

Option	%	
Program 1	12.8%	
Program 2	51.1%	*correct option*
Program 3	17.0%	
Program 4	19.1%	

Table 4.25: Students' self-confidence for tasklet T5/ii (equivalence).

Option	%	
1	21.3%	not confident at all
2	35.1%	
3	28.7%	
4	14.9%	fully confident

Tasklet T6

Areas and topics: A.2.a, B.1.b, B.2.b, B.3.a, D.2, SC.
T6 main topic: correct loop condition.

Table 4.26: Rates of chosen options for tasklet T6 (correct loop condition).

Option	%	
$i \leq n$	18.1%	
$i < n$	28.7%	
$i < n - 1$	9.6%	
$i \leq n - k$	23.4%	*correct option*
$i < n + k$	10.6%	
$i < k$	9.6%	

Table 4.27: Students' self-confidence for tasklet T6 (correct loop condition).

Option	%	
1	23.4%	not confident at all
2	37.2%	
3	30.9%	
4	8.5%	fully confident

Tasklet T7a

Areas and topics: A.1.b, A.2.b, B.1.a, B.2.b, B.3.a, D.2, D.3, SC.
T7a/i main topic: reversibility.

Table 4.28: Rates of chosen options for tasklet T7a/i (reversibility).

Option	%	
v = 's', 't', 'o', 'p'	7.4%	
v = 's', 's', 't', 'o'	2.1%	
v = 't', 'o', 'p', 's'	29.8%	
v = 'p', 's', 't', 'o'	40.4%	*correct option*
v = 'p', 'o', 't', 's'	18.1%	
v = 't', 's', 'p', 'o'	2.1%	

Table 4.29: Students' self-confidence for tasklet T7a/i (reversibility).

Option	%	
1	18.1%	not confident at all
2	20.2%	
3	34.0%	
4	27.7%	fully confident

T7a/ii main topic: functional purpose.

Table 4.30: Rates of chosen options for tasklet T7a/ii (functional purpose).

Option	%	
by running the program, the state of the array v does not change	4.3%	
to move the elements of v one place to the left, losing the leftmost element	8.5%	
to rotate the elements of v one place to the right, inserting the rightmost one at the beginning	23.4%	
to rotate the elements of v one place to the left, inserting the leftmost one at the end	45.7%	*correct option*
to reverse the order of the elements of v	17.0%	
to swap all subsequent pairs of elements of v	1.1%	

Table 4.31: Students' self-confidence for tasklet T7a/ii (functional purpose).

Option	%	
1	14.9%	not confident at all
2	26.6%	
3	25.5%	
4	33.0%	fully confident

Tasklet T8

Areas and topics: A.1.b, B.1.a, B.2.c, B.3.b, C.2, D.2, FC, SC.
T8/i main topic: number of iterations.

Table 4.32: Rates of chosen options for tasklet T8/i (number of iterations).

Option	%		% code	% flowchart
no iteration	11.7%		16.3%	7.8%
1 iteration	2.1%		0.0%	3.9%
2 iterations	19.1%		20.9%	17.6%
3 iterations	33.0%	*correct option*	30.2%	35.3%
4 or more iterations	29.8%		27.9%	31.4%
the loop never ends	4.3%		4.7%	3.9%

T8/ii main topics: reversibility.

Table 4.33: Rates of chosen options for tasklet T8/ii (reversibility).

Option	%		% code	% flowchart
s = "11010000" can never be the program output	31.9%		30.2%	33.3%
b = "11011111"	3.2%		0.0%	5.9%
b = "11000001"	10.6%		9.3%	11.8%
b = "11001111"	23.4%	*correct option*	18.6%	27.5%
b = "11010101"	2.1%		2.3%	2.0%
We cannot know the input string b because there are multiple possibilities	28.7%		39.5%	19.6%

Table 4.34: Students' self-confidence for tasklet T8/ii (reversibility).

Option	%		% code	% flowchart
1	45.7%	not confident at all	48.8%	43.1%
2	28.7%		25.6%	31.4%
3	19.1%		23.3%	15.7%
4	6.4%	fully confident	2.3%	9.8%

Tasklet T9

Areas and topics: A.1.a, B.1.a, B.3.a, C.2, D.2, FC, SC.
T9/i main topic: number of iterations.

Table 4.35: Rates of chosen options for tasklet T9/i (number of iterations).

Option	%		% code	% flowchart
4 iterations	9.6%		11.6%	7.8%
5 iterations	16.0%		11.6%	19.6%
6 iterations	44.7%	*correct option*	41.9%	47.1%
7 iterations	10.6%		11.6%	9.8%
8 iterations	4.3%		4.7%	3.9%
none of the previous answers	14.9%		18.6%	11.8%

T9/ii main topics: equivalence.

Table 4.36: Rates of chosen options for tasklet T9/ii (equivalence).

Option	%		% code	% flowchart
Program 1	9.6%		14.0%	5.9%
Program 2	53.2%		48.8%	56.9%
Program 3	20.2%	*correct option*	20.9%	19.6%
Program 4	17.0%		16.3%	17.6%

Table 4.37: Students' self-confidence for tasklet T9/ii (equivalence).

Option	%		% code	% flowchart
1	22.3%	not confident at all	18.6%	25.5%
2	21.3%		18.6%	23.5%
3	35.1%		44.2%	27.5%
4	21.3%	fully confident	18.6%	23.5%

4.6 Discussion

In this section we use the experimental results presented in Section 4.5 to answer the research questions listed in Section 4.3. The analysis of the questions is based on the schema presented in Table 4.1.

Some tasklets are proposed in two distinct versions, the first version easier and can be achieved simply by tracing the program execution for a given input, while the second one presents a higher degree of difficulty and is more abstract.

4.6.1 Students' grasp of technical features implied by iteration

Q1. *To what extent are students at ease with a range of "technical" features implied by iteration? (E.g., structure and role of loop conditions, loop-control variables, nesting of flow-control constructs, looping over arrays.)*

The tasklets related to this question can refer to the following areas.

- Structural features:
  Plain loop (6) / Loop with nested conditional (3) / Nested loops (3)
- Features of the implied conditions:
  Simple condition (8) / Composite condition (4)
- Basic data types:
  Only numerical data (4) / Non-numerical data (8)
- Additional data-related features:
  Boolean variable (2) / Data array (4)
- Focus of the processing plan:
  Exit condition (2) / Loop control variable (5) / Downward for loop (2)

The results of the single tasklets presented in the previous section do not allow an overview that answers the question posed. Therefore, we tried to categorize the students' answers as correct, incorrect and seriously incorrect by comparing the results of the tasklets. Figure 4.23 shows the degree of accuracy of the students' answers, for all the tasklets.

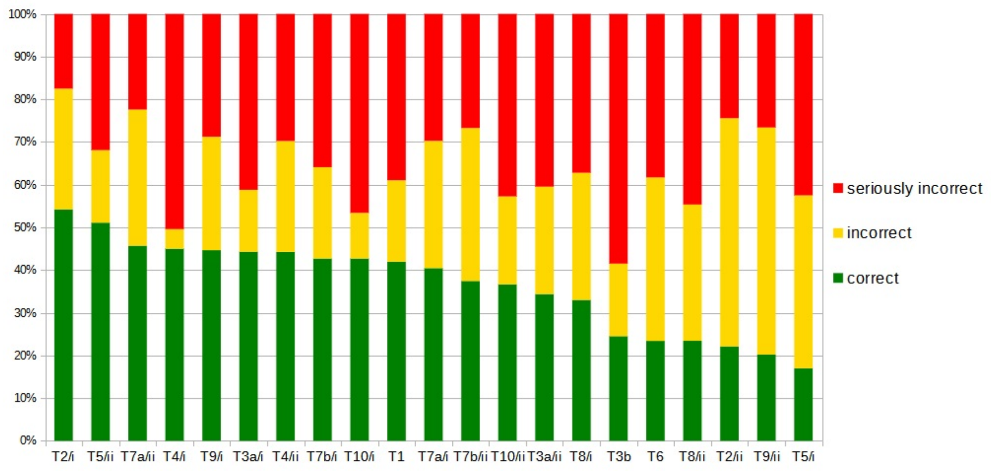


Figure 4.23: Students' degree of accuracy, in decreasing order.

The results shown in Figure 4.23 highlight that only in tasklet T2/i more than 50% of the students answered correctly, hence it appears that the majority of them is at ease with the functioning of iteration combined with a nested conditional, as well as with the interpretation of a composite (loop) condition including logical operators. The result is clearly different in T2/ii, where a reversibility problem was proposed for the same code fragment, which evidently has a greater degree of difficulty, perhaps due to the fact that this type of reversible problem is rarely proposed in high schools.

Interestingly, many answered incorrectly in tasklet T1 (more than 60%), apparently an easy problem with a very simple condition — a similar result was obtained in the previous students' survey in Chapter 2 Section 2.2.6 —. The difference between $x \geq n$ and $x > n$ can be found in the analysis of the last iteration, verifying with tracing, but evidently not all students thought to test, by tracing, their hypothesis. The same result was obtained for T3b in survey 2, with the difference between $x \leq y$, the correct option, and $x < y$.

Particularly surprising is what emerged in T4/i, apparently a problem of identifying the output state making use of tracing. The two nested While-loops have caused some confusion, especially in the code version, and flow-chart may help students in nested construct analysis.

In survey 2 tasklet T5/ii is the one with a level of correct answers greater than 50%. This result is interesting as the students are not used to these types of exercises related to the equivalence of code fragments, probably due to i) familiarity with the various types of iterations, ii) the practice of teachers to show the equivalence between While-loop and For-loop, iii) and to propose alternative versions of solutions with various types of iteration (as seen in Chapter 2, Section 2.1.5). Otherwise, in T5/i, for the same code fragment, the students could not find the correct solution. This result may be due to various factors: i) difficulty in managing the double For-loop — especially for second year students —, ii) lack of familiarity with tracing nested iterations, iii) little practice in handling backward iterations.

Even in T9/ii the number of correct answers is low, although students are well aware of the equivalence between While-loop and Do-While-loop. The reason might be related to a lack of mastery in string management. Tasklet T6 highlights some students' difficulties in identifying the correct condition. In this case many answered $i < n$ rather than $i \leq n - k$, which was the correct option; this result can be attributed to students' habit of manipulating strings from the first character to the last.

In tasklet T8 a factor that compromised the accuracy of the proposed solution could be the composite and negated condition of the While-loop. Not everyone may have grasped the concepts related to Boolean operators, as well as De Morgan's laws, but they might also have some difficulties with managing strings.

It emerged that survey 2 was more challenging, students' performance is slightly worse, with tasklets that highlighted greater difficulties for the students, and in fact more abstract questions were present.

In conclusion, we can briefly describe the following.

Structural features We can then presume that the majority of students develop a viable and accurate enough mental model of the *notional machine* underlying code execution, including the functioning of nested constructs. However, difficulties have arisen when complex nested constructs are present, or when the nested loop is a *downward loop*.

Features of the implied conditions Simple conditions are less problematic than composite conditions. The composite conditions require the mastery of Boolean operators, and in particular the difficulty in managing the *not* operation has emerged. However, some simple conditions also highlighted difficulties: i) lack of mastery in the use of order operators (e.g. $x < y$ vs. $x \leq y$), ii) stereotypes in the conditions related to the manipulation of strings (e.g. $i < n$ prevalence).

Basic data types The data collected show better performance of students in tasklets with numerical data only, while in those with non-numerical data the results are worse. The students did not acquire sufficient mastery with arrays but especially with strings.

Additional data-related features Students have some difficulty with both Arrays and Boolean values. In particular, when composite loop conditions contain Boolean values, it is difficult for students to identify the exit condition. Fewer difficulties are encountered with arrays, especially when the tasklets are similar to the problems proposed in the teacher's lesson (as it emerged in Chapter 3, Section 3.4.3).

Focus of the processing plan Performances show that a large part of students are unable to master the relationships between loop condition and accurate specification in the application domain. This may possibly be ascribed to confusion about the role of the loop condition, meant as an 'exit' condition instead of a 'continue' condition, or to some more basic lack of problem-solving skills. Moreover, this may be due to a lack of mathematical prerequisites (e.g. manage order operators), or to the lack of familiarity when dealing with uncommon problems (e.g. downward loops).

4.6.2 Tracing and higher-level thinking

Q2. *Does the effort to trace code facilitate thinking of the overall computation at a higher abstraction level?*

The tasklets related to this question can refer to the following areas.

- Correlation between answers to "tracing-based" and "more abstract" questions:

1st question (misc)		*2nd question (high level)*	
Output/final state	→	Equivalence	(3)
Number of iterations	→	Equivalence	(1)
Number of iterations	→	Reversibility	(2)
Number of iterations	→	Functional purpose	(1)
Reversibility	→	Functional purpose	(2)
no question	→	Completion	(3)

The analysis of the previous question has highlighted that students have mastered tracing quite well (see Figure 4.23), and they use it easily when it comes to indicating the output status or the number of iterations (e.g. T2/i, T3a/i, T4a/i, T10/i, T9/i). However, they fail to effectively exploit this mastery of tracing when they have to indicate the functional purpose (e.g. T3a/ii).

In any case, the identification of the functional purpose is simpler for problems similar to those proposed to them by teacher, such as e.g. T7b/ii, T7a/ii.

For equivalence problems (e.g. T4/i, T5/ii), rather than resorting to tracing, they work by analogies and similarities, identifying similarities between While-loop and For-loop, or between While-loop and Do-While-loop; equivalence schemes between the various types of iterations are often proposed by teachers (see Chapter 3, Section 3.4). However, in T9/ii it emerges that the equivalence between While-loop and Do-While-loop is not well mastered, but above all that the development of a solution hypothesis is not preceded by an adequate tracing phase, which would allow a better evaluation of the hypothesis.

Furthermore, in survey 1 students' performance in tracing-based tasklets is better than in survey 2, where tasklets are more abstract. In survey 2, this result is not confirmed for tasklets T5/ii and T7a/ii; in particular, regarding T7a, the students probably recognized a problem similar to one that was presented to them in class by the teacher. However, in T5/i the tracing phase was not effective, and this could be due to i) difficulty in tracing the double For-loop, ii) lack of familiarity with the downward For-loop, iii) little practice with a Java-like syntax.

Considering these results, it can be said that students perform better with simpler and tracing-based problems, rather than when facing problems that present greater difficulties and level of abstraction, such as questions regarding equivalence, reversibility, or identifying functional purpose.

4.6.3 Flow-chart vs. code representation

Q3. *Are students' answers more accurate when using flow-chart or textual code representations of programs?*

The tasklets related to this question can refer to the following areas.

- Flow-chart vs. code in two randomly assigned versions of the same task:
  Plain loop (3) / Loop with nested conditional (2) / Nested loops (1)
  (including focus on exit condition and downward for loop)

The results of the individual tasklets presented in the previous section do not allow to have a clear idea of the performance of the students in the two different versions. Figure 4.24 allows to compare the code version and flow-chart version performance.

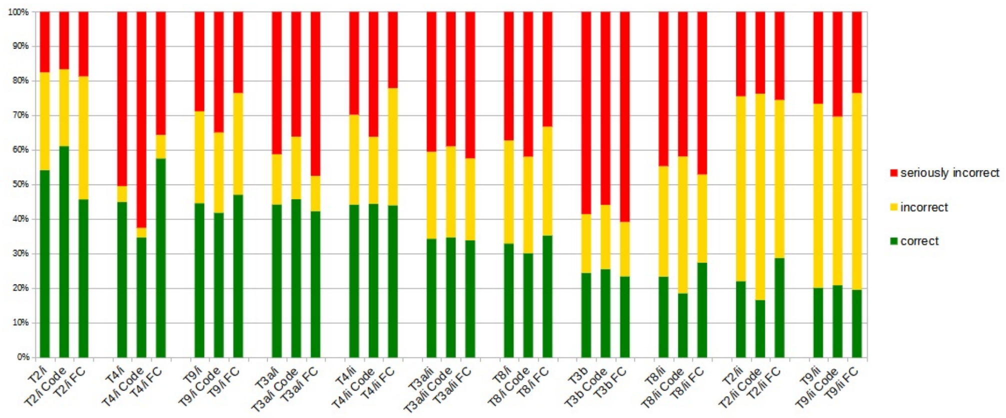


Figure 4.24: Students' performance comparing average, code and flow-chart versions, in decreasing order by average.

Tasklets T2/i shows better performance in code version, perhaps due to easier readability of the While-loop composite condition (e.g. AND vs. &&). In fact, the same result was also obtained in tasklet T2/ii. In T4/i the flow-chart version could better clarify the flow of actions to be performed, in the presence of two nested While-loops. In survey 2, the only relevant difference is found in T8. In the flow-chart version the performances are better, perhaps due to a greater readability of the composite condition, where the Boolean operations are written in English and not in Java-like style.

It seems that students have better results in the flow-chart version when the problem is simple and numerical, while when the problem becomes more complex or manipulates arrays the students perform better in the code version. However, the difference in performance is not significant.

From the collected data it is not clear whether using the flow-chart facilitates understanding, as Rahimi et al. [RBH17] argued instead.

In summary, the following results can be identified.

Plain loop The students have mastered the subject, there are no significant deviations between the code version and the flow-chart version, apart from the cases of reversibility analysis and composite condition. In the case of reversibility tasklets, it would seem that the flow-chart version allows for higher and more successful performances, as well as in the presence of composite conditions, where the Boolean operators are explicit and better identifiable.

Loop with nested conditional The performance in code version or in flow-chart version are similar.

Nested loops As in the previous point, the performances in code version or in flow-chart version are similar, but T4/i highlights that the use of flow-charts can help students understand nested constructs.

4.6.4 Students' self-confidence

Q4. *To what extent are students self-confident about their comprehension of a program's overall computation and purpose?*

The tasklets related to this question can refer to the following areas.

- Subjective perception of self-confidence on a 4-grade Likert scale:
  Program equivalence (4) / Reversibility (4) / Plan completion (3) / Functional purpose (3)

The students' perception of their own self-confidence varies according to the tasklets, as Figure 4.25 shows. There are tasklets, e.g. T1, T2, T4, T7a, T7b/ii, T9, where students declared significant level of self-confidence. Other tasklets, e.g. T3a, T3b, T7b/i, T5, T6, T8, T10, were evidently more challenging, and students declared a lower level of self-confidence.

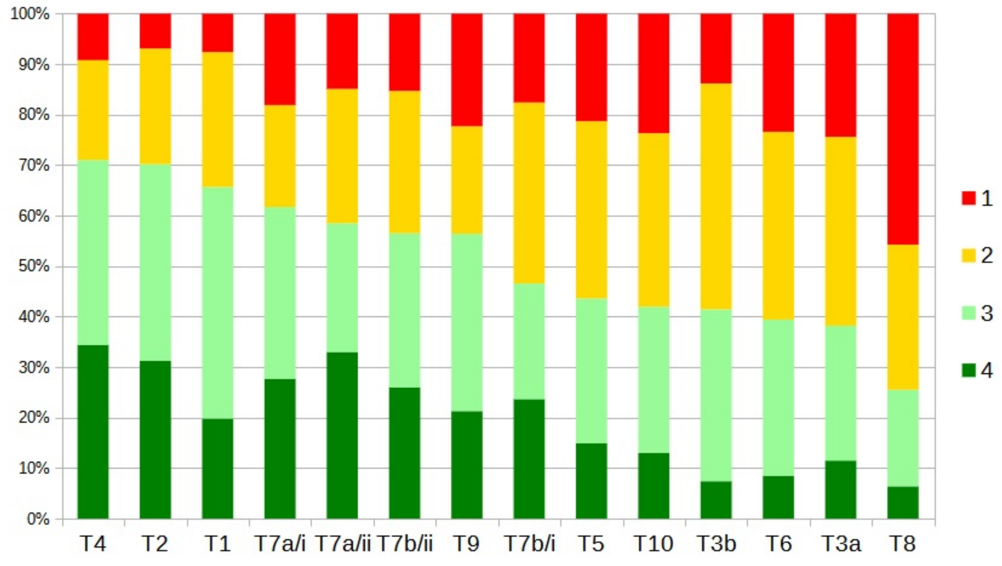


Figure 4.25: Students' self-confidence, in decreasing order

In the previous survey (see Chapter 2, Section 2.2.7) we established that students understand the functioning of an iteration, and we can therefore presume that most high school students develop a viable and accurate enough mental model of the *notional machine* underlying code execution, including the functioning of nested constructs and the evaluation of relatively complex conditions. This awareness is greatest where students faces simple problems or problems similar to those presented to them by their teacher (see Chapter 3, Section 3.4). The students therefore tend to be more confident when dealing with iterations where they can use the tracing capability to search for the output state, the numbers of iterations or identify simple loop condition. When they are required to identify the functional purpose, both in code and flow-chart version, they are less confident (e.g. T3a).

The confidence level is very low when they have to manage reversibility problems, e.g. T8, probably because they are not used to thinking in these terms, or working on the input state instead of the output state.

The same hypothesis can be made for functional equivalence problems, e.g. T10, where students are probably unfamiliar with working in terms of equivalence. However, in tasklet T4 students' self-confidence is high, and maybe this result is due to two factors: i) enough mental model of the notional machine underlying loop execution, including the functioning of nested constructs; ii) ability to recognize the functional equivalence between While-loop and Do-While-loop.

The result of T3b tasklet is surprising, where students had to identify the correct condition of the loop, a composite condition. In this case it can be assumed that the following factors play an important part:

- unfamiliarity with handling Boolean values, even in the condition;
- higher complexity in nested flow-control construct than those students are used to;
- difficulty in understanding the syntax of the language, e.g. % operator;
- students tend to not exploit their tracing abilities in order to test their conjectures about program behavior.

Figure 4.26 shows the students' levels of self-confidence compared to the accuracy of their responses. The chart compares the level of accuracy in the responses to the tasklets with the level of self-confidence declared by the student, highlighting, in general, that students overestimate their knowledge and skills.

Generally, students have a higher level of self-confidence with respect to their own performance [2], thus the students seem to underestimate their difficulties when dealing with loop conditions. This underestimation of their own difficulties seems higher when tasklets are similar to the problems that the teacher tends to present in class, or when tasklets seem apparently easy. In T5, where the tasklet presented a double level of difficulty, with nested loops and array manipulation, students stated their uncertainty, in terms of self-confidence.

[2]We have compared each student's confidence with his accuracy.

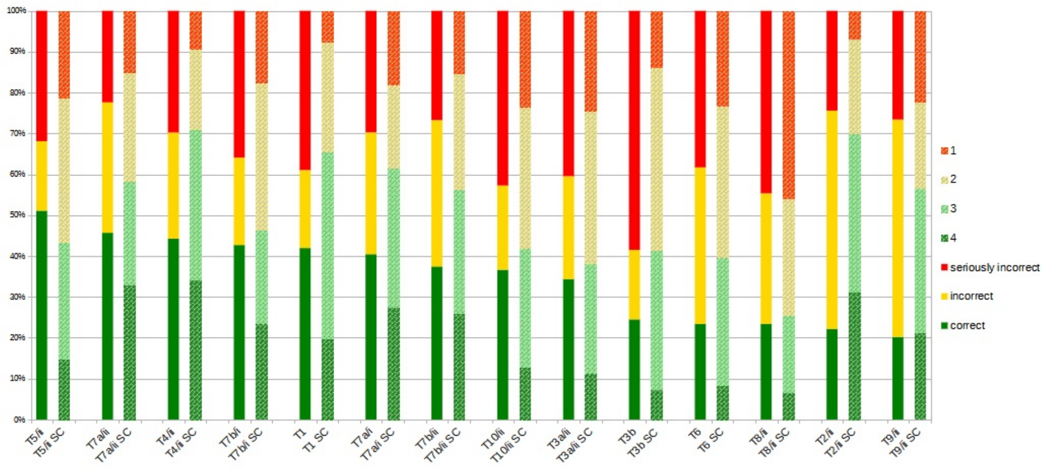


Figure 4.26: Students' accuracy and self-confidence level, in decreasing order.

The result in T1 confirms what has already been found previously (see Chapter 2, Section 2.2.7), and this may be ascribed to confusion about the role of the loop condition, meant as an 'exit' condition instead of a 'continue' condition, to some more basic lack of problem-solving skills, or to fragile mathematical prerequisites and confusion about order relations.

The outcomes in T2 and T9 are significant. Apparently T2 could have been considered easy by the students, but probably the composite condition was more difficult to analyze than expected, as perhaps novices does not grasp Boolean operators. In T9 the simplicity of the loop, as well as the knowledge of the transformation schemes from Do-While-loop to While-loop, induced the students to trust their own assumptions without verifying them with an adequate tracing phase.

Difference in students' self-confidence, comparing code version and the flow-chart version, is shown in Figure 4.27.

A difference in students' self-confidence between the two versions clearly emerges in tasklet T4, where a flow-chart probably allows to better identify the flow of actions, as well as the distinction of the two nested While-loops. In T9, higher levels of self-confidence are reported in the flow-chart version, but in general students are more confident in the code version. This result is interesting since it could be due to the fact that the more prepared or more confident students master flow-chart representation, but in general students, when dealing with strings, prefer the code version, perhaps because they have been used to dealing with strings directly by coding. A similar result emerges in T3a, where higher levels of self-confidence are reported in the flow-chart version, but in general they are more confident in the code version. A flow-chart probably illustrates the workflow more clearly in a a Do-While-loop, but pupils are unsure about handling Boolean values with this tool. Tasklet T8 highlights a higher level of self-confidence in the flow-chart version, and

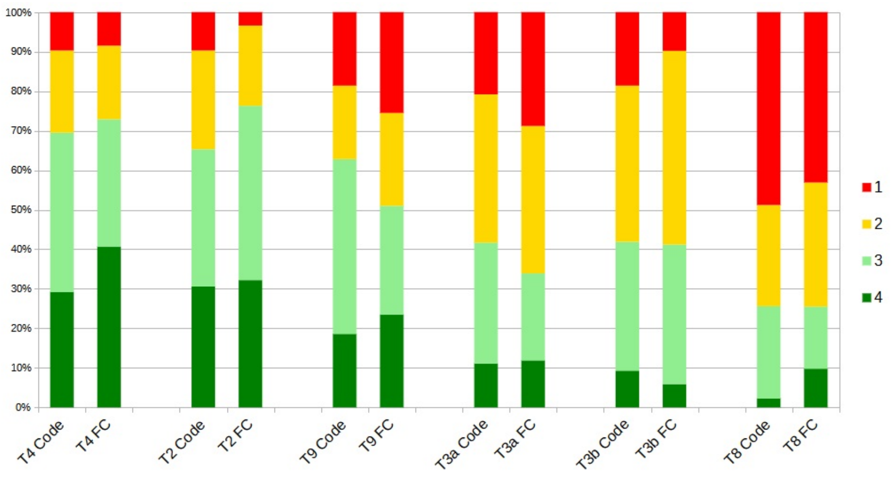


Figure 4.27: Students' self-confidence comparing code and flow-chart version, orderly decreasing by average.

also in this case the flow-chart better explains the workflow. Besides, the Boolean operations are written in English and not in Java-like style.

Differently, in T3b the code version highlights more self-confidence level than the flow-chart version, which, in the representation of the two nested control-flow statements, probably caused some confusion.

This result could be due to the importance of the mental imaging [VS11; Men+16] as a tool for understanding the notional machine, especially for students of STEM disciplines, as recent studies state that they have a predominantly spatial cognitive style [WLB09; Ati+20]. This outcome, however, contradicts what emerged in tasklet T3a. As previously mentioned, this result could be due to a criticality of the proposed problem, hence the unfamiliarity with handling Boolean values, even in the condition.

In addition, Pearson correlation coefficient test [CMM07] was computed to assess the relationship between performance levels [3] and self-confidence levels.

Table 4.38 shows Pearson correlation between students' performance and their self-confidence in survey 1 and survey 2 tasklets. While Table 4.39 refers to the correlation between performance and self-confidence in connection with the flow-chart version in survey 1, Table 4.40 refers to survey 2.

[3]To calculate the correlation, the three levels of performance were coded with the values 4 (correct), 2 (incorrect), 1 (seriously incorrect): the choice is aimed at aligning the performance levels with those of self-confidence, taking into account that there are no answers that can be classified as "quite correct" (3).

Table 4.38: Correlation between performance and self-confidence.

Version	**T1**	**T2/ii**	**T3a/ii**	**T4/ii**	**T7b/i**	**T7b/ii**	**T10/ii**	**Overall**
survey 1	0.185	0.173	0.318	0.213	0.201	0.239	0.313	0.235
	T3b	**T5/ii**	**T6**	**T7a/i**	**T7a/ii**	**T8/ii**	**T9/ii**	
survey 2	0.054	0.139	0.144	0.303	0.598	-0.094	-0.012	0.208

Table 4.39: Correlation between performance and self-confidence in connection with flow-chart version in survey 1.

Version	**T2/ii**	**T3a/ii**	**T4/ii**	**Overall**
	0.173	0.318	0.213	0.239
code	-0.016	0.497	0.187	0.230
flow-chart	0.385	0.112	0.241	0.241

Table 4.40: Correlation between performance and self-confidence in connection with flow-chart version in survey 2.

Version	**T3b**	**T8/ii**	**T9/ii**	**Overall**
	0.054	-0.094	-0.012	-0.011
code	0.089	0.082	-0.130	0.043
flow-chart	0.026	-0.196	0.078	-0.050

The data indicate a marginal correlation (overall just over 0.20), which in this case means marginal awareness of one's abilities, and the tables add a further element of analysis compared to Table 4.26. Hence students, on average, consider tasks more within their reach than they actually are, and they distinguish poorly between tasks at which they do better and tasks at which they do worse. Furthermore, awareness does not increase if the task is presented in the flow-chart version, as self-confidence is more related to the specific task than to the type of representation.

In summary, the following results can be identified.

Plan completion In this context, students are more aware of their knowledge and skills, they have in fact demonstrated that they have acquired the concept of notional machine for iteration and nesting. Besides, they know how to manage simple conditions and they use tracing effectively. Tasklets of this type are similar to those that their teachers present as examples or propose to them as exercises.

Functional purposes Despite the skills expressed in the previous point (notional machine, tracing, simple conditions and nesting management) students are not sure when they have to identify the functional purpose of a tasklet, and this may be due to a lack of mathematical prerequisites (especially in the cases of tasklets with numerical data) but also to the lack of habit of analyzing problems in this light.

Program equivalence There is no habit of working on the level of equivalences between algorithms or programs, and students seem to only think about the examples proposed by the teachers, which we have seen in the previous Chapter 3. However, when students encounter problems similar to topics already seen, or have to analyze equivalences between different types of loops (e.g. While-loop vs. Do-While-loop) they demonstrate a higher degree of self-confidence.

Reversibility Although not familiar with reversibility problems, the students demonstrate a certain confidence, especially when the problems are simpler and with numerical data.

4.6.5 Implications for instructors

The previous data analysis shows that the majority of students develop a viable and accurate enough mental model of the *notional machine* underlying code execution, including the functioning of nested constructs. However, difficulties are evident with complex nested constructs, or with *downward loop*. Furthermore, when composite conditions require the mastery of Boolean operators, difficulties have emerged. This result confirms, once again, the importance of mathematical prerequisites. Even with simple conditions, some confusion emerges when managing the last iteration.

Performances show that a large part of students are unable to master the relationships between loop condition and accurate specification in the application domain. This may possibly be ascribed to confusion about the role of the loop condition, meant as an ‘exit’ condition instead of a ‘continue’ condition, or to some more basic lack of problem-solving skills.

Moreover, students’ performance is better when relating to numerical data rather than non-numerical data, as they did not acquire sufficient mastery with arrays, but especially with strings.

Regarding Q2 question, the results show that students present tracing abilities, but they do not adequately exploit these abilities when confirming their hypotheses in questions that relate to more abstract levels. Thus, students perform better in simpler and tracing-based problems than when facing problems with greater difficulty and abstraction levels, such as questions regarding functional equivalence, reversibility, or identifying functional purpose. In effect, these types of problems, more abstract, are not often proposed in pre-tertiary education. From a pedagogical perspective, designing reversibility tasks could be a useful instrument aimed at assessing and fostering students’ mastery of basic program constructs. Reversibility tasks could help to focus on the need to test, and trace, program behavior carefully, it may provide opportunities to examine coding at different levels of abstraction and practice with varied success higher-order thinking skills [MIS20].

Concerning Q3 question, students' performance in tasklets with a code version and a flow-chart version are quite similar. The outcomes obtained do not clarify whether the use of the flow-chart facilitates understanding [RBH17], as it could also be that students, currently, have no mastery in reading the flow-chart. Probably, current teachers do not insist on flow-chart representation but prefer to present algorithms and problems directly with code, or they use the flow-chart representation for simple problems at the beginning of the course, and then prefer the code representation when the problems become more complex. It could be hypothesized that the spatial cognitive style and mental imaging [Pai13] work at a different level from that of the mere reading of a flow-chart, and it would be interesting to analyze this point with further investigations.

Furthermore, regarding Q4 question, generally, students have a higher level of self-confidence with respect to their own performance, thus students seem to underestimate their difficulties when dealing with loop conditions and nested loops.

While we have established that learning to program is a slow and gradual process, as argued by Dijkstra in [Dij89], and that the teacher must therefore grant adequate learning time to be spent on several effective examples, it would still be better if the examples proposed by the teachers were more varied and took more aspects into consideration. In addition, teachers could envision actions to affect students' awareness of potentials and limitations, eliciting a meta-cognitive approach [Cor95].

The results obtained lead us to formulate some suggestions for teachers.

- Grant adequate learning time.
- Spend time proposing examples of various kinds, which require a greater effort of abstraction (e.g. functional equivalence, reversibility, identify functional purpose).
- Propose problems with various types of patterns, to avoid student forming false beliefs or misconceptions, as well as ineffective stereotypes.
- Represent by flow-chart the proposed problems, to allow students with a spatial cognitive style to form their own mental imaging.
- Consider the importance of feedback, even frequent, so that the student understands their mistakes, misconceptions and ineffective stereotypes.
- Pay attention to alignment between the learning of mathematical and logical prerequisites.
- Evaluate interventions to increase students' meta-cognitive thinking.

Concerning more abstract tasks to propose to learners, we suggest considering our tasksets — consisting of tasklets with a balanced distribution over the areas and topics, both in their code and flow-chart versions — as starting point for planning lessons and exercises.

4.7 Concluding remarks

We have here presented an instrument to investigate in more depth high-school students' understanding of iteration in terms of code reading abilities. Each tasklets has been designed to verify, from various points of view, high school students' mastery of iteration constructs, where by mastery we mean *conceptual* mastery of code structures, focusing on program *comprehension* rather than construction. The original contributions of this chapter are:

1. define a comprehensive instrument to verify students' performance concerning several areas in connection with iteration;
2. verify students' performance and compare outcomes to pupils' subjective perception of self-confidence.

The designed tasklets can be used to investigate multiple features at the same time:

- the connection between loops and reflective abstraction;
- code features in connection with:
  - plain loop,
  - loop with nested conditional,
  - nested loops,
  - simple condition,
  - composite condition,
  - exit condition,
  - loop control variable,
  - downward for-loop;
- tracing skills, in particular output/final state and the number of iterations;
- whether mathematical skills influence the results;
- students' performance in code version and flow-chart version;
- students' subjective perception of self-confidence in more challenging tasklets.

The obtained results, in connection with those from the teacher survey discussed in Chapter 3, led us to suggest possible instructional interventions.

References

[Alz+18] Nabeel Alzahrani et al. "An analysis of common errors leading to excessive student struggle on homework problems in an introductory programming course". In: *2018 ASEE Annual Conference & Exposition*. 2018.

[Ati+20] Kinnari Atit et al. “Examining the role of spatial skills and mathematics motivation on middle school mathematics achievement”. In: *International Journal of STEM Education* 7.1 (2020), pp. 1–13.

[Ban00] Albert Bandura. *Autoefficacia: Teoria e applicazioni.(Presentazione all’edizione italiana di Gian Vittorio Caprara).* Edizioni Erickson, 2000.

[BFC82] John Biggs and Kevin F Collis. “Evaluating the Quality of Learning: the SOLO Taxonomy”. In: *SERBIULA (sistema Librum 2.0)* (Jan. 1982).

[CD17] Ibrahim Cetin and Ed Dubinsky. “Reflective abstraction in computational thinking”. eng. In: *The Journal of mathematical behavior* 47 (2017), pp. 70–80.

[Cet15] Ibrahim Cetin. “Student’s Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective”. In: *Canadian Journal of Science, Mathematics and Technology Education* 15.2 (Feb. 2015), pp. 155–170. DOI: 10.1080/14926156.2015.1014075.

[Cet+20] Ibrahim Cetin et al. “Teaching Loops Concept through Visualization Construction”. In: *Informatics in Education-An International Journal* 19.4 (2020), pp. 589–609.

[CHR13] Pavol Cerny, Thomas A Henzinger, and Arjun Radhakrishna. “Quantitative abstraction refinement”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 2013, pp. 115–128.

[CLM20] Umberto Costantini, Violetta Lonati, and Anna Morpurgo. “How Plans Occur in Novices’ Programs: A Method to Evaluate Program-Writing Skills”. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* SIGCSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 852–858. DOI: 10.1145/3328778.3366870.

[CMM07] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research methods in education.* London, New York: Routledge, 2007.

[Cor95] Cesare Cornoldi. *Metacognizione e apprendimento.* Strumenti. Psicologia. Il Mulino, 1995.

[CW15] Jacqui Chetty and Duan van der Westhuizen. “Towards a Pedagogical Design for Teaching Novice Programmers: Design-Based Research as an Empirical Determinant for Success”. In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research.* Koli Calling ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 5–12. DOI: 10.1145/2828959.2828976.

[DB86] Benedict Du Boulay. “Some Difficulties of Learning to Program”. In: *Journal of Educational Computing Research* 2 (Jan. 1986), pp. 57–73.

[Dij89] Edsger W. Dijkstra. “On the cruelty of really teaching computing science”. English. In: *Communications Of The Acm* 32.12 (1989), pp. 1398–1404.

[EHR20] Barbara Ericson, Beryl Hoffman, and Jennifer Rosato. “CSAwesome: AP CSA Curriculum and Professional Development (Practical Report)”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. WiPSCE ’20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421593.

[Fin+20] Sally Fincher et al. “Notional Machines in Computing Education: The Education of Attention”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 21–50. DOI: 10.1145/3437800.3439202.

[Fre+18] Stephen Frezza et al. “Modelling Competencies for Computing Education beyond 2020: A Research Based Approach to Defining Competencies in the Computing Disciplines”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 148–174.

[Ful+07] Ursula Fuller et al. “Developing a computer science-specific learning taxonomy”. eng. In: *ACM SIGCSE Bulletin* 39.4 (2007), pp. 152–170.

[GA06] David Ginat and Michal Armoni. “Reversing: An Essential Heuristic in Program and Proof Design”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’06. New York, NY, USA: ACM, 2006, pp. 469–473. DOI: 10.1145/1121341.1121488.

[Gin04] David Ginat. “On Novice Loop Boundaries and Range Conceptions”. In: *Computer Science Education* 14 (Sept. 2004), pp. 165–181. DOI: 10.1080/0899340042000302709.

[Gin18] David Ginat. “Algorithmic Cognition and Pencil-Paper Tasks”. In: *Olympiads in Informatics* 12 (May 2018), pp. 43–52. DOI: 10.15388/ioi.2018.04.

[GP18b] Shuchi Grover and Roy Pea. “Computational Thinking: A Competency Whose Time Has Come”. In: *Computer Science Education: Perspectives on teaching and learning in school*. Ed. by S. Sentance, E. Barendsen, and S. Carsten. London, UK: Bloomsbury Academic, 2018, pp. 19–38.

[Gro+17] Shuchi Grover et al. "A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments". In: *ACM Transactions on Computing Education (TOCE)* 17.3 (2017), pp. 1–25.

[Haz99] Orit Hazzan. "Reducing Abstraction Level When Learning Abstract Algebra Concepts". In: *Educational Studies in Mathematics* 40.1 (1999), pp. 71–90.

[HH82] Brian C. Hansford and John A. Hattie. "The relationship between self and achievement/performance measures". In: *Review of Educational Research* 52.1 (1982), pp. 123–142.

[HK99] Mary Hegarty and Maria Kozhevnikov. "Types of visual–spatial representations and mathematical problem solving." In: *Journal of educational psychology* 91.4 (1999), p. 684.

[IM20] Cruz Izu and Claudio Mirolo. "Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers". In: *Proc. of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE '20. New York, NY, USA: ACM, 2020, pp. 466–472.

[IMW18] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. "Novice Programmers' Reasoning About Reversing Conditional Statements". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE '18. New York, NY, USA: ACM, 2018, pp. 646–651. DOI: 10.1145/3159450.3159499.

[IPW17] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. "On the Ability to Reason About Program Behaviour: A Think-Aloud Study". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE '17. New York, USA: ACM, 2017, pp. 305–310. DOI: 10.1145/3059009.3059036.

[IPW19] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. "Up or Down? An Insight into Programmer's Acquisition of Iteration Skills". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* SIGCSE '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 941–947. DOI: 10.1145/3287324.3287350.

[IWP16] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. "A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research.* ICER '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 251–259. DOI: 10.1145/2960310.2960324.

[Izu+19] Cruz Izu et al. “Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories”. In: *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’19. New York, NY, USA: ACM, 2019, pp. 27–52.

[KD03] Amruth Kumar and Garrett Dancik. “A tutor for counter-controlled loop concepts and its evaluation”. In: *33rd Annual Frontiers in Education, 2003. FIE 2003.* Vol. 1. Nov. 2003, T3C–7. DOI: `10.1109/FIE.2003.1263331`.

[KD10] Herman Koppelman and Betsy van Dijk. “Teaching Abstraction in Introductory Courses”. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 174–178. DOI: `10.1145/1822090.1822140`.

[Kes19] Max Kesselbacher. “Supporting the Acquisition of Programming Skills with Program Construction Patterns”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 188–189. DOI: `10.1109/ICSE-Companion.2019.00077`.

[LDC20] Elise Lockwood and Adaline De Chenne. “Enriching Students’ Combinatorial Reasoning through the Use of Loops and Conditional Statements in Python”. In: *International Journal of Research in Undergraduate Mathematics Education* 6 (Oct. 2020). DOI: `10.1007/s40753-019-00108-2`.

[Lis+04] Raymond Lister et al. “A Multi-national Study of Reading and Tracing Skills in Novice Programmers”. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’04. New York, NY, USA: ACM, 2004, pp. 119–150. DOI: `10.1145/1044550.1041673`.

[Lis+06] Raymond Lister et al. “Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy”. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE ’06. New York, NY, USA: ACM, 2006, pp. 118–122.

[Lis11] Raymond Lister. “Concrete and other neo-piagetian forms of reasoning in the novice programmer”. In: *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), pp. 9–18.

[Lop+08] Mike Lopez et al. “Relationships Between Reading, Tracing and Writing Skills in Introductory Programming”. In: *Proc. 4th Int. Workshop on Comput. Educ. Research.* ICER ’08. New York, USA: ACM, 2008, pp. 101–112.

[LR+18] Andrew Luxton-Reilly et al. “Introductory Programming: A Systematic Literature Review”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 55–106.

[Man+20] Linda Mannila et al. “Programming in Primary Education: Towards a Research Based Assessment Framework”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education.* WiPSCE ’20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421598.

[MBŽ18] Monika Mladenovic, Ivica Boljat, and Žana Žanko. “Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level”. In: *Education and Information Technologies* 23 (July 2018), pp. 1483–1500. DOI: 10.1007/s10639-017-9673-3.

[Men+16] Chiara Meneghetti et al. “The role of visual and spatial working memory in forming mental models derived from survey and route descriptions”. In: *British journal of psychology (London, England : 1953)* 108 (Mar. 2016). DOI: 10.1111/bjop.12193.

[MHD09] Shuhaida Mohamed Shuhidan, Margaret Hamilton, and Daryl D’Souza. “A Taxonomic Study of Novice Programming Summative Assessment”. In: *Proc. 11th Australasian Conf. on Computing Education - Volume 95.* ACE ’09. Darlinghurst, Australia: Australian Computer Society, Inc., 2009, pp. 147–156.

[MI19] Claudio Mirolo and Cruz Izu. “An Exploration of Novice Programmers’ Comprehension of Conditionals in Imperative and Functional Programming”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 436–442. DOI: 10.1145/3304221.3319746.

[Mir12] Claudio Mirolo. “Is Iteration Really Easier to Learn Than Recursion for CS1 Students?” In: *Proc. of the 9th Annual International Conference on International Computing Education Research.* ICER ’12. New York, NY, USA: ACM, 2012, pp. 99–104.

[MIS20] Claudio Mirolo, Cruz Izu, and Emanuele Scapin. "High-School Students' Mastery of Basic Flow-Control Constructs through the Lens of Reversibility". In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. WiPSCE '20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421603.

[MS21] Craig S. Miller and Amber Settle. "Mixing and Matching Loop Strategies: By Value or By Index?" In: *Proc. of the 52nd SIGCSE*. SIGCSE '21. Virtual Event, USA, 2021, pp. 1048–1054.

[MS68] H. Edward Massengill and Emir H. Shuford. *The effect of 'Degree of Confidence' in student testing*. eng. Tech. rep. 1968.

[Pai13] Allan Paivio. *Imagery and Verbal Processes*. English. OCLC: 869091762. Hoboken: Taylor and Francis, 2013.

[Per+86] David N. Perkins et al. "Conditions of learning in novice programmers". In: *Journal of Educational Computing Research* 2.1 (1986), pp. 37–55.

[PRW07] Anne Philpott, Phil Robbins, and J Whalley. "Assessing the steps on the road to relational thinking". In: *Proceedings of the 20th annual conference of the National Advisory Committee on Computing Qualifications*. Vol. 286. 2007.

[RAVD83] H. Rudy Ramsey, Michael E. Atwood, and James R. Van Doren. "Flowcharts versus Program Design Languages: An Experimental Comparison". In: *Commun. ACM* 26.6 (June 1983), pp. 445–449. DOI: 10.1145/358141.358149.

[RBH17] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. "Identifying Students' Misconceptions on Basic Algorithmic Concepts Through Flowchart Analysis". In: *Informatics in Schools: Focus on Learning Programming*. Ed. by Valentina Dagienė and Arto Hellas. Cham: Springer International Publishing, 2017, pp. 155–168.

[RDLR20] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. "A Miss is as Good as a Mile: Off-By-One Errors and Arrays in an Introductory Programming Course". In: *Proc. of the 22nd Australasian Computing Education Conference*. 2020, pp. 31–38.

[SM19] Emanuele Scapin and Claudio Mirolo. "An Exploration of Teachers' Perspective About the Learning of Iteration-Control Constructs". In: *Informatics in Schools. New Ideas in School Informatics*. Ed. by Sergei N. Pozdniakov and Valentina Dagienė. Cham: Springer, 2019, pp. 15–27.

[SM20] Emanuele Scapin and Claudio Mirolo. “An Exploratory Study of Students’ Mastery of Iteration in the High School”. In: *Proceedings of the International Conference on Informatics in School: Situation, Evaluation and Perspectives, Tallinn, Estonia, November 16-18, 2020.* Ed. by Külli Kori and Mart Laanpere. Vol. 2755. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 43–54.

[SPV20] Phil Steinhorst, Andrew Petersen, and Jan Vahrenhold. “Revisiting Self-Efficacy in Introductory Programming”. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research.* 2020, pp. 158–169.

[SS88] Elliot M. Soloway and James C. Spohrer. *Studying the Novice Programmer.* USA: L. Erlbaum Associates Inc., 1988.

[SWS17] Renske Smetsers-Weeda and Sjaak Smetsers. “Problem Solving and Algorithmic Development with Flowcharts”. In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education.* WiPSCE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 25–34. DOI: 10.1145/3137065.3137080.

[TG10] Allison Tew and Mark Guzdial. “Developing a validated assessment of fundamental CS1 concepts”. In: Jan. 2010, pp. 97–101. DOI: 10.1145/1734263.1734297.

[Tho+06] Errol Thompson et al. “Code Classification as a Learning and Assessment Exercise for Novice Programmers”. English. In: *The 19th Annual Conference of the National Advisory Committee on Computing Qualifications.* Ed. by Samuel Mann and Noel Bridgeman. National Advisory Committee on Computing Qualifications, 2006, pp. 291–298.

[TL14a] Donna Teague and Raymond Lister. “Blinded by their Plight: Tracing and the Preoperational Programmer”. In: *PPIG.* June 2014.

[TL14b] Donna Teague and Raymond Lister. “Programming: Reading, Writing and Reversing”. In: *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’14. New York, USA: ACM, 2014, pp. 285–290. DOI: 10.1145/2591708.2591712.

[TL14c] Donna Teague and Raymond Lister. “Programming: Reading, Writing and Reversing”. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education.* ITiCSE ’14. New York, NY, USA: ACM, 2014, pp. 285–290. DOI: 10.1145/2591708.2591712.

[VS11] André Vandierendonck and Arnaud Szmalec. *Spatial working memory.* Jan. 2011.

[VTL09] Anne Venables, Grace Tan, and Raymond Lister. "A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer". In: *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*. ICER '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–128. DOI: 10.1145/1584322.1584336.

[WLB09] Jonathan Wai, David Lubinski, and Camilla P. Benbow. "Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance." en. In: *Journal of Educational Psychology* 101.4 (2009), pp. 817–835. DOI: 10.1037/a0016127.

[YK07] Gavriel Yarmish and Danny Kopec. "Revisiting Novice Programmer Errors". In: *SIGCSE Bull.* 39.2 (June 2007), pp. 131–137. DOI: 10.1145/1272848.1272896.

[Zim00] Barry J. Zimmerman. "Self-efficacy: An essential motive to learn". In: *Contemporary educational psychology* 25.1 (2000), pp. 82–91.

Chapter 5

Conclusions

This chapter concludes the thesis. Section 5.1 summarizes the overall work and the major insights gained, while Section 5.2 mentions some implications for the Italian high school curricula. Finally, in Section 5.3 we present possible directions for future developments.

5.1 Work summary and major insights

First of all, we investigated the learning of iteration, starting from interviews of experienced upper secondary teachers of introductory programming in different kinds of schools, which aimed at ascertaining their beliefs about major sources of difficulties with basic programming concepts, as well as their approach to the teaching of iteration constructs (Chapter 2, Section 2.1). The interviews have established that iteration is among the few most central concepts for novice programmers, a statement with which all the interviewed teachers appear to agree.

According to the teachers, a number of difficulties could be ascribed to lack of prerequisites, in particular those implied in the mathematical and text comprehension skills. Problems with the former often manifest in the treatment of loop conditions. A possible explanation of students' difficulties with the application of mathematical and logical concepts is that the subject matter of Informatics (Computer Science) and Mathematics are not well integrated in the standard high school curriculum — see the next section for more details. In other words, some of the mathematical topics may be covered either too early or too late to be effective when they are required to learn programming.

Problems related to text comprehension, on the other hand, are also found in the study of natural languages. As documented in the literature, the differences between natural and artificial languages can generate difficulties and misconceptions. Unlike natural languages, programming languages have a rigorous syntax and a well-defined formal syntax. A programming language must be compiled and executed by a machine, so it cannot contain ambiguities. If, for example, we consider the taxonomy proposed by Meneghetti [Men+20], several characteristics of a natural language cannot be found in a programming language, but only in the text of the problem, which, however, is usually expressed in a form other than a narrative text.

As a further insight emerging from the interviews, in the teachers' opinion a possible way to induce students to develop their abstraction skills is to contrast their tendency to approach a task by trial-and-error and require them to analyze the problem at hand with paper and pencil first. Another possibility is to instruct the students to organize their programs into several functions and procedures aimed at introducing meaningful levels of abstraction.

Our next task was to administer a survey addressed to students, which included both questions on their subjective perception of difficulty and a few small tasks probing their understanding of iteration (Chapter 2, Section 2.2). In order to develop this exploratory instrument we also took into account the insights gained from the teachers' interviews.

The respondents identified as major sources of difficulty, in their perception, nested iterations and complex expressions including logical operators. Based on performance in the small tasks, however, the students seem to underestimate their difficulties to deal even with simple loop conditions. A large part of them were indeed unable to master the relationships between loop condition and accurate specification in the application domain in a straightforward situation. This may partly be ascribed to confusion about the role of the loop condition, meant as an 'exit' condition instead of a 'continue' condition, but some more basic lack of problem-solving skills is also conceivable in a number of cases. This result is perhaps not too surprising, in light of the fact that the common examples presented by teachers to illustrate iteration are often straightforward and tend to reproduce stereotypical patterns.

Apparently, students' difficulties with nested constructs are not so much about the mechanics of code execution but they relate more to the ability of grasping code behavior at a more abstract level. Most students seem to have developed a viable mental model of the basic workings of the underlying machine, hence the crucial point is how to develop their abstraction skills. Furthermore, students tend to not exploit their tracing abilities in order to test their conjectures about program behavior. This observation could be explained either by a general idle attitude or, which is more relevant from a pedagogical perspective, by a lack of method when approaching programming tasks.

To sum up, teachers and students agree that iteration is a key programming concept and that the treatment of conditions and nested constructs are major sources of students' difficulties with iteration, the former also confirming the weakness of mathematical prerequisites.

In order to support the development and assessment of students' skills, we have built a preliminary catalog collecting program comprehension (code reading) tasks which require to trace, explain, or evaluate small programs (Appendix B). The programs are presented in code (C/C ++ or Java, currently the most widespread options in the Italian schools) as well as flow-chart versions.

Our attempt was to propose a selection of examples with a more varied and richer structure than the usual ones we normally find in textbooks. In particular, the program samples use non-trivial loop conditions and nested combinations of flow-control constructs. The catalog is meant to inspire teachers when designing problems and exercises. To analyze the cognitive challenge presented by each program comprehension task, they may refer to Schulte's *Block Model* framework [Sch08], in particular by following the suggestions of Izu et al. [Izu+19].

Based on the outcome of the previous steps, namely the literature review and the interviews, we then designed an online survey to collect further information about teachers' perspectives and good practices (Chapter 3). As major guiding factors underlying our analysis we considered the instructor's orientation towards more conceptual versus more practical objectives, on the one hand, and towards process-based versus product-based assessment, on the other. As a result of this approach to the analysis, teachers seem to give prominence to practical objectives, rather than conceptual ones, but also tend to focus their assessment more on processes than on products. Furthermore, once again, text comprehension and basic mathematical skills are identified as important prerequisites. In introductory courses, students are probably required to deal with topics they do not yet master, so that they find themselves working with languages and formalisms different from those they have been accustomed to.

As a last step, by elaborating on the insights gained from the pilot studies, we designed, developed and administered a new online survey addressed to high school students, specifically focused on their understanding of iteration (Chapter 4). This instrument is built upon the catalog of programming tasks, or *tasklets*, listed in Appendix B. Besides, covering aspects such as specific technical program features, correlation between tracing effort and abstraction, role of flow-chart vs. textual code, the survey asked for students' perception of self-confidence when achieving tasks where they are supposed to engage high-level thinking skills.

The data collected appear to confirm that the majority of students develop a viable and accurate enough mental model of the *notional machine* underlying code execution, including the operational treatment of composite Boolean conditions and the functioning of nested constructs. Nevertheless, students face difficulties when required to abstract on nested constructs or downcounting loops. We could also observe that, although students are at ease with tracing code execution, they do not adequately exploit this skill in order to test their conjectures about program behavior at a higher level of abstraction, such as that required to evaluate functional equivalence, to answer reversibility questions, or to experiment on a program's functional purpose. In addition, a large part of students are unable to master the relationships between loop condition and accurate specification in the problem domain.

As to the implications of textual code versus flow-charts, students' performance appears to be essentially unaffected. We can then speculate that a spatial cognitive style (mental imaging, according to [Pai13]) is not restricted to basing the analysis on a flow-chart. Finally, from the subjective assessment of self-confidence, students seem to underestimate their difficulties to deal with loop conditions and nested loops.

5.2 Curricular implications

Teachers, both during the interviews and through the online survey, pointed out the importance of mathematical prerequisites and the disalignment between Informatics and mathematical topics. This issue emerges from the analysis of the curricula of a number of upper secondary schools[1], plans that are closely based on the ministerial guidelines[2]. In other words, Informatics and Mathematics are not well integrated in the standard high-school curriculum, with some of the mathematical topics being covered either too early or too late to be effective when students are required to learn programming. For instance, as we can see in Table 5.1, in the "Mathematics" subject matter Boolean algebra and propositional calculus are introduced too early, whereas vectors and matrices come too late.

Table 5.1: Schedule of Informatics and Mathematics topics in technical high schools.

Year	**Computer Science**	**Mathematics**
2	assignment, conditional	set theory, Boolean algebra, propositional calculus
3	conditional, iterations, nested flow control, nested iterations, arrays, strings, two-dimensional arrays	algebraic equations and inequalities, goniometry, trigonometry, complex numbers, exponential and logarithmic functions, analytic geometry
4	abstract data type, OOP	limits, functions, derivatives, linear algebra: vector and matrix

Hence, in this respect we would recommend to exploit the opportunities of cross-disciplinary collaboration between Mathematics and Informatics teachers, as a means to achieve a better alignment of the interconnected topics.

[1]Disciplinary plans concerning Mathematics and Computer Science are available online, we have consulted ITT Chilesotti (`https://www.chilesotti.edu.it/`), ITIS Fermi (`https://www.fermibassano.edu.it/`) and IIS Marzotto-Luzzati (`https://www.iisvaldagno.it/`) websites, technical upper secondary schools.

[2]A concise and schematic view of the ministerial guidelines is available at the link: `https://www.zanichelli.it/scuola/piani-di-studio-scuola-secondaria-di-ii-grado`

5.3 Future directions of work

As remarked in the introduction, this work can also lay the basis for further developments which we think are worth considering.

To begin with, some natural extensions of our endeavor include broadening the sample of students engaged in the (second) survey and investigating the extent to which the catalog tasks can be helpful to foster development of abstract thinking skills. With regards to the former, it would be especially insightful to address a larger number of girls in order to analyze possible gender issues [McK00; SGK20].

It could be also interesting to explore the effectiveness of methodological tools inspired by the notion of loop invariant, see e.g. the pedagogical work in [Tam92; Gin03], suitably adapted to fit less formal learning styles [Ast91]. It was actually one of the objectives of our original project, but unfortunately, as mentioned above, we were not able to plan interventions in the schools to investigate this approach because of the Covid-19 pandemic.

Another interesting topic is related to the potentials of using flow-charts when reasoning about programs in connection with spatial abilities. Spatial activities are meant as activities that involve reasoning about qualities of space (e.g., distance, proportion), practicing mental visualization [Pet+20] (e.g., figuring out spatial layouts or spatial trajectories). In this respect, several works examined the role of spatial skills in Mathematics. In social cognitive theory (see Bandura [Ban05]) it is suggested that they are relevant to good performance in Mathematics. Hegarty & Kozhevnikov [HK99] pointed out that the "use of schematic representations is positively related to success in mathematical problem solving", and according to Atit at al. [Ati+20], "the ability to mentally visualize and manipulate images is a cognitive skill critical for success in Mathematics". Moreover, Wai et al. [WLB09] investigated the importance of spatial ability for in STEM domains, i.e. in Science, Technology, Engineering and Mathematics. Here are some possible lines of investigation:

- To which extent may spatial cognitive styles of learning be beneficial to novice programmers?
- Do flow-charts play a prominent role in this respect or does mental imaging arise independently of the use of this form of program representation?
- More in general, what are the implications of different cognitive styles when learning programming and, more specifically, iteration?

In light of the weak correlation between performance and self-confidence while trying to achieve a task (see Chapter 4), a more ambitious project would be to explore the impact of teaching meta-cognitive skills explicitly. According to Schraw [Sch98], indeed, meta-cognitive awareness *can* be taught. Meta-cognitive knowledge can be defined as *"the subject's awareness of his own cognitive processes"* [Cot06], or the ideas that a person has developed about mental functioning, which include: impressions, intuitions, notions, feelings, and self-perception [Cor95].

By elaborating on Flavell's distinction [Fla92] between declarative and procedural knowledge, Schraw [Sch98] identifies three types of meta-cognitive awareness, regarding:

– *declarative knowledge*, expressing knowledge as to the way of learning and how this can be affected;
– *procedural knowledge*, expressing knowledge about strategies that can be used to perform certain tasks;
– *conditional knowledge*, concerning the "when" and the "why" when using declarative and procedural knowledge.

In general, meta-cognitive knowledge is achieved gradually, starting from primary school, and becomes effective in the last years of high school. For the learning of Computer Science, where students are currently involved only in later school stages, the development of meta-cognitive awareness is not as settled as in other fields, such as mother language literacy or Mathematics. Therefore, students do not master the implied meta-cognitive control processes, namely, integrating and planning the use of previous knowledge, anticipating times and results of individual study, choosing effective study methods, monitoring and verifying *in itinere* the learning achievements [Bro87] — what is referred to as self-management of cognitive activity [PCL84].

In this framework, a stimulating research direction stems from the idea that the problem-solving activities implied by the tasks of our catalog can also be exploited as means of eliciting students' own cognitive functioning. A possible instructional strategy, rarely adopted in computing education, although widely considered in the studies of learning [Ian04], is to have students verbalize the mental operations they carry out while solving the problem at hand. Meanwhile, novice programmers should be encouraged to keep asking themselves questions about the problem at hand and the objectives of their work (see, in particular, the discussion of explicit strategies presented by De Raadt [DR08; DRWT09b]). Other standard guidelines to develop students' meta-cognitive skills include frequent *feedback* from teachers on their mistakes and on the quality of their work [Hat12], as well as employing teamwork and prompting feedback from peers while engaging in laboratory tasks [HLR11].

Appendices

Appendix A

Publications

A.1 List of papers

Part of this work is based on the following peer-reviewed publications.

1. Scapin Emanuele, Mirolo Claudio. (2019). An Exploration of Teachers' Perspective About the Learning of Iteration-Control Constructs. In: Pozdniakov S., Dagienė V. (eds) Informatics in Schools. New Ideas in School Informatics. ISSEP 2019. Lecture Notes in Computer Science, vol 11913. Springer, Cham. https://doi.org/10.1007/978-3-030-33759-9_2

2. Mirolo Claudio, Izu Cruz, and Scapin Emanuele. (2020). High-school students' mastery of basic flow-control constructs through the lens of reversibility. In Proceedings of the 15th Workshop on Primary and Secondary Computing Education (WiPSCE '20). Association for Computing Machinery, New York, NY, USA, Article 15, p. 1–10. DOI: https://doi.org/10.1145/3421590.3421603

3. Scapin Emanuele, Mirolo Claudio. (2020). An Exploratory Study of Students' Mastery of Iteration in the High School. In K. Kori & M. Laanpere (Eds.), Local Proceedings of ISSEP 2020 – 13th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives. Tallinn, Estonia: University of Tallinn. (CEURWS Volume). p. 43–54.

4. Claudio Mirolo, Cruz Izu, Violetta Lonati, Emanuele Scapin. (2021). Abstraction in Computer Science Education: An Overview, Informatics in Education 20, no. 4, p. 615–639, DOI 10.15388/infedu.2021.27

5. Scapin Emanuele, Mirolo Claudio (2020). An Investigation of High School Students' difficulties with Iteration-Control Constructs, Mondo Digitale 89, p. 1–11, http://mondodigitale.aicanet.net

6. Scapin Emanuele, Mirolo Claudio (2021). Design and development of an instrument to investigate high-school students' understanding of iteration. In

E. Barendsen & C. Chytas (Eds.), Local Proceedings of ISSEP 2021 – 14th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives. Nijmegen, The Netherlands: Radboud University.

A.2 Abstracts

Abstracts of the documents listed above.

1. **An Exploration of Teachers' Perspective About the Learning of Iteration-Control Constructs**
A number of studies report about students' difficulties with basic flow-control constructs, and specifically with iteration. Although such issues are less explored in the context of pre-tertiary education, this seems to be especially the case for high-school programming learning, where the difficulties concern both the "mechanical" features of the notional machine as well as the logical aspects connected with the constructs, ranging from the implications of loop conditions to a more abstract grasp of the underlying algorithms. As part of a project whose long-term goal is to identify methodological tools to improve the learning of iteration constructs, we interviewed 20 experienced upper secondary teachers of introductory programming in different kinds of schools from a large area in the North-East of Italy. In addition, a sample of 164 students from the same schools answered a survey which included both questions on their subjective perception of difficulty and simple tasks probing their understanding of iteration. The interviews were mainly aimed at ascertaining teachers' beliefs about major sources of issues for basic programming concepts and their approach to the teaching and learning of iteration constructs. Each interview was conducted according to a grid of 20 questions, informed by related frameworks to characterize teachers' pedagogical content knowledge and to design concept inventories. In essence, data from teachers and students confirm that iteration is a central programming concept and indicate that the treatment of conditions and nested constructs are major sources of students' difficulties with iteration.

2. **High-school students' mastery of basic flow-control constructs through the lens of reversibility**
High-school students specializing in computing fields need to develop the abstraction skills required to understand and create programs. Novices' difficulties at high-school level, ranging from mastery of the "notional machine" to recognition of a program's purpose, have not been investigated as extensively as at tertiary level. This work explores high-school students' code comprehension by asking to reason about reversing conditional and iteration constructs. 205 K11–13 students from different institutions were asked to engage in a set of "reversibility tasklets". For each code fragment, they needed to identify if its computation was reversible and either provide the code to reverse or an example of a value that cannot be reversed. For 4 such items, after extracting the recurrent patterns in students' answers, we carried out an analysis within the framework of the SOLO taxonomy. Overall, 74% of answers correctly identified whether the code was reversible but only 42 could provide the full

explanation/code. The rate of relational answers varied from 51% down to 21%, the poorest performance arising for a small array-processing loop (and although 65% of the subjects had correctly identified the loop as reversible). The instruction level did not have a strong impact on performance, indicating such tasks are suitable for K11, when the basic flow-control constructs are usually introduced. In particular, the reversibility concept could be a useful pedagogical instrument both to assess and to help develop students' program comprehension.

3. **An Exploratory Study of Students' Mastery of Iteration in the High School**
   Although a number of studies report about novices' difficulties with basic flow-control constructs, concerning both the understanding of the underlying notional machine and the logical connections with the application domain, these issues have not yet been extensively explored in the context of high-school education. As part of a project whose long-term goal is identifying methodological tools to improve the learning of iteration, we analyzed how a sample of 164 high-school students approached three small programming tasks involving basic looping constructs, as well as two questions on their subjective perception of areas of difficulty. If, on the one hand, most students seem to have developed a viable mental model of the basic workings of the underlying machine, on the other hand, dealing at a more abstract level with loop conditions and nested flow-control structures appears to be challenging. As to the implications for teachers, the results of the analysis suggest that more efforts should be put into developing a method for testing the conjectures about program behavior, as well as into the treatment of loop conditions in connection with the problem statement.

4. **Abstraction in Computer Science Education: An Overview**
   When we "think like a computer scientist," we are able to systematically solve problems in different fields, create software applications that meet various needs, and design artifacts that model complex systems. Abstraction is a soft skill embedded in all those endeavors, being a main cornerstone of computational thinking. Our overview of abstraction is intended to be not so much systematic as thought provoking, inviting the reader to (re)think abstraction from different – and perhaps unusual perspectives. After presenting a range of its characterizations, we will explore abstraction from a cognitive point of view. Then we will discuss the role of abstraction in a range of Computer Science areas, including whether and how abstraction is taught. Although it is impossible to capture the essence of abstraction in one sentence, one section or a single paper, we hope our insights into abstraction may help Computer Science educators to better understand, model and even dare to teach abstraction skills.

5. **An Investigation of High School Students' difficulties with Iteration-Control Constructs**
   A number of studies report about students' difficulties with basic flow-control constructs, and specifically with iteration. As part of a project whose long-term goal is to identify methodological tools to improve the learning of iteration constructs, we analyzed the answers of a sample of 164 high school students to three small programming tasks and two questions on their perception of difficulty. The results of the analysis suggest that more teaching efforts should go towards the development of a method to approach programming tasks and, more specifically for iteration, to the treatment of loop conditions in connection with the specifications in the application domain.

6. **Design and development of an instrument to investigate high-school students' understanding of iteration**
   Loops and conditionals turn out to be potential sources of novices' misconceptions. In this respect, our purpose is to outline here the design and development of an instrument, composed of a set of small tasks, or tasklets, to investigate in more depth high-school students' understanding of iteration in terms of code reading abilities. In particular, we will try to summarize the motivations underlying the choice of the tasks and the overall structure of the instrument. A major aim of this contribution is indeed to invite the interested educators and researchers to take part in the project in order to broaden the scope of the empirical study.

Appendix B

Catalog

The activities presented below, organized by main topic of analysis, are a possible catalog of problems and algorithms to be presented to students in the context of teaching iterations in introductory programming courses.

The main topic indicated does not exclude the possibility of using the code or the algorithm to analyze other areas related to iterations, and more.

Each tasklet can be used to analyze some students' skills, such as reading, tracing, explaining and evaluating.

Examples and suggestions for the tasklets proposed in the students' survey are taken from this catalog, see Chapter 4, Section 4.4, or via the link: `http://nid.dimi.uniud.it/additional_material/iteration_survey.html`

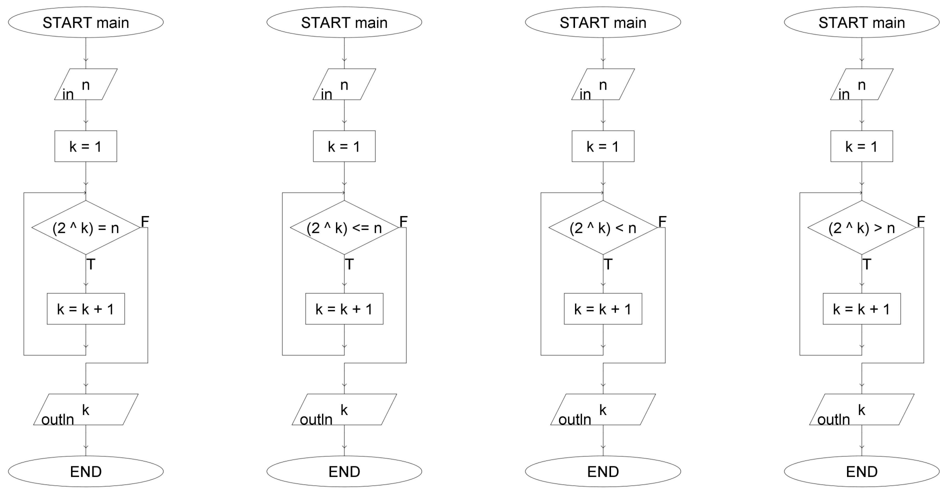
The cataloged tasklets are proposed with code or flow-chart version, sometimes with both representations. For coding, the languages C/C ++ or Java were used, currently the most taught languages in Italian upper secondary schools.

B.1 Loop condition

B.1.1 Tasklet L1

Problem statement: *Determine the number of bits needed to represent a positive integer n in base two, number of bits given by the smallest exponent k such that 2^k is greater than n. Which of the proposed algorithms seems correct to you?*

Loop condition options: $2^k = n$, $2^k \leq n$, $2^k < n$, $2^k > n$.



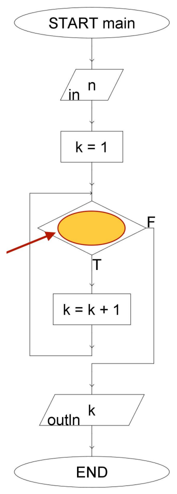
option 1 option 2 option 3 option 4

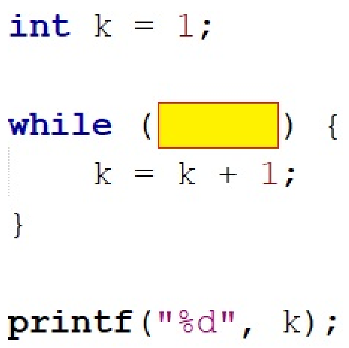
Figure B.1: *Which option has the correct loop condition?*

Other topics that can be explored: notional machine, tracing.

B.1.2 Tasklet L2

Problem statement: *Determine the number of bits necessary to represent a positive integer n in base two, number of bits given by the smallest exponent k such that 2^k is greater than n. What condition do you think is correct for the While Loop?*
Loop condition options: $2^k = n$, $2^k \leq n$, $2^k < n$, $2^k > n$.





```
int k = 1;

while (        ) {
    k = k + 1;
}


printf("%d", k);
```

flow-chart version code version

Figure B.2: *Which option has the correct loop condition?*

Other topics that can be explored: notional machine, tracing.

B.1.3 Tasklet L3

Problem statement: *Consider the following program for a 4-character string w = "hello", could you tell how many loops it does?*
Number of iterations options: 0, 3, 4, 5, more the 5, the loop never ends.

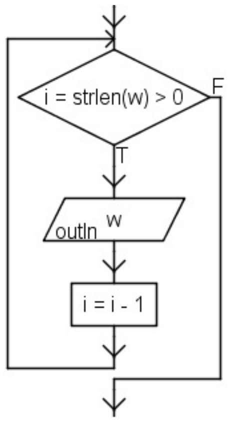


Figure B.3: Flow-chart version.

```c
int i;

while (i = strlen(w) > 0) {
  printf("%s\n", w);
  i--;
}

```

Listing B.1: Code version.

Other topics that can be explored: notional machine, tracing, loop control variable, not numerical values.

B.2 Loop complex condition

B.2.1 Tasklet LC1

Problem statement: *Consider the following program to check if two positive integers m, n are prime to each other. If the input values are $m = 15$ and $n = 44$, how many while loops will be performed?*
Numbers of iterations options: 0, 1, 2, 3, 4 or more, the loop never ends.

```c
int x = m, y = n;

while (x > 1 && y > 1 && x != y) {
  if (x < y)
    y = y - x;
  else
    x = x - y;
}

if (x == 1 || y == 1)
  printf("m=%d e n=%d are coprime to each other", m,n);
else
  printf("m=%d e n=%d are not coprime to each other", m,n);
```

Listing B.2: Loop complex condition C/C++ example.

Other topics that can be explored: notional machine, tracing, nested flow control constructs.

B.2.2 Tasklet LC2

Problem statement: *Consider the following program for calculating the square root of an integer n. If the input value is $n = 16$, how many while loops will be performed?*
Numbers of iterations options: 0, 3, 4, 5, more than 5, the loop never ends.

```c
float y = 0, z = 1;
float x = n; //n is the input value

while (z > 0.1 && n > 0) {
  y = 0.5 * (x + n / x);
  z = x - y;
  x = y;
}

printf("%f square root is approximately: %f\n", n,y);
```

Listing B.3: Numbers of iterations with loop complex condition, C/C++ example.

Other topics that can be explored: notional machine, tracing.

B.2.3 Tasklet LC3

Problem statement: *Consider the following program, can you tell what it can be used for?*
Functional purpose options:

(a) Calculate the nth element of the Fibonacci series;

(b) Calculate the average value of the numbers between 1 and n;

(c) Calculate the square root of n;

(d) It doesn't calculate anything in particular.

```c
float y = 0, z = 1;
float x = n; //n is the input value

while (z > 0.1 && n > 0) {
  y = 0.5 * (x + n / x);
  z = x - y;
  x = y;
}

printf("%f square root is approximately: %f\n", n,y);
```

Listing B.4: Functional purpose and loop complex condition, C/C++ example.

Other topics that can be explored: notional machine, tracing, functional purpose.

B.2.4 Tasklet LC4

Problem statement: *Consider the following program, if the input value $n = 25$, how many while loops will be done?*
Numbers of iterations options: 0, 3, 4, 5, 6, the loop never ends.
Functional purpose options:

(a) Calculate a divisor of the number;

(b) Calculate the number of digits that make it up;

(c) Calculate the square root;

(d) It doesn't calculate anything in particular.

```c
int z = 0;

while ((n > 0) && ((z+1)*(z+1) <= n)) {
  z = z + 1;
}

printf("%d\n", z);
```

Listing B.5: Numbers of iterations with loop complex condition, C/C++ example.

Other topics that can be explored: notional machine, tracing, functional purpose.

B.2.5 Tasklet LC5

Problem statement: *What are the values of $var1$ and $var2$ at the end of the code execution?*
Output state options:

(a) $var1 = 1, var2 = 1$;

(b) $var1 = 2, var2 = 0$;

(c) $var1 = 3, var2 = -1$;

(d) $var1 = 0, var2 = 2$;

(e) The loop will cause an error due to division by zero.

```c
int var1 = 0;
int var2 = 2;

while (var2 != 0) && ((var1 / var2) >= 0) {
  var1 = var1 + 1;
  var2 = var2 - 1;
}
```

Listing B.6: Numbers of iterations with loop complex condition, C/C++ example.

Other topics that can be explored: notional machine, tracing.

B.2.6 Tasklet LC6

Problem statement: *What condition should the While have in order to get* $Var1 = 2$ *and* $Var2 = 0$*?*
Loop condition options:

(a) $(var2 \mathrel{!=} 0)\ \&\&\ ((var1/var2) > 0;$

(b) $(var2\ == \ 0)\ \&\&\ ((var1/var2) >= 0;$

(c) $(var2 \mathrel{!=} 0)\ \&\&\ ((var1/var2) >= 0;$

(d) $(var2 \mathrel{!=} 0)\ ||\ ((var1/var2) >= 0.$

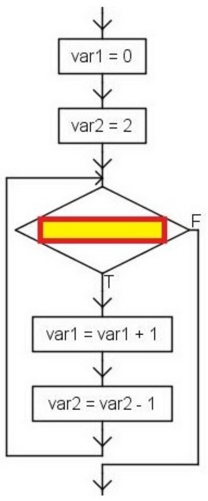
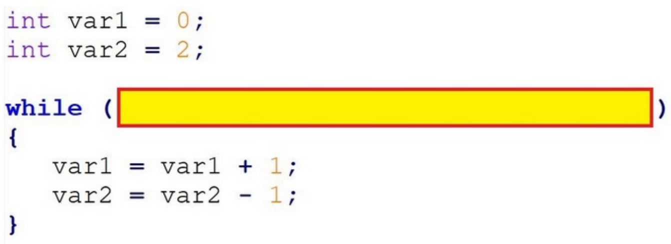


Figure B.4: Flow-chart version.



```
int var1 = 0;
int var2 = 2;

while (                                   )
{
   var1 = var1 + 1;
   var2 = var2 - 1;
}
```

Figure B.5: Code version.

Other topics that can be explored: notional machine, tracing.

B.2.7 Tasklet LC7

Problem statement: *What condition should the While have in order to get* $x = 4$ *and* $y = 3$*?*
Loop condition options:

(a) $y > 2\ ||\ x < y;$

(b) $y > 2\ \&\&\ x < y;$

(c) $y > 2\ ||\ x <= y;$

(d) $x > 2\ \&\&\ x < y.$

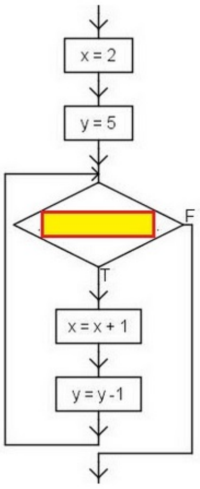
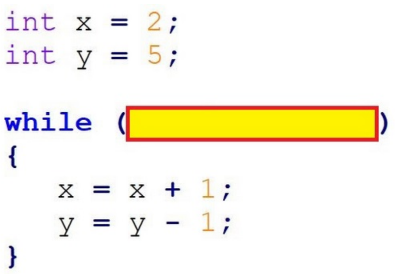


Figure B.6: Flow-chart version.



```
int x = 2;
int y = 5;

while (                )
{
    x = x + 1;
    y = y - 1;
}
```

Figure B.7: Code version.

Other topics that can be explored: notional machine, tracing.

B.2.8 Tasklet LC8

Problem statement: *What are the values of x and y when the code ends executing?* Output state options:

(a) x = 5, y = 2;

(b) x = 2, y = 5;

(c) x = 3, y = 4;

(d) x = 4, y = 3.

```
int x = 2;
int y = 5;

while (y > 2 && x < y) {
  x = x + 1;
  y = y - 1;
}
```

Listing B.7: Output state with loop complex condition, C/C++ example.

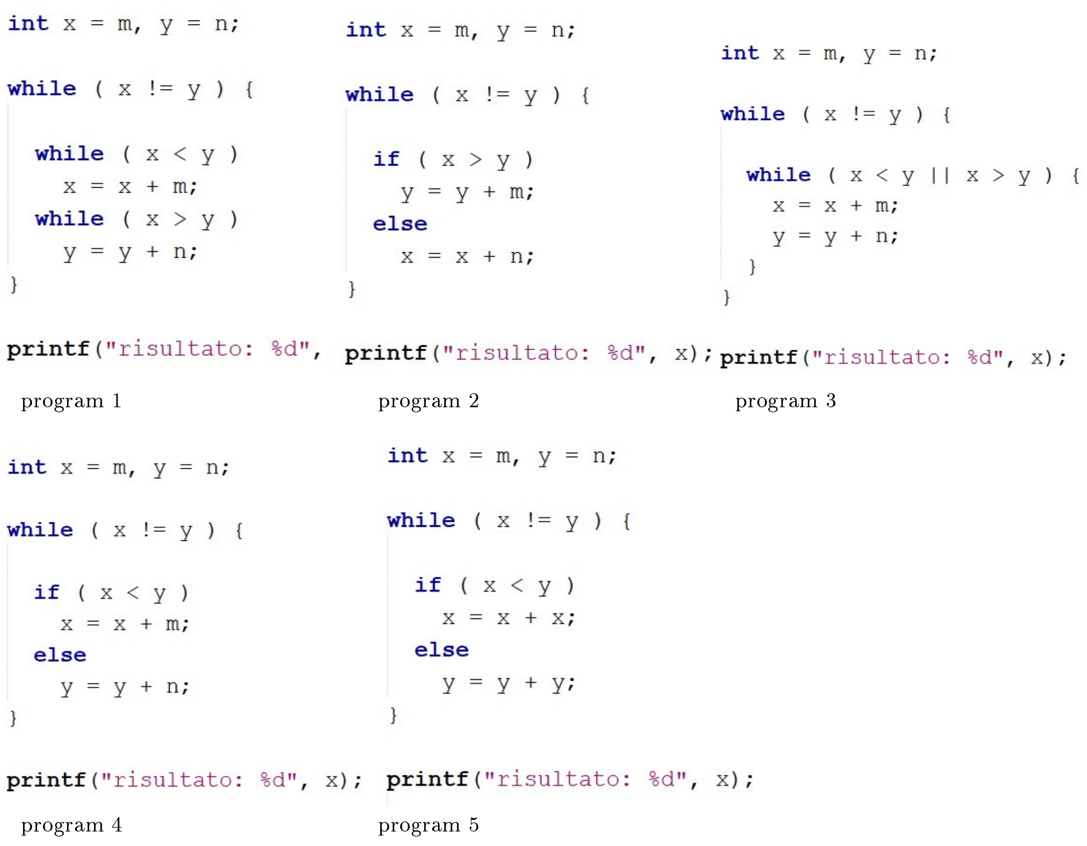
Other topics that can be explored: notional machine, tracing.

B.3 Equivalence

B.3.1 Tasklet E1

Problem statement: *The following programs are executed for positive values of m, n in input. Furthermore, two (or more) programs are equivalent if in every possible* *situation, with the same input data (positive values of m, n) they calculate the same result x. In your opinion, which of the five programs listed below are equivalent?*

Equivalence: select programs that are equivalent.



```
int x = m, y = n;

while ( x != y ) {

  while ( x < y )
    x = x + m;
  while ( x > y )
    y = y + n;
}

printf("risultato: %d",
```

program 1

```
int x = m, y = n;

while ( x != y ) {

  if ( x > y )
    y = y + m;
  else
    x = x + n;
}

printf("risultato: %d", x);
```

program 2

```
int x = m, y = n;

while ( x != y ) {

  while ( x < y || x > y ) {
    x = x + m;
    y = y + n;
  }
}
printf("risultato: %d", x);
```

program 3

```
int x = m, y = n;

while ( x != y ) {

  if ( x < y )
    x = x + m;
  else
    y = y + n;
}

printf("risultato: %d", x);
```

program 4

```
int x = m, y = n;

while ( x != y ) {

  if ( x < y )
    x = x + x;
  else
    y = y + y;
}

printf("risultato: %d", x);
```

program 5

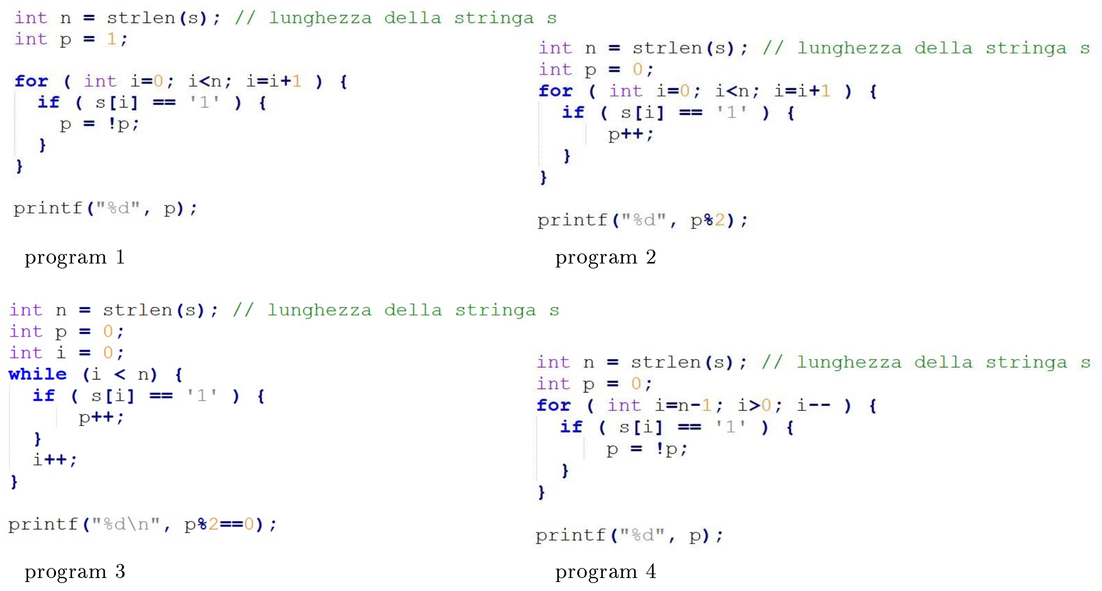
Figure B.8: *Which programs are equivalent?*

Other topics that can be explored: notional machine, tracing, nested flow control constructs, abstraction.

B.3.2 Tasklet E2

Problem statement: *The following programs run on integer arrays. Furthermore, two (or more) programs are equivalent if in every possible situation, with the same input data (array values), they calculate the same result. In your opinion, which of the four programs listed below are equivalent?*

Equivalence: select programs that are equivalent.



```
int n = strlen(s); // lunghezza della stringa s
int p = 1;

for ( int i=0; i<n; i=i+1 ) {
  if ( s[i] == '1' ) {
    p = !p;
  }
}

printf("%d", p);
```

program 1

```
int n = strlen(s); // lunghezza della stringa s
int p = 0;
for ( int i=0; i<n; i=i+1 ) {
  if ( s[i] == '1' ) {
    p++;
  }
}

printf("%d", p%2);
```

program 2

```
int n = strlen(s); // lunghezza della stringa s
int p = 0;
int i = 0;
while (i < n) {
  if ( s[i] == '1' ) {
    p++;
  }
  i++;
}

printf("%d\n", p%2==0);
```

program 3

```
int n = strlen(s); // lunghezza della stringa s
int p = 0;
for ( int i=n-1; i>0; i-- ) {
  if ( s[i] == '1' ) {
    p = !p;
  }
}

printf("%d", p);
```

program 4

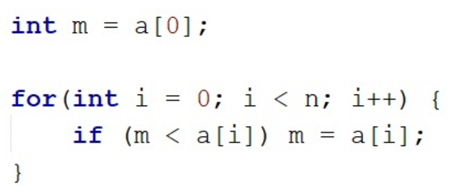
Figure B.9: *Which programs are equivalent?*

Other topics that can be explored: notional machine, tracing, nested flow control constructs, abstraction, non numerical values

B.3.3 Tasklet E3

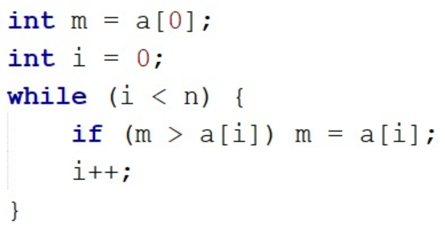
Problem statement: *Given an array a of size n which of these code fragments are equivalent?*

Equivalence: select programs that are equivalent.



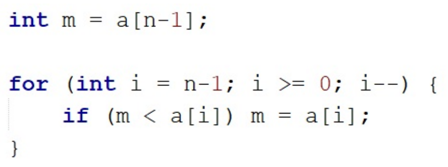
```
int m = a[0];

for(int i = 0; i < n; i++) {
    if (m < a[i]) m = a[i];
}
```

program 1



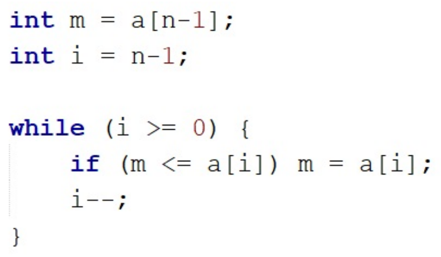
```
int m = a[0];
int i = 0;
while (i < n) {
    if (m > a[i]) m = a[i];
    i++;
}
```

program 2



```
int m = a[n-1];

for (int i = n-1; i >= 0; i--) {
    if (m < a[i]) m = a[i];
}
```

program 3



```
int m = a[n-1];
int i = n-1;

while (i >= 0) {
    if (m <= a[i]) m = a[i];
    i--;
}
```

program 4

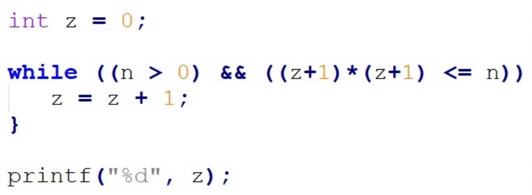
Figure B.10: *Which programs are equivalent?*

Other topics that can be explored: notional machine, tracing, nested flow control constructs, abstraction.

B.3.4 Tasklet E4

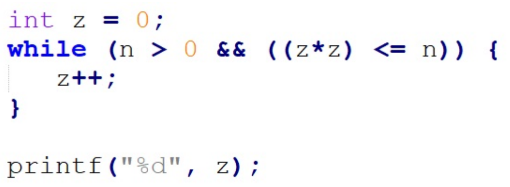
Problem statement: *Given an integer value n which of these code fragments are equivalent?*

Equivalence: select programs that are equivalent.



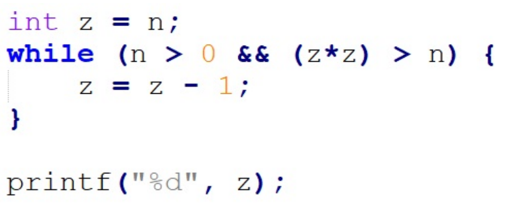
```
int z = 0;

while ((n > 0) && ((z+1)*(z+1) <= n))
    z = z + 1;
}

printf("%d", z);
```

program 1



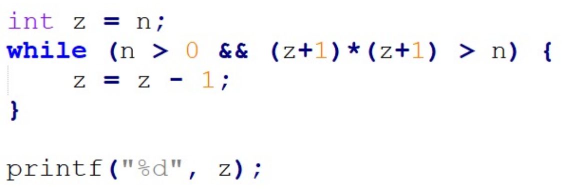
```
int z = 0;
while (n > 0 && ((z*z) <= n)) {
    z++;
}

printf("%d", z);
```

program 2



```
int z = n;
while (n > 0 && (z*z) > n) {
    z = z - 1;
}

printf("%d", z);
```

program 3



```
int z = n;
while (n > 0 && (z+1)*(z+1) > n) {
    z = z - 1;
}

printf("%d", z);
```

program 4

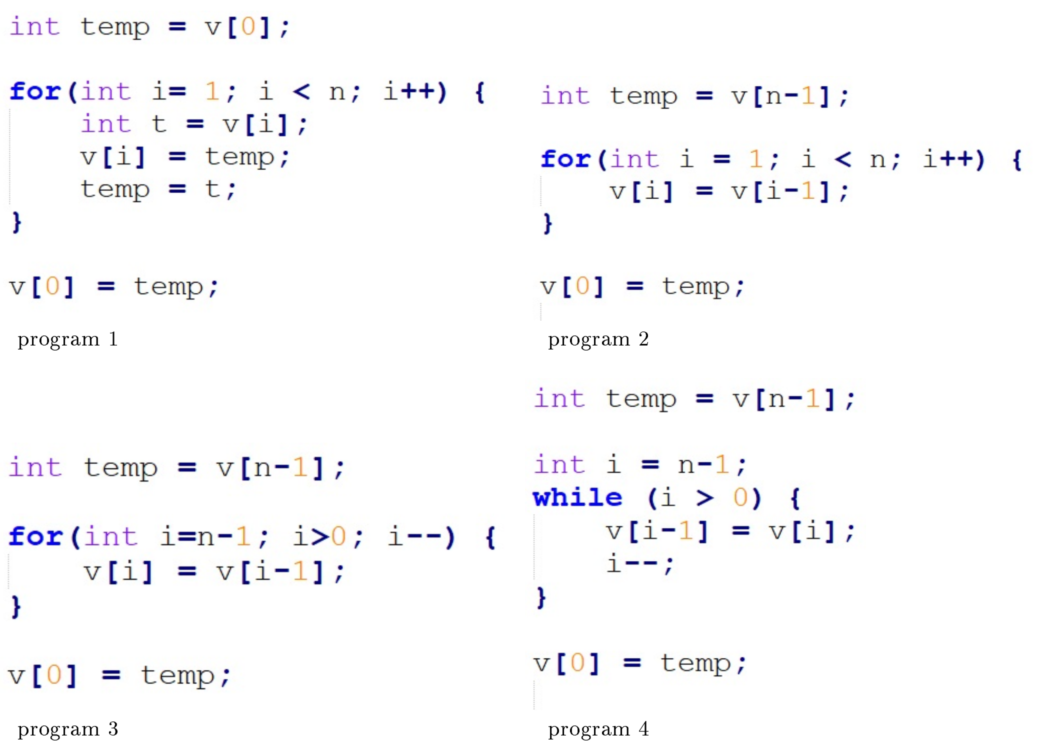
Figure B.11: *Which programs are equivalent?*

Other topics that can be explored: notional machine, tracing, loop complex conditions.

B.3.5 Tasklet E5

Problem statement: *The following programs run on integer arrays. Furthermore, two (or more) programs are equivalent if in every possible situation, with the same input data (array values), they calculate the same result. In your opinion, which of the four programs listed below are equivalent?*

Equivalence: select programs that are equivalent.



```
int temp = v[0];

for(int i= 1; i < n; i++) {
    int t = v[i];
    v[i] = temp;
    temp = t;
}

v[0] = temp;
```

program 1

```
int temp = v[n-1];

for(int i = 1; i < n; i++) {
    v[i] = v[i-1];
}

v[0] = temp;
```

program 2

```
int temp = v[n-1];

for(int i=n-1; i>0; i--) {
    v[i] = v[i-1];
}

v[0] = temp;
```

program 3

```
int temp = v[n-1];

int i = n-1;
while (i > 0) {
    v[i-1] = v[i];
    i--;
}

v[0] = temp;
```

program 4

Figure B.12: *Which programs are equivalent?*

Other topics that can be explored: notional machine, tracing.

B.3.6 Tasklet E6

Problem statement: *Given that both count and n are integers, which of the following statements is true for both blocks of code?*
Functional purpose options:

(a) I and II are exactly equivalent for all input values n.;

(b) I and II are equivalent only when n is an even number.;

(c) I and II are equivalent only when n = 0;

(d) I and II are equivalent for all values except when n = 0;

(e) I and II will never have the exact same outputs.

```
// code 1

for(count=0; count<=n; count++) {
  printf("%d", count);
}
```

Listing B.8: I for-loop code.

```
// code 2

count = 0;
while(count<=n) {
  count++;
  printf("%d", count);
}

```

Listing B.9: II while-loop code.

Other topics that can be explored: notional machine, tracing.

B.4 Nested loops

B.4.1 Tasklet NL1

Problem statement: *What is the result printed by the following code?*
Output state options: 4, 10, 16, 24, 30.

```
int total = 0;
int i;

for (int i = 1; i <= 4; i++) {
  int j;
  for (j = 1; j <= i; j++) {
    total = total + i;
  }
}

printf("%d", total);
```

Listing B.10: Output state with nested while-loops, C/C++ example.

Other topics that can be explored: tracing.

B.4.2 Tasklet NL2

Problem statement: *How many loops does the following code perform?*
Number of iterations options: 0, 7, 8, 10, more than 10, the loop never ends.

```c
int total = 0;
int i;

for (int i = 1; i <= 4; i++) {
  int j;
  for (j = 1; j <= i; j++) {
    total = total + i;
  }
}

printf("%d", total);
```

Listing B.11: Number of iterations with nested while-loops, C/C++ example.

Other topics that can be explored: tracing.

B.4.3 Tasklet NL3

Problem statement: *Given two input values n and m both integers, could you tell what the following code fragments can be used for?*
Functional purpose options:

(a) Calculate least common multiple (lcm);

(b) Calculate greatest common divisor (GCD);

(c) Calculate the root the greater of the two numbers;

(d) None of the above options.

```c
int x = m, y = n;

while (x != y) {
  while (x < y)
    x = x + m;
  while (x > y)
    y = y + n;
}

printf("result: %d", x);
```

Listing B.12: Functional purpose options with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, function purpose.

B.4.4 Tasklet NL4

Problem statement: *Given two input values n and m both integers, could you tell what the following code fragments can be used for?*
Functional purpose options:

(a) Calculate least common multiple (lcm);

(b) Calculate greatest common divisor (GCD);

(c) Calculate the root the greater of the two numbers;

(d) None of the above options.

```c
int x = m, y = n;

while (x != y) {
  if (x < y)
    x = x + m;
  else
    y = y + n;
}

printf("result: %d", x);
```

Listing B.13: Functional purpose options with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, function purpose.

B.4.5 Tasklet NL5

Problem statement: *Given two input values n and m both integers, could you tell what the following code fragments can be used for?*
Functional purpose options:

(a) Calculate least common multiple (lcm);

(b) Calculate greatest common divisor (GCD);

(c) Calculate the root the greater of the two numbers;

(d) None of the above options.

```c
int x = m, y = n;

while (x != y) {
  if (x > y)
    y = y + m;
  else
    x = x + n;
}

printf("result: %d", x);
```

Listing B.14: Functional purpose options with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, function purpose.

B.4.6 Tasklet NL6

Problem statement: *Given a string (array of characters) a of n elements, could you indicate what the following program does?*
Functional purpose options:

(a) Calculate the number of '1' present;

(b) Calculate the parity bit;

(c) Calculate the length of the maximum subsequence of '1';

(d) Nothing in particular.

Numbers of iterations options: 0, 5, 6, 7, the loop never ends.

```
int i = 0;
int x = 0;

while (i < n)) {
  int y = 0;
  int j = i + 1;

  if (a[i] == '1') y++;

  while (a[i] == a[j] && a[i] == '1') {
    y++;
    j++;
  }
  i++;
  if (y > x) x = y;
}

printf("result: %d", x);
```

Listing B.15: Functional purpose options with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, function purpose, non numerical values.

B.4.7 Tasklet NL7

Problem statement: *Consider the following code segment. How many times is the string "Hi!" printed as a result of executing the code segment?*
Number of iterations options: 8, 10, 12, 15, the loop never ends.

```
int i = 0;

while (i <= 4) {
  int j;
  for (j = 0; j < 3; j++) {
    printf("Hi!");
  }
  i++;
}
```

Listing B.16: Functional purpose options with nested while-loops, C/C++ example.

Other topics that can be explored: tracing, function purpose, non numerical values.

B.5 Reversibility

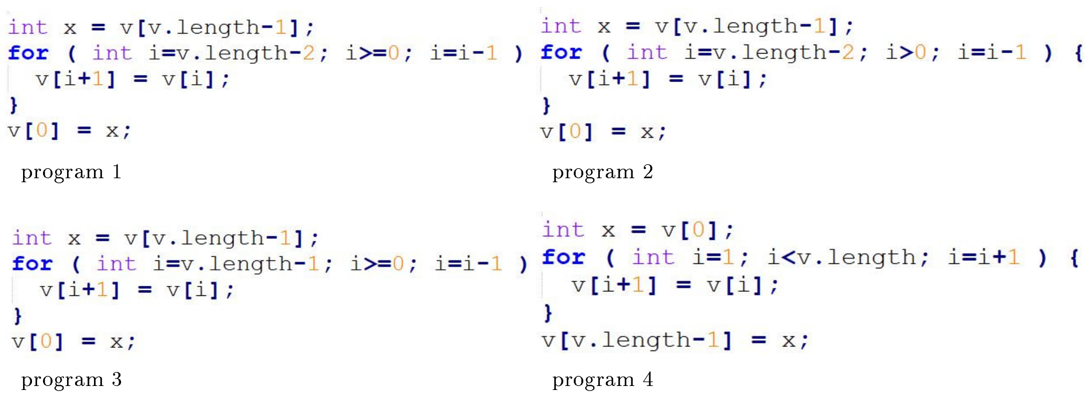
B.5.1 Tasklet R1

Problem statement: *The code presented in the image manipulates an array V, select the code fragments that restores the initial values of the array.*

Code that reverses the array and restores the starting values: four options (see Figure B.13).

```
int x = v[0];

for (int i = 1; i <= v.length; i++) {
  v[i-1] = v[i];
}

v[v.length-1] = x;
```

Listing B.17: Code manipulates an array with nested while-loops, Java example.



```
int x = v[v.length-1];
for ( int i=v.length-2; i>=0; i=i-1 )
  v[i+1] = v[i];
}
v[0] = x;
```

program 1

```
int x = v[v.length-1];
for ( int i=v.length-2; i>0; i=i-1 ) {
  v[i+1] = v[i];
}
v[0] = x;
```

program 2

```
int x = v[v.length-1];
for ( int i=v.length-1; i>=0; i=i-1 )
  v[i+1] = v[i];
}
v[0] = x;
```

program 3

```
int x = v[0];
for ( int i=1; i<v.length; i=i+1 ) {
  v[i+1] = v[i];
}
v[v.length-1] = x;
```

program 4

Figure B.13: *Which programs restore the initial values of the array?*

Other topics that can be explored: notional machine, tracing, function purpose, abstraction.

B.6 Functional purpose

B.6.1 Tasklet F1

Problem statement: *Analyze the following code fragment, identified any value x you could tell what it can be used for?*
Functional purpose options:

(a) Subtract a certain number of times 5 from the starting x number;

(b) Calculate the remainder of a division by 5;

(c) List multiples of 5 through x;

(d) It doesn't calculate anything in particular.

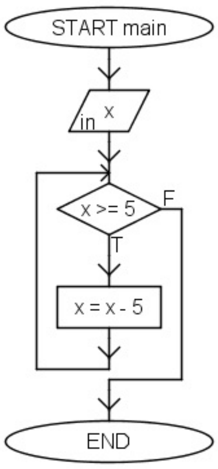


Figure B.14: Flow-chart version.

```
// int x; // x a integer value

while (x >= 5) {
  x = x - 5;
}

```

Listing B.18: Code version.

Other topics that can be explored: notional machine, tracing.

B.6.2 Tasklet F2

Problem statement: *Analyze the following code fragments, it contains an array a of N integers. Could you tell what it can be used for?*
Functional purpose options:

(a) Count the number of items present;

(b) Returns the item repeated multiple times;

(c) Returns the largest element;

(d) Sort the array;

(e) It doesn't calculate anything in particular.

```c
int c = 0;
int b = 0;
int x = 0;
int y = 0;

int i, j;

for (i = 0; i < N; i++) {
  x=0;
  for (j = 0; j < N; j++) {
    if (a[i] ==a [j]) {
      c = i;
      x++;
    }
  }
  if (x > y) {
    y = x;
    b = c;
  }
}

printf("%d, %d\n", a[b], y);
```

Listing B.19: Functional purpose with nested for-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, nested flow control construct.

B.6.3 Tasklet F3

Problem statement: *Given an array of N integers and b an array of M integers, can you tell what the following code fragments can do?*
Functional purpose options:

(a) Prints the elements belonging to the intersection of the two sets;

(b) Prints the elements belonging to the union of the two sets;

(c) Print elements that have the same position;

(d) Sort array a;

(e) It doesn't calculate anything in particular.

```c
int i=0;

while (i < N) {
  int j=0;
  while (j < M) {
    if (a[i] == b[j]) printf("%d\n", a[i]);
    j++;
  }
```

```
    i++;
  }
```

Listing B.20: Functional purpose with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, nested flow control construct.

B.6.4 Tasklet F4

Problem statement: *Given a sequence/string of 0s and 1s, what does the following program calculate?*
Functional purpose options:

(a) Count the number of '1' present;

(b) Calculate parity bit 1 when the number of '1' is even;

(c) Calculate parity bit 1 when the number of '1' is odd;

(d) It doesn't calculate anything in particular.

```
int n = strlen(s); // length of the string s
int p = 0;
int i = 0;

while (i < n) {
  if (s[i] == '1') {
    p++;
  }
  i++;
}

printf("%d\n", p%2==0);
```

Listing B.21: Functional purpose with nested while-loops, C/C++ example.

Other topics that can be explored: notional machine, tracing, not numerical values.

B.6.5 Tasklet F5

Problem statement: *Given a sequence/string of 0s and 1s, what does the following program calculate?*
Functional purpose options:

(a) Calculate the minimum value;

(b) Calculate the average value;

(c) Calculate the maximum value;

(d) It doesn't calculate anything in particular.

```
int m = a[n-1];
int i = n-1;

while (i >= 0) {
  if (m <= a[i]) m = a[i];
  i--;
}
```

Listing B.22: Functional purpose with nested flow-control construnct, C/C++ example.

Other topics that can be explored: notional machine, tracing.

B.6.6 Tasklet F6

Problem statement: *Given a sequence/string of 0s and 1s, what does the following program calculate?*
Functional purpose options:

(a) Calculate the minimum value;

(b) Calculate the average value;

(c) Calculate the maximum value;

(d) It doesn't calculate anything in particular.

```
int m = a[n-1];
int i;

for (i = n-1; i >= 0; i--) {
  if (m <= a[i]) m = a[i];
}
```

Listing B.23: Functional purpose with nested flow-control construnct, C/C++ example.

Other topics that can be explored: notional machine, tracing.

B.7 Loop control variable

B.7.1 Tasklet CV1

Problem statement: *If we input an array v of 5 elements 2,3,5,7,8, after the execution of the code how will the array be changed?*
Output state options:

(a) 2, 3, 5, 7, 8;

(b) 3, 4, 6, 8, 9;

(c) 2, 5, 10, 15, 25;

(d) 2, 5, 10, 17, 25.

Problem statement: *Given an array v, what does the following program calculate?* Functional purpose options:

(a) Calculate the sum of the elements of the array;

(b) Move each element one position (shift);

(c) Calculate the cumulative sum;

(d) It doesn't calculate anything in particular.

```
int x = 0;
int i;

for (i = 0; i < n; i = i + 1) {
  x = x + v[i];
  v[i] = x;
}
```

Listing B.24: For-loop code that manipulates an array, C/C++ example.

Other topics that can be explored: notional machine, tracing, functional purpose.

Appendix C

Computer Science in Italian school

In Italy the secondary school system is organized on two levels: (1) Lower secondary education, grades 6th to 8th (pupils aged 11–13); (2) Upper secondary education, grades 9th to 13th (pupils aged 14–18). Schooling is compulsory for ten years, thus pupils normally have to attend school until the age of 16 (grade 10th) [Bel+14].

The lower secondary education level is planned over three years. Informatics is not a subject taught by itself, but pupils attend "Mathematics and Science" and "Technology" in which informatics should be introduced.

Upper secondary education in Italy is divided into three types of high school: lyceum, technical school, vocational school. Courses last 5 years.

– *Lyceum* aims at a general education typically refined with further tertiary studies and available with different main focuses: Classical, Scientific, Linguistic, Artistic, Music, Human Sciences.

– *Technical schools* aim at vocational education with two main orientations: Technology and Economics. The education given in a technical school offers both theoretical education and also qualified technical specialization in a specific field of studies.

– *Professional schools* aim at vocational education with two main orientations: Services and Manufacturing.

Computer Science education is offered in Scientific Lyceum of Applied Sciences, and also in Economics high school and technological high school.

C.1 Scientific Lyceum of Applied Sciences

Scientific Lyceum of Applied Sciences offers Computer Science for all 5 years of studies, 2 hours per week.

The lessons should include the main topics of the discipline: hardware architectures, operating systems, algorithms and programming languages, coding, computer networks, databases, numerical simulations, mark-up languages, and some web application.

C.2 Technical high school

High school with technical specialization in Computer Science offers a variety of courses, as shown in Table C.1.

Table C.1: Study hours per week in technical subjects.

Course	**1**	**2**	**3**	**4**	**5**
Technology of Informatics	3				
Science and Applied Technology		3			
Mathematics	4	4	4	4	3
Informatics			6	6	6
Systems and Networks			4	4	4
Technology and System Design			3	3	4
Project management and business organization					3
Telecommunications			3	3	

The Technology of Informatics course, in the first year, is common to all the curricula present in the technical schools. Technology of Informatics should provide basic knowledge concerning: information, data and codification; computer architecture; human-computer communication; functionalities of an operating system; software tools and applications; problem solving and representation; fundamentals of programming; structure of a computer network; Internet networking; privacy and copyright issues.

C.3 Economics high school

For the Economics sector, the main emphasis is on the potential of ICT as a means of reorganizing businesses and optimizing processes and data management.

High school with an Economics specialization in *Company Information Systems* offers a smaller variety of courses, as shown in Table C.2.

Table C.2: Study hours per week in technical subjects.

Course	**1**	**2**	**3**	**4**	**5**
Mathematics	4	4	3	3	3
Informatics	2	2	4	5	5

High school with an Economics specialization in *Administration, Finance and Marketing* offers a smaller variety of courses and fewer hours per week, as shown in Table C.3.

Table C.3: Study hours per week in technical subjects.

Course	**1**	**2**	**3**	**4**	**5**
Mathematics	4	4	3	3	3
Informatics	2	2	2	2	

Appendix D

Teacher Pilot Interview Protocol

English version of the questions asked to the teachers, reported in the order followed during the pilot interviews.

```
0. General information
Male/Female
Where do you teach? (Your institution)
In which field are your students specializing?
What subject do you teach?
How long have you been teaching?
What is your academic background?

1. What programming language do you use to introduce the basics
of programming?
2. What are the most challenging key programming concepts in your
teaching perspective?
3. What is the major learning obstacle that students face before
being introduced to object-oriented
programming?

[Answers to question 2 may include OO concepts, but here the intended
focus is on imperative and procedural concepts.]

4. In which order do you teach the basic programming-related concepts?
5. How much time do you plan for each basic concept?
6. Can you show some of your favorite examples to make students learn
how to apply the iteration constructs?
7. Are the tasks assigned to students simple variations of those dealt
with in class? Or do they address unfamiliar
situations as well?
8. What are the extra-computing prerequisites necessary to understand
```

the basic programming concepts as well as the examples you show?
9. In your experience, to what extent can students master the termination condition of a loop?
10. In your teaching, do you cover the mappings between different iteration constructs (for, while, do-while, repeat-until)?
11. How do you usually assess an incorrect termination condition? And oversights about the first or last iteration?
12. How do you assess a working solution if it is inefficient, or convoluted, or somehow at odds with what you expected?
13. While trying to achieve the assigned tasks, do you expect your students to apply the models introduced in class? Or do you also appreciate “creative” solutions?
14. Which features of the iteration constructs are usually understood by (most) students, and which are more difficult to them?
15. What are your more frequent suggestions to students for improving their programming performance?
16. Are the different solutions proposed by students compared in class? How?
17. What are your strategies to motivate students?
18. How do you attempt to manage different learning styles?
19. How do you deal with students’ criticisms in order to address their possible needs?

20. Any other issues you deem important to consider about the teaching/learning of programming?

Appendix E

Student Pilot Survey Protocol

English version of the student pilot survey, including questions about their subjective perception as well as three tasklets on the iteration constructs.

Student Survey

General information

What is your gender?

- ○ Male
- ○ Female

In which school are you studying?

- ○ Technological school
- ○ Scientific lyceum
- ○ Economic school

What year do you attend?

- ○ First
- ○ Second
- ○ Third
- ○ Fourth
- ○ Fifth

What is your average rating in Informatics?

[*Choose among: 10 / 9 / 8 / 7 / 6 / Insufficient*]

Which are your favorite programming languages? [*multiple options are possible*]

☐ C/C++	☐ Java	☐ Scratch	☐ Python
☐ C#	☐ Visual Basic	☐ Javascript	☐ Other: ....

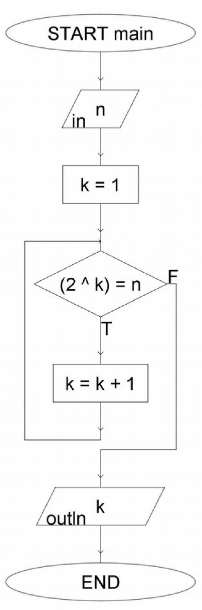
Do you think it would be needed to spend more time on some programming concepts?
Which ones? [*multiple options are possible*]

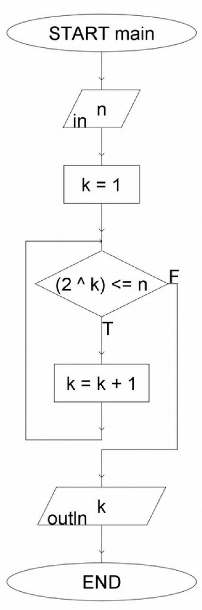
☐ Variables	☐ Iteration loops (*while/for/do-while*)	☐ Recursion
☐ Data types	☐ Arrays	☐ Input/output
☐ Selection (*if/if-else/switch*)	☐ Subroutines (*functions/procedures*)	☐ Other: ....

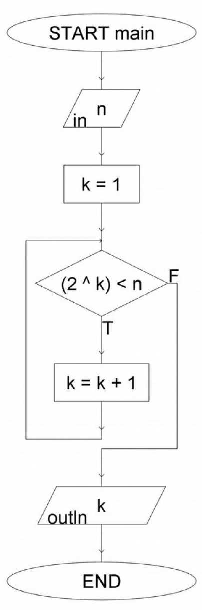
Are you usually successful in solving unfamiliar programming problems? [*Lickert scale: 4 levels*]

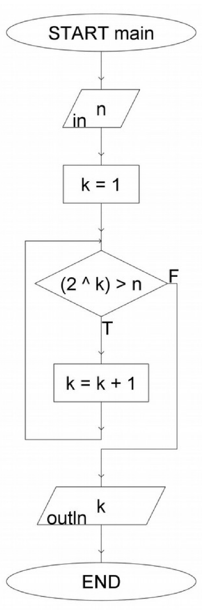
	1	2	3	4	
Never or almost never	○	○	○	○	Always

Problem: Design of an algorithm to determine the number of bits required to represent in binary a positive integer *n*; such number of bits corresponds to the smallest exponent *k* such that 2^k (2 raised to the *k-th* power) is greater than *n*. Which one of the proposed algorithms is correct, in your opinion?









○ Option 1 ○ Option 2 ○ Option 3 ○ Option 4

Problem: The program below is meant to verify if two positive integer *m* and *n* are co-primes. If the input values are *m=15* and *n=44*, how many times the *while* loop will iterate?

```
int x = m, y = n;

while ( x > 1 && y > 1 && x != y ) {

  if ( x < y )
    y = y - x;
  else
    x = x - y;
}
if ( x == 1 || y == 1 )
  printf( "m=%d e n=%d sono primi fra loro", m, n );
else
  printf( "m=%d e n=%d non sono primi fra loro", m, n );
```

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 or more ○ The loop will not terminate

Problem: It is assumed that the input data, *m* and *n*, of the following programs are positive integers. Two (or more) such programs are equivalent if they compute and print the same output value whenever they are executed for the same pair of positive integers (*m*, *n*). Which ones among the five programs reported below are equivalent, in your opinion?

```c
int x = m, y = n;

while ( x != y ) {

  while ( x < y )
    x = x + m;
  while ( x > y )
    y = y + n;
}

printf("risultato: %d", x);
```

☐ Program 1

```c
int x = m, y = n;

while ( x != y ) {

  if ( x > y )
    y = y + m;
  else
    x = x + n;
}

printf("risultato: %d", x);
```

☐ Program 2

```c
int x = m, y = n;

while ( x != y ) {

  while ( x < y || x > y ) {
    x = x + m;
    y = y + n;
  }
}

printf("risultato: %d", x);
```

☐ Program 3

```c
int x = m, y = n;

while ( x != y ) {

  if ( x < y )
    x = x + m;
  else
    y = y + n;
}

printf("risultato: %d", x);
```

☐ Program 4

```c
int x = m, y = n;

while ( x != y ) {

  if ( x < y )
    x = x + x;
  else
    y = y + y;
}

printf("risultato: %d", x);
```

☐ Program 5

Have you been unsuccessful in achieving your programming tasks because of a lack of clear understanding of some mathematical/logic concepts? Which ones? [*multiple options are possible*]

☐ Boolean algebra and logic operators	☐ Concept of variable	☐ De Morgan's formulae	☐ knowledge/meaning of terms and words
☐ Concept of function	☐ Basic concepts of geometry	☐ Set theory	

Which kind of errors has been most penalizing for your grading?

....

What do you find most difficult when trying to use a loop?

- ○ Identifying the loop condition for the *while* or *do-while* constructs
- ○ Defining a complex condition, using logic operators (AND, OR, NOT, XOR)
- ○ Dealing with nested loops
- ○ Realizing, in general, when a loop must terminate to iterate
- ○ Dealing with a variable that counts the iterations

Are the informatics topics you learnt in accordance with your expectations? [*Lickert scale: 4 levels*]

	1	2	3	4	
Not at all	○	○	○	○	Yes, I'm happy

Do you have any suggestion to make learning informatics more interesting?

....

Bibliography

[Abe08] Sandra K. Abell. "Twenty Years Later: Does pedagogical content knowledge remain a useful idea?" eng. In: *International journal of science education* 30.10 (2008), pp. 1405–1416.

[Ade85] Beth Adelson. "Comparing Natural and Abstract Categories: A Case Study from Computer Science". In: *Cognitive Science* 9.4 (1985), pp. 417–430. DOI: `https://doi.org/10.1207/s15516709cog0904\_3`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog0904_3`.

[AGE06] Michal Armoni and Judith Gal-Ezer. "Reduction – an abstract thinking pattern: the case of the computational models course". In: *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education.* New York, NY, USA: ACM, 2006, pp. 389–393. DOI: `http://doi.acm.org/10.1145/1121341.1121461`.

[AIES14] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. "Abstracting and Narrating Novice Programs Using Regular Expressions". In: *Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology.* SAICSIT '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 19–28. DOI: `10.1145/2664591.2664601`.

[AK16] Alicia C. Alonzo and Jiwon Kim. "Declarative and dynamic pedagogical content knowledge as elicited through two video-based interview methods". eng. In: *Journal of research in science teaching* 53.8 (2016), pp. 1259–1286.

[Alm+06] Vicki Almstrum et al. "Concept Inventories in Computer Science for the Topic Discrete Mathematics". In: *ACM SIGCSE Bulletin Inroads* 38 (Dec. 2006), pp. 132–145. DOI: `10.1145/1189136.1189182`.

[Alz+18] Nabeel Alzahrani et al. "An analysis of common errors leading to excessive student struggle on homework problems in an introductory programming course". In: *2018 ASEE Annual Conference & Exposition.* 2018.

[Arm13] Michal Armoni. “On teaching abstraction in computer science to novices”. In: *Journal of Computers in Mathematics and Science Teaching* 32.3 (July 2013), pp. 265–284.

[Arn94] David Arnow. “Teaching Programming to Liberal Arts Students: Using Loop Invariants”. In: *Proceedings of the 25th SIGCSE Symposium on Computer Science Education*. SIGCSE ’94. New York, NY, USA: ACM, 1994, pp. 141–144.

[AS08] Russ Abbott and Chengyu Sun. “Abstraction abstracted”. en. In: *Proceedings of the 2nd international workshop on The role of abstraction in software engineering - ROA ’08*. Leipzig, Germany: ACM Press, 2008, p. 23. DOI: `10.1145/1370164.1370171`.

[Ast91] Owen Astrachan. “Pictures As Invariants”. In: *Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’91. New York, NY, USA: ACM, 1991, pp. 112–118.

[Ati+20] Kinnari Atit et al. “Examining the role of spatial skills and mathematics motivation on middle school mathematics achievement”. In: *International Journal of STEM Education* 7.1 (2020), pp. 1–13.

[BA01] Mordechai Ben-Ari. *Mathematical logic for computer science*. eng. 2nd ed. London: Springer, 2001.

[Ban00] Albert Bandura. *Autoefficacia: Teoria e applicazioni.(Presentazione all’edizione italiana di Gian Vittorio Caprara)*. Edizioni Erickson, 2000.

[Ban05] Albert Bandura. “The evolution of social cognitive theory”. In: *Great minds in management* (2005), pp. 9–35.

[Bar+14] Erik Barendsen et al. “Eliciting computer science teachers’ PCK using the Content Representation format: Experiences and future directions”. In: *Proceedings of the 6th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP’14) – Teaching and Learning Perspectives*. Ed. by Yasemin Gülbahar, Erinç Karataş, and Müge Adnan. Vol. 8730. Istanbul, Turkey: Ankara University Press, Sept. 2014, pp. 71–82.

[Bar+15] Erik Barendsen et al. “Concepts in K-9 Computer Science Education”. In: *Proceedings of the 2015 ITiCSE on Working Group Reports*. ITICSE-WGR ’15. New York, NY, USA: ACM, 2015, pp. 85–116. DOI: `10.1145/2858796.2858800`.

[BC06] Jens Bennedsen and Michael E. Caspersen. “Abstraction Power in Computer Science Education”. In: *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2006*. University of Sussex, Brighton, UK, Sept. 2006.

[Bec21] Brett A. Becker. “What Does Saying That ’programming is Hard’ Really Say, and about Whom?” In: *Commun. ACM* 64.8 (2021), pp. 27–29. DOI: `10.1145/3469115`.

[Bel+14] Carlo Bellettini et al. “Informatics Education in Italian Secondary Schools”. In: *ACM Trans. Comput. Educ.* 14.2 (2014). DOI: `10.1145/2602490`.

[BFC82] John Biggs and Kevin F Collis. “Evaluating the Quality of Learning: the SOLO Taxonomy”. In: *SERBIULA (sistema Librum 2.0)* (Jan. 1982).

[BFL99] Albert Bandura, WH Freeman, and Richard Lightsey. *Self-efficacy: The exercise of control.* 1999.

[BH74] Alan D. Baddeley and Graham Hitch. “Working memory”. In: *Psychology of learning and motivation.* Vol. 8. Elsevier, 1974, pp. 47–89.

[Bis15] Gian Italo Bischi. *Matematica e Letteratura. Dalla Divina Commedia al Noir.* EGEA, Milano, 2015, p. 160.

[BJ72] John D. Bransford and Marcia K. Johnson. “Contextual prerequisites for understanding: Some investigations of comprehension and recall”. In: *Journal of Verbal Learning and Verbal Behavior* 11.6 (1972), pp. 717 –726. DOI: `https://doi.org/10.1016/S0022-5371(72)80006-9`.

[Bla16] Andreas Blass. “Symbioses between mathematical logic and computer science”. In: *Annals of Pure and Applied Logic* 167.10 (2016). Logic Colloquium 2012, pp. 868 –878. DOI: `https://doi.org/10.1016/j.apal.2014.04.018`.

[BLW01] Paolo Bucci, Timothy J. Long, and Bruce W. Weide. “Do we really teach abstraction?” In: *SIGCSE ’01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education.* New York, NY, USA: ACM, 2001, pp. 26–30. DOI: `10.1145/364447.364531`.

[Bro87] A. Richard Brown. “Metacognition, executive control, self-regulation, and other more mysterious mechanisms”. In: *F.E. Weinert & R.H. Kluwe, (Eds.) Metacognition, motivation, and understanding.* 1987.

[BRT05] Susan Bergin, Ronan Reilly, and Desmond Traynor. “Examining the Role of Self-Regulated Learning on Introductory Programming Performance”. In: *Proceedings of the 1st International Workshop on Computing Education Research.* ICER ’05. New York, NY, USA: ACM, 2005, pp. 81–86.

[BS11] Valerie Barr and Chris Stephenson. “Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community?” In: *ACM Inroads* 2 (Mar. 2011). DOI: `10.1145/1929887.1929905`.

[BS13] Teresa Busjahn and Carsten Schulte. “The Use of Code Reading in Teaching Programming”. In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research.* Koli Calling ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 3–11. DOI: `10.1145/2526968.2526969`.

[BS83] Jeffrey Bonar and Elliot M. Soloway. “Uncovering principles of novice programming”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 1983, pp. 10–13.

[BS85] Jeffrey Bonar and Elliot M. Soloway. “Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers”. In: *Human-Computer Interaction* 1 (June 1985), pp. 133–161. DOI: `10.1207/s15327051hci0102_3`.

[BSS13] Malte Buchholz, Mara Saeli, and Carsten Schulte. “PCK and reflection in computer science teacher education”. In: *ACM International Conference Proceeding Series* (Nov. 2013). DOI: `10.1145/2532748.2532752`.

[Cac+16] Ricardo Caceffo et al. “Developing a Computer Science Concept Inventory for Introductory Programming”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* SIGCSE ’16. New York, NY, USA: ACM, 2016, pp. 364–369. DOI: `10.1145/2839509.2844559`.

[CAD19] Hasnaa Chaabi, Amina Azmani, and Juan Manuel Dodero. “Analysis of the relationship between computational thinking and mathematical abstraction in primary education”. In: *Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality.* 2019, pp. 981–986.

[CADM05] Romeo Crapiz, Franca Alborini, and Mirka De Marchi. “Letteratura e Informatica Un’esperinza didattica al Liceo Copernico di Udine”. it. In: *Didamatica 2005: Didattica Informatica.* Potenza, Italy: AICA, Oct. 2005, pp. 994–1002.

[CCDB07] Barbara Carretti, Cesare Cornoldi, and Rossana De Beni. *Il disturbo di comprensione del testo.* 2007.

[CD17] Ibrahim Cetin and Ed Dubinsky. “Reflective abstraction in computational thinking”. eng. In: *The Journal of mathematical behavior* 47 (2017), pp. 70–80.

[Cet15] Ibrahim Cetin. "Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective". In: *Canadian Journal of Science, Mathematics and Technology Education* 15.2 (Feb. 2015), pp. 155–170. DOI: `10.1080/14926156.2015.1014075`.

[Cet+20] Ibrahim Cetin et al. "Teaching Loops Concept through Visualization Construction". In: *Informatics in Education-An International Journal* 19.4 (2020), pp. 589–609.

[Che18] Eugenia Cheng. *The Art of Logic: How to Make Sense in a World that Doesn't*. Profile, 2018.

[CHR13] Pavol Cerny, Thomas A Henzinger, and Arjun Radhakrishna. "Quantitative abstraction refinement". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2013, pp. 115–128.

[CLM20] Umberto Costantini, Violetta Lonati, and Anna Morpurgo. "How Plans Occur in Novices' Programs: A Method to Evaluate Program-Writing Skills". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 852–858. DOI: `10.1145/3328778.3366870`.

[CMM07] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research methods in education*. London, New York: Routledge, 2007.

[CO99] Kate Cain and Jane V. Oakhill. "Inference making ability and its relation to comprehension failure in young children". In: *Reading and Writing* 11 (1999), pp. 489–503.

[Con94] Bradford R. Connatser. "Setting the Context for Understanding". In: *Technical Communication* 41.2 (1994), pp. 287–291.

[Cor+18] Cesare Cornoldi et al. *Processi cognitivi, motivazione e apprendimento*. Bologna: il Mulino, 2018.

[Cor95] Cesare Cornoldi. *Metacognizione e apprendimento*. Strumenti. Psicologia. Il Mulino, 1995.

[Cot06] Lucio Cottini. *La didattica metacognitiva*. 2006.

[CPC13] Donna Christy, Christine Payson, and Patricia Carnevale. "The Bridge to Mathematics and Literature". In: *Mathematics Teaching in the Middle School* 18.9 (2013), pp. 572–577.

[CPP12] Michelle Craig, Sarah Petersen, and Andrew Petersen. "Following a Thread: Knitting Patterns and Program Tracing". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 233–238. DOI: `10.1145/2157136.2157204`.

[Csi+15] Andrew Csizmadia et al. “Computational thinking - a guide for teachers”. In: *Computing At School* (Jan. 2015).

[Cur+19] Paul Curzon et al. “Computational thinking”. In: *The Cambridge Handbook of Computing Education Research* (2019), pp. 513–546.

[CW15] Jacqui Chetty and Duan van der Westhuizen. “Towards a Pedagogical Design for Teaching Novice Programmers: Design-Based Research as an Empirical Determinant for Success”. In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. Koli Calling ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 5–12. DOI: 10.1145/2828959.2828976.

[CZP14] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. “Identifying Challenging CS1 Concepts in a Large Problem Dataset”. In: *Proc. of the 45th ACM Tech. Symp. on Computer Science Education*. SIGCSE ’14. New York, NY, USA: ACM, 2014, pp. 695–700.

[Dav95] Martin Davis. “The Universal Turing Machine (2Nd Ed.)” In: ed. by Rolf Herken. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1995. Chap. Influences of Mathematical Logic on Computer Science, pp. 289–299.

[DB86] Benedict Du Boulay. “Some Difficulties of Learning to Program”. In: *Journal of Educational Computing Research* 2 (Jan. 1986), pp. 57–73.

[DBP93] Rossana De Beni and Francesca Pazzaglia. *Lettura e metacognizione. Attività didattiche per la comprensione del testo*. Guide per l’educazione speciale. Centro Studi Erickson, 1993.

[DC80] Meredyth Daneman and Patricia A Carpenter. “Individual differences in working memory and reading”. In: *Journal of verbal learning and verbal behavior* 19.4 (1980), pp. 450–466.

[Deh09] Saeed Dehnadi. “A cognitive study of learning to program in introductory programming courses.” PhD thesis. Middlesex University, 2009.

[Dij72] Edsger W. Dijkstra. “The Humble Programmer”. In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[Dij89] Edsger W. Dijkstra. “On the cruelty of really teaching computing science”. English. In: *Communications Of The Acm* 32.12 (1989), pp. 1398–1404.

[DMP15] Liesbeth De Mol and Giuseppe Primiero. “When logic meets engineering: introduction to logical issues in the history and philosophy of computer science”. eng. In: *History and Philosophy of Logic* 36.3 (2015), pp. 195–204.

[Dou04] Mark Dougherty. “What Has Literature to Offer Computer Science?” In: *Humanit* 7.1 (2004), pp. 74–91.

[DR08] Michael De Raadt. "Teaching programming strategies explicitly to novice programmers". PhD thesis. University of Southern Queensland, 2008.

[Dre91] Tommy Dreyfus. "Advanced Mathematical Thinking Processes". In: *Advanced Mathematical Thinking*. Ed. by David Tall. Dordrecht: Springer Netherlands, 1991, pp. 25–41. DOI: 10.1007/0-306-47203-1_2.

[DRWT09a] Michael De Raadt, Richard Watson, and Mark Toleman. "Teaching and assessing programming strategies explicitly". In: *Proceedings of the 11th Australasian Conference on Computing Education - Volume 95*. ACE '09. Darlinghurst, Australia: Australian Computer Society, Inc., 2009, pp. 45–54.

[DRWT09b] Michael De Raadt, Richard Watson, and Mark Toleman. "Teaching and assessing programming strategies explicitly". In: *Proceedings of the 11th Australasian Computing Education Conference (ACE 2009)*. Vol. 95. Australian Computer Society Inc. 2009, pp. 45–54.

[DSL18] Rodrigo Duran, Juha Sorva, and Sofia Leite. "Towards an Analysis of Program Complexity From a Cognitive Perspective". In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 21–30. DOI: 10.1145/3230977.3230986.

[DT19] Peter J. Denning and Matti Tedre. *Computational thinking*. Mit Press, 2019.

[DT21] Peter J. Denning and Matti Tedre. "Computational Thinking: A disciplinary perspective". In: *Informatics in Education* 20.3 (2021), pp. 361–390.

[Eck+06] Anna Eckerdal et al. "Putting threshold concepts into context in computer science education". In: *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*. ITICSE '06. New York, NY, USA: ACM, 2006, pp. 103–107.

[EHR20] Barbara Ericson, Beryl Hoffman, and Jennifer Rosato. "CSAwesome: AP CSA Curriculum and Professional Development (Practical Report)". In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. WiPSCE '20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421593.

[Erl73] Stanley H. Erlwanger. "Benny's conception of rules and answers in IPI mathematics". In: *Journal of Children's Mathematical Behaviour 1, 2, Autumn* (1973), pp. 7–26.

[FAO10] José Luis Fernández Alemán and Youssef Oufaska. “SAMtool, a Tool for Deducing and Implementing Loop Patterns”. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’10. New York, NY, USA: ACM, 2010, pp. 68–72. DOI: 10.1145/1822090.1822111.

[Fin+20] Sally Fincher et al. “Notional Machines in Computing Education: The Education of Attention”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 21–50. DOI: 10.1145/3437800.3439202.

[Fis14] Kathi Fisler. “The Recurring Rainfall Problem”. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 35–42. DOI: 10.1145/2632320.2632346.

[Fla92] John H. Flavell. “Cognitive development: Past, present, and future.” en. In: *Developmental Psychology* 28.6 (1992), pp. 998–1005. DOI: 10.1037/0012-1649.28.6.998.

[FM93] Luciana Ferraboschi and Nadia Meini. *Strategie semplici di lettura. Esercizi guida per la comprensione del testo*. Materiali di recupero e sostegno. Centro Studi Erickson, 1993.

[FMV14] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. “Loop invariants: Analysis, classification, and examples”. eng. In: *ACM Computing Surveys (CSUR)* 46.3 (2014), pp. 1–51.

[Fre+18] Stephen Frezza et al. “Modelling Competencies for Computing Education beyond 2020: A Research Based Approach to Defining Competencies in the Computing Disciplines”. In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 148–174.

[Ful+07] Ursula Fuller et al. “Developing a computer science-specific learning taxonomy”. eng. In: *ACM SIGCSE Bulletin* 39.4 (2007), pp. 152–170.

[GA06] David Ginat and Michal Armoni. “Reversing: An Essential Heuristic in Program and Proof Design”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’06. New York, NY, USA: ACM, 2006, pp. 469–473. DOI: 10.1145/1121341.1121488.

[GA12] David Ginat and Ronnie Alankry. “Pseudo Abstract Composition: The Case of Language Concatenation”. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 28–33. DOI: 10.1145/2325296.2325307.

[GB17a] David Ginat and Yoav Blau. “Multiple Levels of Abstraction in Algorithmic Problem Solving”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 237–242. DOI: 10.1145/3017680.3017801.

[GB17b] Shuchi Grover and Satabdi Basu. “Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 267–272. DOI: 10.1145/3017680.3017723.

[GCS18] Kristin L. Gunckel, Beth A. Covitt, and Ivan Salinas. “Learning progressions as tools for supporting teacher content knowledge and pedagogical content knowledge about water in environmental systems”. In: *Journal of Research in Science Teaching* 55.9 (2018), pp. 1339–1362. DOI: https://doi.org/10.1002/tea.21454. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/tea.21454.

[Ger91] Morton Ann Gernsbacher. “Cognitive processes and mechanisms in language comprehension : the structure building framework”. In: *Psychology of Learning and Motivation* 27 (1991), pp. 217–263.

[Gin03] David Ginat. “Seeking or Skipping Regularities? Novice Tendencies and the Role of Invariants”. In: *Informatics in Education* 2 (2003), pp. 211–222.

[Gin04] David Ginat. “On Novice Loop Boundaries and Range Conceptions”. In: *Computer Science Education* 14 (Sept. 2004), pp. 165–181. DOI: 10.1080/0899340042000302709.

[Gin18] David Ginat. “Algorithmic Cognition and Pencil-Paper Tasks”. In: *Olympiads in Informatics* 12 (May 2018), pp. 43–52. DOI: 10.15388/ioi.2018.04.

[Gin97] Herbert Ginsburg. *Entering the Child’s Mind: The Clinical Interview In Psychological Research and Practice*. Cambridge books online. Cambridge University Press, 1997.

[GM07] Anabela Gomes and Antonio Mendes. “Learning to program - difficulties and solutions”. In: *International Conference on Engineering Education – ICEE*. Jan. 2007, pp. 283–287.

[Gol+08] Ken Goldman et al. “Identifying important and difficult concepts in introductory computing courses using a delphi process”. eng. In: *Proceedings of the 39th SIGCSE technical symposium on computer science education*. SIGCSE ’08. ACM, 2008, pp. 256–260.

[Gol+10] Ken Goldman et al. “Setting the Scope of Concept Inventories for Introductory Computing Subjects”. eng. In: *ACM Transactions on Computing Education* 10.2 (2010).

[Gom+20] Anabela Gomes et al. “Study methods in introductory programming courses”. In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. 2020, pp. 898–904. DOI: 10.1109/EDUCON45650.2020.9125228.

[GP18a] Shuchi Grover and Roy Pea. “Computational Thinking: A Competency Whose Time Has Come”. In: *Computer Science Education: Perspectives on teaching and learning in school*. Ed. by S. Sentance, E. Barendsen, and S. Carsten. London, UK: Bloomsbury Academic, 2018, pp. 19–38.

[GP18b] Shuchi Grover and Roy Pea. “Computational Thinking: A Competency Whose Time Has Come”. In: *Computer Science Education: Perspectives on teaching and learning in school*. Ed. by S. Sentance, E. Barendsen, and S. Carsten. London, UK: Bloomsbury Academic, 2018, pp. 19–38.

[Gri02] David Gries. “Where is Programming Methodology These Days?” In: *SIGCSE Bull.* 34.4 (Dec. 2002), pp. 5–7. DOI: 10.1145/820127.820129.

[Gro+17] Shuchi Grover et al. “A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments”. In: *ACM Transactions on Computing Education (TOCE)* 17.3 (2017), pp. 1–25.

[Gro89] Pamela L. Grossman. “A study in contrast: sources of pedagogical content knowledge for secondary English”. In: *Journal of Teacher Education* 40.5 (1989), pp. 24–31.

[Hab04] Bruria Haberman. “High-School Students’ Attitudes Regarding Procedural Abstraction”. In: *Education and Information Technologies* 9.2 (May 2004), pp. 131–145. DOI: 10.1023/B:EAIT.0000027926.99053.6f.

[Har96] Geoff Hart. “The Five W’s: An Old Tool for the New Task of Audience Analysis.” In: *Technical Communication: Journal of the Society for Technical Communication* 43 (1996).

[Hat12] John Hattie. *Visible learning for teachers: Maximizing impact on learning.* Routledge, 2012.

[Haz03] Orit Hazzan. “How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science”. In: *Computer Science Education* 13.2 (2003), pp. 95–122. DOI: 10.1076/csed.13.2.95.14202.

[Haz08] Orit Hazzan. “Reflections on Teaching Abstraction and Other Soft Ideas”. In: *SIGCSE Bull.* 40.2 (June 2008), pp. 40–43. DOI: 10.1145/1383602.1383631.

[Haz99] Orit Hazzan. “Reducing Abstraction Level When Learning Abstract Algebra Concepts”. In: *Educational Studies in Mathematics* 40.1 (1999), pp. 71–90.

[Her10] Matthew Hertz. “What Do "CS1" and "CS2" Mean? Investigating Differences in the Early Courses”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education.* SIGCSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 199–203. DOI: 10.1145/1734263.1734335.

[Her+12] Geoffrey L. Herman et al. “Describing the What and Why of Students’ Difficulties in Boolean Logic”. In: *ACM Trans. Comput. Educ.* 12.1 (2012). DOI: 10.1145/2133797.2133800.

[HH82] Brian C. Hansford and John A. Hattie. “The relationship between self and achievement/performance measures”. In: *Review of Educational Research* 52.1 (1982), pp. 123–142.

[HK99] Mary Hegarty and Maria Kozhevnikov. “Types of visual–spatial representations and mathematical problem solving.” In: *Journal of educational psychology* 91.4 (1999), p. 684.

[HLR11] Orit Hazzan, Tami Lapidot, and Noa Ragonis. *Guide to Teaching Computer Science: An Activity-Based Approach.* 1st. Springer Publishing Company, Incorporated, 2011.

[HT05] Orit Hazzan and James Tomayko. “Reflection and abstraction in learning software engineering’s human aspects”. In: *Computer* 38 (June 2005), pp. 39–45. DOI: 10.1109/MC.2005.200.

[HT07] John Hattie and Helen Timperley. “The power of feedback”. In: *Review of educational research* 77.1 (2007), pp. 81–112.

[Ian04] Dario Ianes. *Metacognizione e insegnamento: spunti teorici e applicativi.* Italian. OCLC: 1020164396. Trento: Centro studi Erickson, 2004.

[IM20] Cruz Izu and Claudio Mirolo. “Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers”. In: *Proc. of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’20. New York, NY, USA: ACM, 2020, pp. 466–472.

[IMW18] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. “Novice Programmers’ Reasoning About Reversing Conditional Statements”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* SIGCSE ’18. New York, NY, USA: ACM, 2018, pp. 646–651. DOI: 10.1145/3159450.3159499.

[IPW17] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. “On the Ability to Reason About Program Behaviour: A Think-Aloud Study”. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’17. New York, USA: ACM, 2017, pp. 305–310. DOI: 10.1145/3059009.3059036.

[IPW19] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. “Up or Down? An Insight into Programmer’s Acquisition of Iteration Skills”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* SIGCSE ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 941–947. DOI: 10.1145/3287324.3287350.

[IST11] ISTE/CSTA Steering Committee. *Computational Thinking Teacher Resources, 2nd ed.* ISTE/CSTA. retrieved: april 2022. 2011.

[IWP16] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. “A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research.* ICER ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 251–259. DOI: 10.1145/2960310.2960324.

[Izu+19] Cruz Izu et al. “Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories”. In: *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education.* ITiCSE-WGR ’19. New York, NY, USA: ACM, 2019, pp. 27–52.

[Jen02] Tony Jenkins. “On the Difficulty of Learning to Program”. In: *Proceedings of the 3rd annual LTSN ICS Conference.* Loughborough, UK, 2002.

[JL83] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness.* 6. Harvard University Press, 1983.

[Kac+10] Lisa C. Kaczmarczyk et al. “Identifying Student Misconceptions of Programming”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. New York, NY, USA: ACM, 2010, pp. 107–111.

[Kap16] Manu Kapur. “Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning”. In: *Educational Psychologist* 51 (Apr. 2016), pp. 1–11.

[KD03] Amruth Kumar and Garrett Dancik. “A tutor for counter-controlled loop concepts and its evaluation”. In: *33rd Annual Frontiers in Education, 2003. FIE 2003.* Vol. 1. Nov. 2003, T3C–7. DOI: 10.1109/FIE.2003.1263331.

[KD10] Herman Koppelman and Betsy van Dijk. “Teaching Abstraction in Introductory Courses”. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 174–178. DOI: 10.1145/1822090.1822140.

[KD78] Walter Kintsch and Teun A. van Dijk. “Toward a model of text comprehension and production.” en. In: *Psychological Review* 85.5 (1978), pp. 363–394. DOI: 10.1037/0033-295X.85.5.363.

[Kes19] Max Kesselbacher. “Supporting the Acquisition of Programming Skills with Program Construction Patterns”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 188–189. DOI: 10.1109/ICSE-Companion.2019.00077.

[Knu76] Donald E. Knuth. “Mathematics and Computer Science: Coping with Finiteness”. eng. In: *Science* 194.4271 (1976), pp. 1235–1242.

[Kon19] Siu-cheung Kong. “Components and Methods of Evaluating Computational Thinking for Fostering Creative Problem-Solvers in Senior Primary School Education”. In: *Computational thinking education*. Springer, Singapore, May 2019, pp. 119–141. DOI: 10.1007/978-981-13-6528-7_8.

[Kra07] Jeff Kramer. “Is abstraction the key to computing?” In: *Commun. ACM* 50 (Apr. 2007), pp. 36–42. DOI: 10.1145/1232743.1232745.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A Study of the Difficulties of Novice Programmers”. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 14–18. DOI: 10.1145/1067445.1067453.

[LDC20] Elise Lockwood and Adaline De Chenne. "Enriching Students' Combinatorial Reasoning through the Use of Loops and Conditional Statements in Python". In: *International Journal of Research in Undergraduate Mathematics Education* 6 (Oct. 2020). DOI: 10.1007/s40753-019-00108-2.

[Lew+05a] Gary Lewandowski et al. "What novice programmers don't know". eng. In: *Proceedings of the first international workshop on computing education research*. ICER '05. ACM, 2005, pp. 1–12.

[Lew+05b] Gary Lewandowski et al. "What Novice Programmers Don'T Know". In: *Proceedings of the First International Workshop on Computing Education Research*. ICER '05. New York, NY, USA: ACM, 2005, pp. 1–12. DOI: 10.1145/1089786.1089787.

[LGG86] Gaea Leinhardt and James G. Greeno. "The Cognitive Skill of Teaching". In: *Journal of Educational Psychology* 78 (Apr. 1986), pp. 75–95. DOI: 10.1037/0022-0663.78.2.75.

[Lib08] Julie Libarkin. *Concept Inventories in Higher Education Science*. Jan. 2008.

[Lis+04] Raymond Lister et al. "A Multi-national Study of Reading and Tracing Skills in Novice Programmers". In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '04. New York, NY, USA: ACM, 2004, pp. 119–150. DOI: 10.1145/1044550.1041673.

[Lis+06] Raymond Lister et al. "Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy". In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE '06. New York, NY, USA: ACM, 2006, pp. 118–122.

[Lis11] Raymond Lister. "Concrete and other neo-piagetian forms of reasoning in the novice programmer". In: *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), pp. 9–18.

[Lis+12] Raymond Lister et al. "Toward a Shared Understanding of Competency in Programming: An Invitation to the BABELnot Project". In: *Proceedings of the 14th Australasian Computing Education Conference (ACE 2012)*. Ed. by Michael de Raadt and Angela Carbone. RMIT University, Melbourne: Australian Computer Society, Jan. 2012.

[LMB04] John Loughran, Pamela Mulhall, and Amanda Berry. "In Search of Pedagogical Content Knowledge in Science: Developing Ways of Articulating and Documenting Professional Practice". In: *Journal of Research in Science Teaching* 41 (Apr. 2004), pp. 370 –391. DOI: `10.1002/tea.20007`.

[LMB08] John Loughran, Pamela Mulhall, and Amanda Berry. "Exploring Pedagogical Content Knowledge in Science Teacher Education". In: *International Journal of Science Education - INT J SCI EDUC* 30 (Aug. 2008), pp. 1301–1320. DOI: `10.1080/09500690802187009`.

[Lop+08] Mike Lopez et al. "Relationships Between Reading, Tracing and Writing Skills in Introductory Programming". In: *Proc. 4th Int. Workshop on Comput. Educ. Research*. ICER '08. New York, USA: ACM, 2008, pp. 101–112.

[LP15] Sally I. Lipsey and Bernard S. Pasternack. "Mathematics in Literature". In: (2015).

[LR+17a] Andrew Luxton-Reilly et al. "Developing Assessments to Determine Mastery of Programming Fundamentals". In: *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. ITiCSE-WGR '17. New York, USA: ACM, 2017, pp. 47–69. DOI: `10.1145/3174781.3174784`.

[LR+17b] Andrew Luxton-Reilly et al. "Developing Assessments to Determine Mastery of Programming Fundamentals". In: *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. ITiCSE-WGR '17. New York, NY, USA: ACM, 2017, pp. 47–69. DOI: `10.1145/3174781.3174784`.

[LR+18] Andrew Luxton-Reilly et al. "Introductory Programming: A Systematic Literature Review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. New York, NY, USA: ACM, 2018, pp. 55–106.

[LRR17] Katharina Loibl, Ido Roll, and Nikol Rummel. "Towards a Theory of When and How Problem Solving Followed by Instruction Supports Learning". In: *Educational Psychology Review* 29.4 (Dec. 2017), pp. 693–715.

[Ma+09] Linxiao Ma et al. "Improving the Mental Models Held by Novice Programmers Using Cognitive Conflict and Jeliot Visualisations". In: *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 166–170. DOI: `10.1145/1562877.1562931`.

[Man+20] Linda Mannila et al. “Programming in Primary Education: Towards a Research Based Assessment Framework”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. WiPSCE ’20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3421590.3421598.

[Mas89] John Mason. “Mathematical abstraction as the result of a delicate shift of attention”. In: *Learn. Math.* 9.2 (1989), pp. 2–8.

[May14] Philipp Mayring. *Qualitative Content Analysis: Theoretical Foundation, Basic Procedures and Software Solution*. Klagenfurt, 2014.

[MBŽ18] Monika Mladenovic, Ivica Boljat, and Žana Žanko. “Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level”. In: *Education and Information Technologies* 23 (July 2018), pp. 1483–1500. DOI: 10.1007/s10639-017-9673-3.

[McK00] Peter McKenna. “Transparent and opaque boxes: do women and men have different computer programming psychologies and styles?” In: *Computers & Education* 35.1 (2000), pp. 37–49. DOI: 10.1016/S0360-1315(00)00017-8.

[Mea91] H. Willis Means. “Using Literature in a Computer Science Service Course: Improving Abstract/Critical Thinking Skills”. In: *J. Comput. Sci. Coll.* 6.5 (Apr. 1991), pp. 30–34.

[Men+16] Chiara Meneghetti et al. “The role of visual and spatial working memory in forming mental models derived from survey and route descriptions”. In: *British journal of psychology (London, England : 1953)* 108 (Mar. 2016). DOI: 10.1111/bjop.12193.

[Men+20] Chiara Meneghetti et al. *Nuova guida alla comprensione del testo 1: Introduzione teorica generale al programma. Le prove criteriali livello A e B*. Edizioni Centro Studi Erickson, 2020.

[Met17] Janet Metcalfe. “Learning from Errors”. In: *Annual Review of Psychology* 68 (Jan. 2017), pp. 465–489.

[MHD09] Shuhaida Mohamed Shuhidan, Margaret Hamilton, and Daryl D’Souza. “A Taxonomic Study of Novice Programming Summative Assessment”. In: *Proc. 11th Australasian Conf. on Computing Education - Volume 95*. ACE ’09. Darlinghurst, Australia: Australian Computer Society, Inc., 2009, pp. 147–156.

[MI19] Claudio Mirolo and Cruz Izu. “An Exploration of Novice Programmers’ Comprehension of Conditionals in Imperative and Functional Programming”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 436–442. DOI: 10.1145/3304221.3319746.

[Mir12] Claudio Mirolo. "Is Iteration Really Easier to Learn Than Recursion for CS1 Students?" In: *Proc. of the 9th Annual International Conference on International Computing Education Research.* ICER '12. New York, NY, USA: ACM, 2012, pp. 99–104.

[MIS20] Claudio Mirolo, Cruz Izu, and Emanuele Scapin. "High-School Students' Mastery of Basic Flow-Control Constructs through the Lens of Reversibility". In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education.* WiPSCE '20. New York, NY, USA: Association for Computing Machinery, 2020. DOI: `10.1145/3421590.3421603`.

[MKB99] Shirley Magnusson, Joseph Krajcik, and Hilda Borko. "Nature, Sources, and Development of Pedagogical Content Knowledge for Science Teaching". In: *Examining Pedagogical Content Knowledge.* Ed. by Julie Gess-Newsome and NormanG. Lederman. Vol. 6. Science & Technology Education Library. Springer Netherlands, 1999, pp. 95–132. DOI: `10.1007/0-306-47217-1\_4`.

[MS21] Craig S. Miller and Amber Settle. "Mixing and Matching Loop Strategies: By Value or By Index?" In: *Proc. of the 52nd SIGCSE.* SIGCSE '21. Virtual Event, USA, 2021, pp. 1048–1054.

[MS68] H. Edward Massengill and Emir H. Shuford. *The effect of 'Degree of Confidence' in student testing.* eng. Tech. rep. 1968.

[MSABA10] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. "Learning Computer Science Concepts with Scratch". In: *Proceedings of the Sixth International Workshop on Computing Education Research.* ICER '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 69–76. DOI: `10.1145/1839594.1839607`.

[Mye90] J.Paul Myers. "The Central Role of Mathematical Logic in Computer Science". In: *ACM SIGCSE Bulletin* 22.1 (1990), pp. 22–26.

[Nar20] Enrico Nardelli. *Coding e oltre. L'informatica nella scuola.* Liscianilibri, 2020.

[Nut07] Graham Nuthall. *The Hidden Lives of Learners.* NZCER Press, 2007.

[NV+21] Jacqueline Nijenhuis-Voogt et al. "Teaching algorithms in upper secondary education: a study of teachers' pedagogical content knowledge". In: *Computer Science Education* 0.0 (2021), pp. 1–33. DOI: `10.1080/08993408.2021.1935554`. eprint: `https://doi.org/10.1080/08993408.2021.1935554`.

[Oak84] Jane V. Oakhill. "Inferential And Memory Skills In Children's Comprehension Of Stories". In: *British Journal of Educational Psychology* 54 (1984), pp. 31–39.

[Odi00] Piergiorgio Odifreddi. “Metodi Matematici Della Letteratura”. In: *Nuova Civiltà Delle Macchine* 18.3 (2000), pp. 116–133.

[Pai13] Allan Paivio. *Imagery and Verbal Processes.* English. OCLC: 869091762. Hoboken: Taylor and Francis, 2013.

[PCL84] Scott G. Paris, David R. Cross, and Marjorie Y. Lipson. “Informed strategies for learning: A program to improve children’s reading awareness and comprehension”. In: *Journal of Educational psychology* 76.6 (1984), pp. 1239–1252.

[Pea86] Roy D. Pea. “Language-Independent Conceptual “Bugs” in Novice Programming”. In: *Journal of Educational Computing Research* 2 (1986), pp. 25–36.

[Pen87] Nancy Pennington. “Comprehension Strategies in Programming”. In: *Empirical Studies of Programmers: Second Workshop.* USA: Ablex Publishing Corp., 1987, pp. 100–113.

[Per+86] David N. Perkins et al. “Conditions of learning in novice programmers”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 37–55.

[Pet+20] Emily Grossnickle Peterson et al. “Spatial activity participation in childhood and adolescence: consistency and relations to spatial thinking in adolescence”. In: *Cognitive Research: Principles and Implications* 5.1 (2020), pp. 1–13.

[PF18] Amanda Peel and Patricia Friedrichsen. “Algorithms, Abstractions, and Iterations: Teaching Computational Thinking Using Protein Synthesis Translation”. In: *The American Biology Teacher* 80 (Jan. 2018), pp. 21–28. DOI: 10.1525/abt.2018.80.1.21.

[Pia+69] Jean Piaget et al. *Psychology Of The Child.* The Psychology of the Child. Basic Books, 1969.

[Pie17] Aleksander Piecuch. “Zaniedbana algebra a nauczanie informatyki”. eng. In: *Edukacja - Technika - Informatyka* VIII.3 (2017), pp. 288–295.

[PRW07] Anne Philpott, Phil Robbins, and J Whalley. “Assessing the steps on the road to relational thinking”. In: *Proceedings of the 20th annual conference of the National Advisory Committee on Computing Qualifications.* Vol. 286. 2007.

[PSS88] D.N. Perkins, Steve Schwartz, and Rebecca Simmons. “Instructional Strategies for the Problems of Novice Programmers”. In: *Teaching and Learning Computer Programming.* Ed. by Richard E. Mayer. New York, USA: Routledge, 1988, pp. 153–178. DOI: 10.4324/9781315044347.

[QL17] Yizhou Qian and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review”. In: *ACM Trans. Comput. Educ.* 18.1 (Oct. 2017). DOI: 10.1145/3077618.

[Qui14] Margareth Quindeless. “Logic in the curricula of Computer Science”. spa. In: *Revista AntioqueÃ+a de las Ciencias Computacionales y la IngenierÃ-a de Software (RACCIS)* 4.2 (2014), pp. 47–51.

[RAVD83] H. Rudy Ramsey, Michael E. Atwood, and James R. Van Doren. “Flowcharts versus Program Design Languages: An Experimental Comparison”. In: *Commun. ACM* 26.6 (June 1983), pp. 445–449. DOI: 10.1145/358141.358149.

[RBH16] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. “Typifying informatics teachers’ PCK of designing digital artefacts in Dutch upper secondary education”. In: *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives.* Springer. 2016, pp. 65–77.

[RBH17] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. “Identifying Students’ Misconceptions on Basic Algorithmic Concepts Through Flowchart Analysis”. In: *Informatics in Schools: Focus on Learning Programming.* Ed. by Valentina Dagienė and Arto Hellas. Cham: Springer International Publishing, 2017, pp. 155–168.

[RDLR20] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. “A Miss is as Good as a Mile: Off-By-One Errors and Arrays in an Introductory Programming Course”. In: *Proc. of the 22nd Australasian Computing Education Conference.* 2020, pp. 31–38.

[Rep16] Alexander Repenning. “Transforming "Hard and Boring" into "Accessible and Exciting"”. In: *CoPDA@NordiCHI.* 2016.

[RHG06] Anthony Robins, Patricia Haden, and Sandy Garner. “Problem Distributions in a CS1 Course”. In: *Proc. of the 8th Australasian Conference on Computing Education - Volume 52.* ACE ’06. Darlinghurst, Australia: Australian Computer Society, Inc., 2006, pp. 165–173.

[Rij+18] Wouter J. Rijke et al. “Computational thinking in primary school: An examination of abstraction and decomposition in different age groups”. In: *Informatics in education* 17.1 (2018), pp. 77–92.

[RM87] Lauren B. Resnick and S.T.E. Committee on Research in Mathematics. *Education and Learning to Think.* Online access: National Academy of Sciences National Academies Press. National Academies Press, 1987.

[Rob19] Anthony V. Robins. “Novice Programmers and Introductory Programming”. In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 327–376. DOI: 10.1017/9781108654555.013.

[RRR03] Anthony Robins, Janet Rountree, and Nathan Rountree. “Learning and Teaching Programming: A Review and Discussion”. In: *Computer Science Education* 13.2 (2003), pp. 137–172.

[RW88] Charles Rich and Richard Waters. “The Programmer’s Apprentice Project: A Research Overview”. In: *Computer* 21 (Dec. 1988), pp. 10 –25. DOI: 10.1109/2.86782.

[SA16] David Statter and Michal Armoni. “Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders”. In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. WiPSCE ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 80–83. DOI: 10.1145/2978249.2978261.

[Sad10] D. Royce Sadler. “Beyond feedback: Developing student capability in complex appraisal”. In: *Assessment & evaluation in higher education* 35.5 (2010), pp. 535–550.

[Sad89] D. Royce Sadler. “Formative assessment and the design of instructional systems”. In: *Instructional science* 18.2 (1989), pp. 119–144.

[Sae+11] Mara Saeli et al. “Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective”. In: *Informatics in Education* 10 (Apr. 2011), pp. 73–88.

[Sae12] Mara Saeli. “Teaching programming for secondary school : a pedagogical content knowledge based approach”. English. Proefschrift. PhD thesis. Eindhoven School of Education, 2012. DOI: 10.6100/IR724491.

[SB06] Carsten Schulte and Jens Bennedsen. “What do teachers teach in introductory programming?” eng. In: *Proceedings of the second international workshop on computing education research*. Vol. 2006. ICER ’06. ACM, 2006, pp. 17–28.

[SB07] Bharath Sriraman and Astrid Beckmann. *Mathematics and Literature: Perspectives for interdisciplinary classroom pedagogy*. Jan. 2007.

[SBE83] Elliot M. Soloway, Jeffrey Bonar, and Kate Ehrlich. “Cognitive Strategies and Looping Constructs: An Empirical Study”. In: *Commun. ACM* 26.11 (Nov. 1983), pp. 853–860. DOI: 10.1145/182.358436.

[SBGM02] Dominique MA Sluijsmans, Saskia Brand-Gruwel, and Jeroen JG van Merriënboer. “Peer assessment training in teacher education: Effects on performance and perceptions”. In: *Assessment & Evaluation in Higher Education* 27.5 (2002), pp. 443–454.

[Sch08] Carsten Schulte. “Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching”. In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER ’08. New York, NY, USA: ACM, 2008, pp. 149–160. DOI: 10.1145/1404520.1404535.

[Sch12] Dane Schaffer. “An Analysis of Science Concept Inventories and Diagnostic Tests: Commonalities and Differences”. In: *Annual International Conference of the National Association for Research in Science Teaching*. Apr. 2012.

[Sch+13] Stephan Schmelzing et al. “Development, evaluation, and validation of a paper-and-pencil test for measuring two components of biology teachers’ pedagogical content knowledge concerning the “cardiovascular system””. In: *International Journal of Science and Mathematics Education* 11 (Dec. 2013). DOI: 10.1007/s10763-012-9384-6.

[Sch98] Gregory Schraw. “Promoting general metacognitive awareness”. In: *Instructional science* 26.1 (1998), pp. 113–125.

[SG11] Andreas Stefik and Ed Gellenbeck. “Empirical studies on programming language stimuli”. In: *Software Quality Journal* 19 (Mar. 2011), pp. 65–99. DOI: 10.1007/s11219-010-9106-7.

[SG79] Nancy Stein and Christine Glenn. “An Analysis of Story Comprehension in Elementary School Children”. In: *New Directions in Discourse Processing* 2 (Jan. 1979).

[SGK20] Bernadette Spieler, Maria Grandl, and Vesna Krnjic. “The hAPPy-Lab: A Gender-Conscious Way To Learn Coding Basics in an Open Makerspace Setting”. In: *Proceedings of the International Conference on Informatics in School: Situation, Evaluation and Perspectives, Tallinn, Estonia, November 16-18, 2020*. Ed. by Külli Kori and Mart Laanpere. Vol. 2755. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 64–75.

[SH08] Victoria Sakhnini and Orit Hazzan. “Reducing Abstraction in High School Computer Science Education: The Case of Definition, Implementation, and Use of Abstract Data Types”. In: *J. Educ. Resour. Comput.* 8.2 (May 2008). DOI: 10.1145/1362787.1362789.

[Shu05a] Lee S. Shulman. *Signature pedagogies*. 2005.

[Shu05b] Lee S. Shulman. “Teacher education does not exist”. In: *Stanford Educator* 7 (2005).

[Shu86] Lee S. Shulman. “Those Who Understand: Knowledge Growth in Teaching”. eng. In: *Educational Researcher* 15.2 (Feb. 1986), pp. 4–14.

[Sim+06] Beth Simon et al. “Commonsense Computing: What Students Know before We Teach (Episode 1: Sorting)”. In: *Proceedings of the Second International Workshop on Computing Education Research.* ICER ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 29–40. DOI: 10.1145/1151588.1151594.

[Sim13] Simon. “Soloway’s Rainfall Problem Has Become Harder”. In: *2013 Learning and Teaching in Computing and Engineering.* 2013, pp. 130–135. DOI: 10.1109/LaTiCE.2013.44.

[Sle+86] D. Sleeman et al. “Pascal and High School Students: A Study of Errors”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 5–23. DOI: 10.2190/2XPP-LTYH-98NQ-BU77. eprint: https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77.

[SM19] Emanuele Scapin and Claudio Mirolo. “An Exploration of Teachers’ Perspective About the Learning of Iteration-Control Constructs”. In: *Informatics in Schools. New Ideas in School Informatics.* Ed. by Sergei N. Pozdniakov and Valentina Dagienė. Cham: Springer, 2019, pp. 15–27.

[SM20] Emanuele Scapin and Claudio Mirolo. “An Exploratory Study of Students’ Mastery of Iteration in the High School”. In: *Proceedings of the International Conference on Informatics in School: Situation, Evaluation and Perspectives, Tallinn, Estonia, November 16-18, 2020.* Ed. by Külli Kori and Mart Laanpere. Vol. 2755. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 43–54.

[Sor13] Juha Sorva. “Notional Machines and Introductory Programming Education”. In: *Trans. Comput. Educ.* 13.2 (2013), 8:1–8:31.

[SPV20] Phil Steinhorst, Andrew Petersen, and Jan Vahrenhold. “Revisiting Self-Efficacy in Introductory Programming”. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research.* 2020, pp. 158–169.

[SS13] Andreas Stefik and Susanna Siebert. “An empirical investigation into programming language syntax”. In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), pp. 1–40.

[SS88] Elliot M. Soloway and James C. Spohrer. *Studying the Novice Programmer.* USA: L. Erlbaum Associates Inc., 1988.

[Ste18] Friedrich Steimann. “Fatal Abstraction”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 125–130. DOI: 10.1145/3276954.3276966.

[SW80] Elliot M. Soloway and Beverly Woolf. “Problems, Plans, and Programs”. In: *Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’80. New York, NY, USA: ACM, 1980, pp. 16–24. DOI: 10.1145/800140.804605.

[SWS17] Renske Smetsers-Weeda and Sjaak Smetsers. “Problem Solving and Algorithmic Development with Flowcharts”. In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 25–34. DOI: 10.1145/3137065.3137080.

[Tam92] Wing C. Tam. “Teaching Loop Invariants to Beginners by Examples”. In: *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’92. New York, NY, USA: ACM, 1992, pp. 92–96.

[TG10] Allison Tew and Mark Guzdial. “Developing a validated assessment of fundamental CS1 concepts”. In: Jan. 2010, pp. 97–101. DOI: 10.1145/1734263.1734297.

[Tho+06] Errol Thompson et al. “Code Classification as a Learning and Assessment Exercise for Novice Programmers”. English. In: *The 19th Annual Conference of the National Advisory Committee on Computing Qualifications*. Ed. by Samuel Mann and Noel Bridgeman. National Advisory Committee on Computing Qualifications, 2006, pp. 291–298.

[TL14a] Donna Teague and Raymond Lister. “Blinded by their Plight: Tracing and the Preoperational Programmer”. In: *PPIG*. June 2014.

[TL14b] Donna Teague and Raymond Lister. “Programming: Reading, Writing and Reversing”. In: *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’14. New York, USA: ACM, 2014, pp. 285–290. DOI: 10.1145/2591708.2591712.

[TL14c] Donna Teague and Raymond Lister. “Programming: Reading, Writing and Reversing”. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE ’14. New York, NY, USA: ACM, 2014, pp. 285–290. DOI: 10.1145/2591708.2591712.

[Tur02] Sigita Turskienė. “Computer Technology and Teaching Mathematics in Secondary Schools”. eng. In: *Informatics in Education - An International Journal* 1.1 (2002), pp. 149–156.

[Van+10] Tammy Vandegrift et al. “Commonsense computing (episode 6): Logic is harder than pie”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling’10* (Jan. 2010). DOI: `10.1145/1930464.1930479`.

[VDB12] Jan H. Van Driel and Amanda Berry. “Teacher Professional Development Focusing on Pedagogical Content Knowledge”. eng. In: *Educational Researcher* 41.1 (2012), pp. 26–28.

[VDK83] Teun Adrianus Van Dijk and Walter Kintsch. *Strategies of discourse comprehension.* 1983.

[Ver11] Tom Verhoeff. “On Abstraction and Informatics”. In: *Informatics in Schools. Contributing to 21st Century Education: 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives, ISSEP 2011, Bratislava, Slovakia, October 26-29, 2011. Proceedings.* Ed. by Ivan Kalaš and Roland T. Mittermeir. 2011, pp. 1–12. DOI: `www.issep2011.org`.

[VS07] Vesa Vainio and Jorma Sajaniemi. “Factors in novice programmers’ poor tracing skills”. In: *SIGCSE Bull.* 39.3 (2007), pp. 236–240. DOI: `10.1145/1269900.1268853`.

[VS11] André Vandierendonck and Arnaud Szmalec. *Spatial working memory.* Jan. 2011.

[VTL09] Anne Venables, Grace Tan, and Raymond Lister. “A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer”. In: *Proceedings of the Fifth International Workshop on Computing Education Research Workshop.* ICER ’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–128. DOI: `10.1145/1584322.1584336`.

[VVMS99] A. Marie Vans, Anneliese Von Mayrhauser, and Gabriel Somlo. “Program understanding behavior during corrective maintenance of large-scale software”. In: *International Journal of Human-Computer Studies* 51.1 (1999), pp. 31–70. DOI: `https://doi.org/10.1006/ijhc.1999.0268`.

[Wan08] Yingxu Wang. “A Hierarchical Abstraction Model for Software Engineering”. In: *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering.* ROA ’08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 43–48. DOI: `10.1145/1370164.1370174`.

[WD97] Roland Wagner-Dobler. “Science-Technology Coupling: The Case of Mathematical Logic and Computer Science.” eng. In: *Journal of the American Society for Information Science* 48.2 (1997), pp. 171–83.

[Wie89] Susan Wiedenbeck. “Learning iteration and recursion from examples”. In: *International Journal of Man-Machine Studies* 30.1 (1989), pp. 1 –22. DOI: `https://doi.org/10.1016/S0020-7373(89)80018-5`.

[Win11] Jeannette M. Wing. “Computational Thinking: What and Why?” In: *The Link Magazine* (2011).

[Win96] Leon E. Winslow. “Programming pedagogy—a psychological overview”. In: *ACM Sigcse Bulletin* 28.3 (1996), pp. 17–22.

[WLB09] Jonathan Wai, David Lubinski, and Camilla P. Benbow. “Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance.” en. In: *Journal of Educational Psychology* 101.4 (2009), pp. 817–835. DOI: `10.1037/a0016127`.

[WM99] Paul White and Michael Mitchelmore. “Learning mathematics: A New Look at Generalisation and Abstraction”. In: *AARE Annual Conference.* AARE ’99. deakin, ACT, Australia: Australian Association for Research in Education, 1999, pp. 1–12.

[Woo06] Terry Wood. “Teacher Education Does Not Exist”. eng. In: *Journal of Mathematics Teacher Education* 9.1 (2006), pp. 1–3.

[WR99] Susan Wiedenbeck and Vennila Ramalingam. “Novice Comprehension of Small Programs Written in the Procedural and Object-Oriented Styles”. In: *Int. J. Hum.-Comput. Stud.* 51.1 (July 1999), pp. 71–87. DOI: `10.1006/ijhc.1999.0269`.

[WSR87] Suzanne Wilson, Lee S. Shulman, and AE Richert. “" 150 different ways" of knowing: Representations of knowledge in teaching”. In: *Exploring Teachers’ Thinking* (Jan. 1987), pp. 104–124.

[WW15] David Weintrop and Uri Wilensky. “Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research.* ICER ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 101–110. DOI: `10.1145/2787622.2787721`.

[YK07] Gavriel Yarmish and Danny Kopec. “Revisiting Novice Programmer Errors”. In: *SIGCSE Bull.* 39.2 (June 2007), pp. 131–137. DOI: `10.1145/1272848.1272896`.

[ZH98] Rina Zazkis and Orit Hazzan. “Interviewing in mathematics education research: Choosing the questions”. eng. In: *Journal of Mathematical Behavior* 17.4 (1998), pp. 429–439.

[Zim00] Barry J. Zimmerman. “Self-efficacy: An essential motive to learn”. In: *Contemporary educational psychology* 25.1 (2000), pp. 82–91.

[ZR08] Christopher Zaleta and Kim Ruebel. "Exploring Mathematical Concepts in Literature". In: *Middle School Journal* 40 (2008), pp. 36–42.