The Landscape of Computational Thinking Problems for Practice and Assessment

RADEK PELÁNEK, Masaryk University, Czech Republic
TOMÁŠ EFFENBERGER, Masaryk University, Czech Republic

To provide practice and assessment of computational thinking, we need specific problems students can solve. There are many such problems, but they are hard to find. Learning environments and assessments often use only specific types of problems and thus do not cover computational thinking in its whole scope. We provide an extensive catalog of well-structured computational thinking problem sets together with a systematic encoding of their features. Based on this encoding, we propose a four-level taxonomy that provides an organization of a wide variety of problems. The catalog, taxonomy, and problem features are useful for content authors, designers of learning environments, and researchers studying computational thinking.

CCS Concepts: • **Social and professional topics** → **Computational thinking**; *K-12 education*; *CS1*; *Student assessment*; • **Applied computing** → *Interactive learning environments*.

Additional Key Words and Phrases: skills, well-structured problems, taxonomy

ACM Reference Format:
Radek Pelánek and Tomáš Effenberger. 2023. The Landscape of Computational Thinking Problems for Practice and Assessment. *ACM Trans. Comput. Educ.* X, Y, Article Z (January 2023), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Computational thinking is an important skill that is currently getting significant attention in both research and educational practice. The scope and definition of computational thinking are still being settled. However, there is a general agreement on the basic components of computational thinking, which include algorithmic thinking, abstraction and decomposition, testing and debugging, parallelism, and pattern recognition (see, e.g., [20, 25, 55, 65]).

Even though computational thinking is typically described with a rather broad scope, specific learning tools and research studies often focus on just a single type of activity, e.g., block-based programming in Scratch [39] or multiple-choice questions about programs [51]. Some activities (e.g., block-based programming in microworlds) are used repeatedly but often scattered in various publication venues, learning environments, and apps under varied terminology. Researchers often pick one of these common activities and analyze it in detail [23] or study the convergence validity of two activities [50]. Such results are useful, but they need to be considered and interpreted in the broader context of other computational thinking activities.

For further development of learning environments and assessment tools for computational thinking, we need a large database of specific problems and an organizational structure that will

Authors' addresses: Radek Pelánek, Masaryk University, Brno, Czech Republic, xpelanek@fi.muni.cz; Tomáš Effenberger, Masaryk University, Brno, Czech Republic, tomas.effenberger@mail.muni.cz.

1946-6226/2023/1-ARTZ $15.00
https://doi.org/XXXXXXX.XXXXXXX

allow us to use the database efficiently—a recent systematic review on assessing computational thinking concluded that such a database and organization is currently missing [60].

In this work, we aim to cover this gap by mapping a diverse landscape of computational thinking problems. Specifically, we focus on well-structured computational thinking problems intended for beginners. Well-structured problems are problems that have clear problem statements and evaluation criteria; we consider only problems that can be automatically evaluated by the learning environment (e.g., problems with a unique correct answer or with formal goal criteria that can be verified by a program). The well-structured problems that we consider are directly relevant to both practice and assessment. We acknowledge that many activities for computational thinking are not well-structured, e.g., open-ended creative activities in Scratch [67] or CS unplugged activities [15]. We do not make any claims about the relative benefits of well-structured and ill-structured problems; both are clearly useful and should be combined. Although we consider only problems aimed at beginners, we use a rather broad conception of beginners, including both K-12 students and students in introductory university courses (particularly non-majors). Specific examples of the usage of the considered type of problem are the international Bebras challenge [12, 13] and Hour of Code activities [66].

We provide a mapping of the landscape of computational thinking problems. We propose a taxonomy (problems, problem sets, problem families, problem classes) that provides an organization of the many available problems. The taxonomy is based on a systematic mapping of problem features that encode skills addressed by problems, the notation and mode of interaction, and the target audience. We provide an extensive catalog of problem sets, together with the mapping of their features.

The provided results are useful in several ways. For designers and content authors of learning environments and assessments of computational thinking, they provide awareness of a wide range of existing problems for practice and assessment, a tool for orientation and organization of problems, and inspiration for the creation of novel problems (e.g., by combining problem features that are currently not used together). We also provide descriptions and illustrations of an extensive set of problem sets (Appendix B), which give specific inspiration for the designers and content authors.

For researchers, our work provides a way to contextualize the exploration of validity and reliability of computational thinking assessments, which typically involve only some types of problems. Our analysis of the current computational thinking research shows a predominance of single problem class ("code interpretation") and motivates research based on other types of problems. The presented taxonomy of problems also makes predictions about the transfer of learning (practice on one problem type leading to improved performance on a different problem type). This can serve as a foundation for future research.

2 METHOD

This section describes our approach to mapping the landscape of computational thinking problems. We describe the terminology that we use to discuss problems at different levels of abstraction. We justify our choice of analyzed computational thinking skills and describe the iterative process used to perform this study.

2.1 Terminology: Problem Taxonomy

Computational thinking and educational technology are currently discussed from many perspectives using different terminology [46]. Therefore, we start by clarifying the terminology used in this paper, specifically concerning *problems*, which we explore at several levels of abstraction.

There are many other terms that could be considered for describing the used levels of abstraction (e.g., group, category, type, genre, kind). We have chosen a terminology analogous to biological

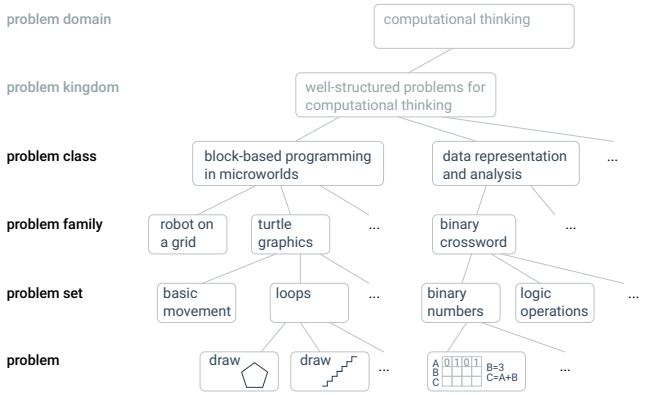


Fig. 1. Illustration of terms used to describe a taxonomy of computational thinking problems

taxonomic ranks: individual (problem), species (set), family, and class. One reason why we choose this terminology is that it communicates the ordering of levels of abstraction more clearly than other alternatives that we considered. Figure 1 provides an illustration of the used terms.

- A *problem* is one specific instance. Other terms used as synonyms for a problem are a question, an item, an exercise, or a challenge. Example: "draw a pentagram using turtle graphics and block-based programming."
- A *problem set* is a set of specific instances that share the same basic principles and presentation aspects and are meant to be used together. They differ primarily in instantiation (e.g., specific values or goals). For the purpose of practice or assessment, they are mostly replaceable (we expect near transfer between them). Example: "using loops to draw basic geometric shapes using turtle graphics and block-based programming."
- A *problem family* is a group of closely related problem sets that have similar rules and principles of user interaction. They focus on the same core skills, but they can differ in the inclusion of some skills and consequently in the target age group. Example: "turtle graphics with block-based programming."
- A *problem class* is a generic form of problems with a similar form of user interaction and similar practical usage (e.g., whether they are applicable in "paper&pencil" assessment). They may differ in the scope of skills, but there is at least some common core. We expect at least some transfer of learning within the problem class, i.e., practice on problems in a learning environment should have a positive impact on an independent assessment using different problems from the same class. Example: "block-based programming in microworlds."

We could also consider higher levels of abstraction (*kingdom* ~ well-structured problems for computational thinking, *domain* ~ computational thinking), but these are fixed throughout our discussion.

2.2 Computational Thinking Skills

In order to perform mapping of problems for the practice of computational thinking, we need to decide what will be the scope of the considered problems – what kind of skills[1] do we consider relevant. There is currently no clear consensus on an exact set of computational thinking skills.

[1]Alternative terms used in literature with similar meaning are concept or knowledge component.

Although different authors mostly agree on the basic demarcation of computational thinking and the main skills, they differ in the exact selection and granularity of considered skills.

To determine which skills to consider, we have done an exploration and mapping of the literature. We have focused only on research dealing with computational thinking for beginners; Denning and Tedre [16] provide a wider discussion of computational thinking and highlight the difference between computational thinking for beginners and professionals. From this area, we have analyzed 11 recent and highly-cited papers[2] that contain an explicit discussion of computational thinking skills. The mapping showed several groups of skills based on their occurrence in literature (see Appendix A for more details of the analysis).

Some skills are mentioned ubiquitously. This is particularly the case of *algorithmic thinking* skills. For these, the difference between authors is mainly in the granularity of discussion and the terminology used. For basic algorithmic thinking skills (sequences, loops, conditions) there is a clear agreement. *Abstraction* and *decomposition* are also mentioned ubiquitously, but in this case, it is much less clear what is precisely meant by these notions and what specific practice or assessment problems should correspond to them.

Parallelism, incrementality, and *testing* are not considered in all analyzed papers, but they occur in most of them. We consider them to be part of the core computational thinking skills. Then there is a group of skills that are mentioned only in some studies and that cover a fuzzy transition between core computational thinking skills, mathematics skills and general intelligence: *data representation, data analysis, pattern recognition, logical thinking, problem solving*. We consider these skills in our analysis since they are potentially relevant and can be naturally practiced and assessed with well-structured problems.

Finally, there are several skills that are mentioned in many analyzed papers, but they are hard to practice or assess with the use of well-structured problems: *modeling and simulation*, *generalization*, and *automation*. These skills are more naturally practiced using open-ended projects or discussions and thus are outside the scope of this paper.

2.3 Iterative Mapping of Problems and Their Features

Our goal is to perform the mapping of the landscape of computational thinking problems and to organize them in a taxonomy. The proposed taxonomy is not meant to be *the* taxonomy of computational thinking problems—in this context, it is not meaningful to search for a single, universally correct taxonomy. Our aim is to have a taxonomy that is useful, in the same sense as discussed, for example, by [37].

There are thousands of specific problems that can be used for the practice of computational thinking. In order to organize them, we performed an iterative procedure that utilizes encoding and analyses of problem features. By the generic term *features*, we denote properties of problem sets; these include skills, mode of interaction with the user, target audience, and others. The central aspect of the used procedure is a table with problem sets in rows and their features in columns. The final table is available as a supplement to the paper (`ct-catalog-features.csv`).

2.3.1 Initial Phase. In the initial phase of the process, we performed the following steps: (1) The initial mapping of computational thinking skills in literature (as described above). (2) The collection of an initial set of well-known problems and problems used in specific, closely relevant sources (particularly `code.org` and Bebras competition). (3) Construction of an initial set of problem features based on existing literature; these included the identified computational thinking skills and the properties mentioned in the exercise classification framework proposed by [45]. (4) Practical

[2]Published within the last 10 years, having more than 100 citations in Google Scholar database.

deployment of selected problem types in the learning environment Umíme informatiku[3]. This helped us to clarify the applicability and features of various problems. (5) The coding of problem features in the form of a table.

2.3.2 Iterative Phase. After the initial phase, we repeatedly performed the following steps that led to extensions, refinements, and structure clarification:

(1) The identification of candidate problem families and classes based on the encoded table. This step combined manual insight and prior knowledge with the use of automatic data-mining techniques (dimensionality reduction, clustering).
(2) Scanning through sources (e.g., `code.org`, Bebras competition) with the goal of identifying problems that differ from the already included ones.
(3) The identification of "missing problems," e.g., features or their combinations not sufficiently covered by already included problems or problem classes with a small number or members. This was followed by a specific search for these missing problems in literature and among colleagues. When we failed to find suitable problems, we designed and deployed original problems with the given characteristics.
(4) The refinement of the used encoding of problem features (including skills), particularly: the splitting of features that usefully distinguish existing problems (finer granularity of those features that occur in many problem sets), the omitting or joining of features that occur only sporadically, and the clarification of the meaning of skills (taking into account their relevance in the context of well-structured problems).

2.4 Criteria for Inclusion of Problem Sets

During the described process, we had to decide which problem sets to include. The choice of problem sets was made based on the following criteria:

- The problem set must be in the scope of our interest, i.e., well-structured problems (clear problem statement, clear solution) that can be evaluated algorithmically, problems for beginners (K-12 or introductory university courses), and relevant to computational thinking (at least to one skill that is commonly considered to be part of computational thinking).
- It must be possible to create a sufficiently large set of instances (at least 10 instances can be easily created).
- The problem set must be sufficiently homogeneous to satisfy the near transfer expectation. Preliminary problem sets with large diversity among instances were split into multiple sets.
- Included problem sets should be sufficiently different. When several preliminary problem sets differed only in presentation aspects or some minor details, we tried to provide a single description that focuses on the core aspects and abstracts the presentation aspects.

Overall, we included 93 problem sets organized into 34 problem families. These sets are briefly described in Appendix B.

3 PROBLEM SET FEATURES

In this section, we describe which problem set features we have chosen for inclusion and how we encoded them. We consider only fundamental principles of problems, not their presentation aspects (e.g., story or time pressure) or their specific usage (e.g., problem ordering, recommendations, or motivation features). Due to the focus on fundamentals, the problem sets and their features are relevant for applications in both learning environments (practice) and assessments (testing).

[3]https://www.umimeinformatiku.cz/

3.1 Skills

We have used the iterative method (described in Section 2.3) to reach a selection of skills that is in line with previous research and useful for our purposes. Table 1 provides a listing of the used skills and their brief description. We further clarify the relations of some interconnected skills.

The skills *planning* and *sequences* are both concerned with sequences of actions and their ordering. We use *planning* to denote focus on the problem-solving aspect, whereas *sequences* to denote focus on explicitly written instructions.

Skills related to abstraction, decomposition, and use of functions are clearly interconnected. We use the following specific skills: *data abstraction* is focused on abstracting data (not behavior); *functions* includes both behavior abstraction (focusing on a specific aspect of problem) and the use of functions (as a method for formally specifying behavior abstraction); *decomposition* entails the use of abstraction within the context of the whole problem (e.g., abstracting some steps into standalone sub-problems).

The skills *pattern recognition* and *pattern description* were typically not distinguished in previous work. However, our iterative process of mapping well-structured problems led to their separate treatment since there are natural problems that clearly practice only one of these skills.

The skills *iterative approach* and *testing* are also closely related. We use *testing* when a problem specifically involves a deliberate search for mistakes or formulation of a hypothesis about behavior. Note that in the case of *iterative approach* (and, to a lesser degree, some other described skills), it is often a description of what the student is doing, not necessarily a skill that is practiced or assessed. This distinction may be important for student modeling or studies of assessment validity; for the purposes of our problem landscape mapping, it is not fundamental.

To encode skills, we use three value encoding: 0 denotes cases where a skill is not relevant to a problem set, 0.5 denotes cases where a skill is used to solve problems in a problem set but it is not completely fundamental or it is used only partially, 1 denotes cases where a skill is required to solve problems in a problem set.

3.2 User Interaction, Problem Appearance, and Audience

Table 2 describes several features that we used to encode user interaction style and problem statement and notation. We encode user interaction properties along two dimensions: *interactivity* and the number of *choices* (from how many possible actions students choose). Each of these dimensions is classified into one of three categories described in the table. We also considered the type of feedback provided by the environment (whether there is just a simple correctness feedback message or more complex repeated feedback during the solution process). However, this feature was strongly correlated with *interactivity* and thus did not provide additional information for analysis. Although the *interactivity* and *choices* features are also mostly correlated, there are multiple exceptions that capture useful aspects of several problem sets.

Computational thinking problems typically deal with some program or rules. The *program notation* can take several forms; we distinguish 5 categories described in Table 2. We note that the distinction between textual and block-based programming may be nuanced and there are environments that facilitate the transition from visual to textual programming [33]. For the purposes of our problem mapping, it was, however, mostly clear how to characterize a given problem.

Problems also differ in the appearance of the problem statement and characteristics of inputs and outputs of programs; we distinguish 4 categories (numbers, text, image, microworld).

We also use a feature denoting the nature of used rules. *Explicit rule* means that the problem statement clearly specifies what is the goal to be achieved and what are the means to achieving it (e.g., programming assignments, logic puzzles like Sudoku). *Hidden rule* means that the problem

Table 1. Description of computational thinking skills used for the mapping of problem sets

skill	description
sequences	constructing sequences of actions, understanding written sequences of actions (particularly the importance of ordering)
loops	understanding and using a repeated invocation of actions using loops (repeat, for, while)
conditions	construction and interpretation of expressions with logical operators (and, or, not); understanding and using conditional execution
functions	understanding and using functions to capture some aspect of functionality; ability to abstract behavior
variables	using variables to capture changing information; understanding of basic data types (integers, strings)
parallelism	design and understanding of algorithms with multiple agents acting in parallel, including coordination and synchronization
logical reasoning	using deduction to derive new information from given premises
planning	choosing actions necessary to reach a goal, deciding the right order or actions
decomposition	dividing problems or data into smaller parts that can be tackled separately
data abstraction	using abstraction to simplify provided information
pattern recognition	detecting patterns in a provided information
pattern description	describing patterns using a concise notation
iterative approach	developing solutions iteratively with continuous monitoring of progress; recognizing and fixing mistakes as they occur
testing	forming hypotheses about behavior; deliberate and targeted searching for mistakes
data representation	understanding and using data representation formats; understanding advanced data types (lists, dictionaries)
data analysis	processing data to answer specific questions; interpretation of data visualizations

statement does not explicitly specify the rule behind the problem; the goal is to determine and apply the rule. A typical example of hidden rule usage is the case of pattern recognition problems.

Finally, we also encode the target audience, i.e., for whom is the problem set suitable. Since many problems are applicable for quite a wide range of students, we use only coarse encoding: elementary school (age 6–10), middle school (age 10–14), high school (age 14–18), and university (age over 18). We use binary encoding for each group (a problem set can be relevant to multiple groups).

3.3 Feature Matrix

For each problem set that we included in our catalog, we encoded all the described features. The complete matrix is available as an electronic supplement (`ct-catalog-features.csv`). Table 3 shows the feature values averaged over problem families.

Note that in the case of skill features, this mapping is closely related to Q-matrix used in student modeling [2]. However, the purpose of our mapping is the exploration of the landscape of problem sets, not student modeling. For example, we do not specify the exact relation between multiple skills

Table 2. Features of problem set describing user interaction style and problem notation

interactivity	
1	minimal interactivity, e.g., multiple-choice questions or simple written answer
2	small interactivity, drag and drop
3	highly interactive environment that repeatedly reacts to student actions and this reaction influences student cognition
choices	
1	very limited choice; can be answered correctly by guessing (e.g., multiple-choice questions with 4 options)
2	medium choice of actions; very small chance of guessing correctly (e.g., constructing program from prepared blocks, ordering of several items)
3	very high number of available actions; impossible to answer by random guessing (e.g., writing a program)
program notation	
program	textual notation with in a general high-level programming language
blocks	notation combining text and visual representation (e.g., block-based programming, flowcharts, or diagrams composed of interconnected blocks with text description)
symbols	program or rules written using a small set of symbols, typically in a linear fashion (e.g., simple rewrite rules, mathematics formulas, or a sequence of arrows for describing movement)
text	natural language text with intuitive semantics
none	no explicitly written program or rule, actions are directly performed by a student
problem statement appearance, inputs, outputs	
numbers	input with numbers, operations with numbers
text	text input, operations with text
image	image input, operations with images
microworld	typically 2D microworlds (grids) with agent position and potentially state information

in one problem set (conjunctive/compensatory), which would be necessary for student modeling purposes [44].

4 PROBLEM CLASSES

In this section, we provide a high-level overview of the landscape of well-structured computational thinking problems by discussing the highest level of our proposed problem taxonomy: problem classes.

These classes were determined using the iterative approach described in Section 2.3. One of the steps in this approach was the repeated use of automated techniques to analyze the matrix of problem set features. These techniques included clustering and dimensionality reduction. Figure 2 shows the principal component analysis (PCA) projection for the final data.

Based on this automatically constructed projection, we manually constructed a "landscape map", which aims to increase readability and also to incorporate aspects of problems that are not fully included in the encoded features. This led to Figure 3, which provides a visual map of problem

Table 3. Average values of features in problem families

	notation					rule		appearance				UI		audience				skills																
	program	blocks	symbols	text	none	explicit	hidden	numbers	text	image	microworld	interaction	choices	elemen. school	middle school	high school	university	sequences	loops	conditions	functions	variables	parallelism	logical reasoning	planning	decomposition	data abstraction	pattern recognition	pattern description	iterative approach	testing	data representation	data analysis
Interactive programming	1					1		1	1			3	3			1	1	1	0.7	1	0.7	1				0.2	0.3	0.3		0.8	0.2	0.3	
Find the answer	1					1		0.5	0.2	0.2		1	3			1	1	1	0.9	1	0.9	1				0.5	0.5	0.1		0.9		0.8	0.8
Turtle graphics with text-based lang.	1					1				1	1	3	3		0.3	0.7	0.7	1	1		0.7	0.7				0.5		0.7	0.8	0.8	0.2		
Parson puzzles	1					1		1	0.5			2	2			1	1	1	1	1	0.8	1			0.2	0.2							
Interactive querying	1					1			1			3	3			0.7	1			1						0.3	0.2	0.3	0.7	1	0.2	0.5	1
Robot on grid (blocks)		1				1					1	3	2		1	1		1	1	0.7			0.3		1	0.3		0.2	0.2	0.8			
Robot on grid (arrows)			1			1					1	3	2	0.3	0.7	0.7	0.7	1	1	0.7	0.3				1	0.2		0.5	0.3	0.8			
Turtle graphics with blocks		1				1				1	1	3	2			1	1	1	0.7		0.7	0.3				0.3	0.3	0.2	0.5	1	0.8	0.2	
Shape drawing		1				1				1	1	3	2	0.3	1	0.7		1	0.7		0.3	0.7				0.2	0.2		0.5	0.5	0.8	0.5		
Interpreting sequence of actions			1			1					1	2	2	1	0.5			1								0.2							
Code interpretation (block-based)		1				1		1	1		1	1	1	0.7	1	0.3		1	1	0.7		0.3											
Code interpretation (text-based)	1					1		1	1			1	1			1	1	1	1	1	0.5	1					0.2				0.5		
Evaluating formulas and queries			1			1		0.8	0.5			1	1.5			0.5	1			0.5		0.8				0.2		0.5	0.5			0.5	0.8
Mazes and sliding blocks					1	1					1	3	2	0.8	0.8	0.5	0.2								1	0.1				0.2			
Commands in the microworld			1			1					1	3	2.3	0.7	1	0.3		1		0.7			0.7		0.7					0.7	0.2		
Planning scenario				1		1			1			1	2	0.7	1	0.3				0.2			0.3	0.5	1	0.2	0.3			0.5			
Constraint satisfaction puzzle					1	1		1				3	2	0.5	1	1	0.5							1	0.5			0.5		0.5			
Logic circuits			1			1		1				3	2.5			1	1			1		0.5	0.5		0.5					1	0.2		
Textual logic puzzles				1		1			1	0.3		1.3	2	0.3	1	0.7	0.7			0.8				1	0.2					0.2			
The same pattern					1	1				1		2	2	0.7	1	0.3	0.3										0.3	1					
Detect the pattern					1	1				1		1.5	3	1	1	0.5									0.5			1					
Complete the sequence					1		1	0.3		0.7		1	1.3	1	1	1										0.2	0.3	1		0.5			
Behavior abstraction					1		1		0.5	0.5		2	2	0.5	1	1	0.5				0.5						0.5	0.5		0.5			
Data abstraction					1	0.5	0.5	0.5	0.5	1		2	2	0.5	1	0.5	0.5										1	0.5		0.2	0.5		
Image analysis					1		1			1		1.7	2	0.3	1	0.7	0.3				0.3	0.3				0.5	0.5	0.8	0.3		0.3		
Word relations					1		1		1	0.5		1.5	1.5	1	0.5												1	0.5	0.2		0.5		
Find a hidden configuration					1	1	1			0.5	0.5	3	3		0.5	1	1							0.8	0.5	0.5		0.8		0.5	1		
Find a hidden acceptance rule					1		1	0.3		1	0.3	3	3	0.3	0.7	0.7	0.7								0.5	0.5	0.3	1	0.5	0.5	1		
Find a hidden mapping					1		1				1	3	3	0.5	0.5	1	0.5					0.5			1	0.5	0.5	1		0.5	1		
Binary puzzles					1	1		1				3	2.3			1	1			0.7		0.5		0.5	0.2					0.5		0.7	
Regexp golf			1			1			1			3	3				1											0.8	1	1			0.8
Image representation			1			0.8	0.2	0.7		1		2	2		0.7	0.7	0.3	0.2								0.2	0.2	0.5	0.8			0.8	
Text encoding					1	0.8	0.2	1	1			1	3		0.7	1	0.7											0.7	0.5			1	0.7
Data visualization and understand.					1	1		1	1	1		1.5	1.5		1	1	0.5							0.5		0.5	0.5	0.5	0.8			0.8	1

classes and problem families. It captures the basic similarity relations among classes and positions of problem families within the classes.

One point of these figures is to highlight the fact that the landscape of computational thinking problems is rather continuous, without sharp boundaries between different classes of problems. Any demarcation into discrete groups is thus slightly artificial (as is the case with most classifications and taxonomies).

In the following discussion of classes, we describe common characteristics of problems in the class. We also briefly mention both typical examples and edge cases that represent a transition to other classes or disciplines other than computational thinking. Appendix B contains a catalog of specific problem families and problem sets. In the following discussion, we reference problem families in the catalog by the abbreviation PF followed by the family number in the catalog.

4.1 Programming in a Text-Based Programming Language

Although the aims behind the computational thinking movement are significantly wider than just teaching students the basics of programming, programming is one of the standard activities that is used as a part of computational thinking practice and assessment. This is for a good reason: programming directly or indirectly uses many computational thinking skills (e.g., algorithmic thinking, abstraction, decomposition, testing, and evaluation). The straightforward approach to

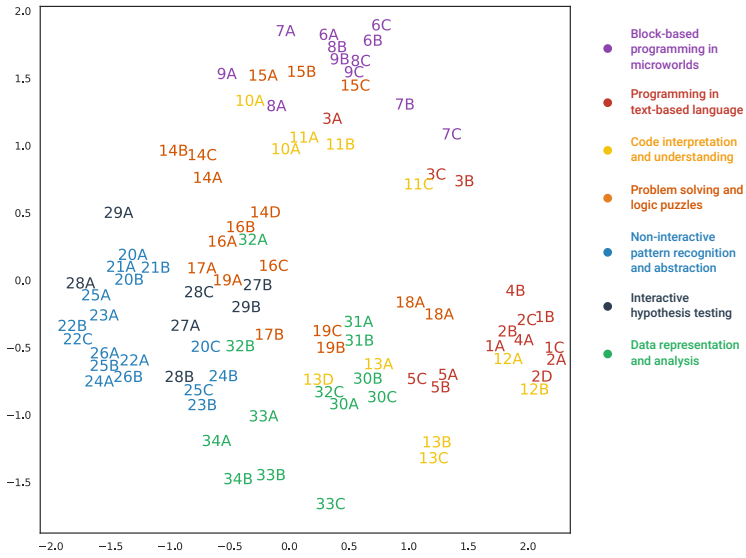


Fig. 2. PCA projection of problem sets based on encoded features (labels were slightly moved to reduce overlaps)

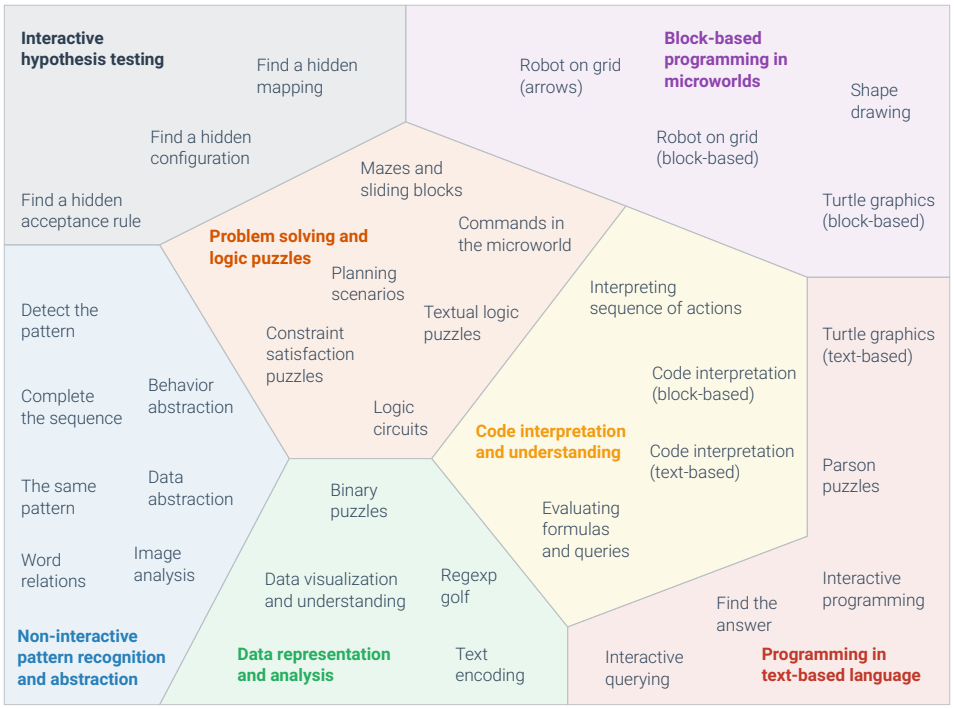


Fig. 3. Manually prepared "landscape map" of problem families and problem classes (based on PCA projection in Figure 2)

practicing programming is to let students write code in a standard, text-based high-level programming language. Programming in a text-based language is mostly relevant for older students. There exists extensive research literature on introductory programming, which is mostly concerned with introductory courses at the university level. [35] provide a good overview.

Typical problems of this type share the following features. They are interactive: students write code, they can execute and test it. When they submit a solution, it is evaluated on hidden testing data; if the solution is incorrect, a student is provided with the specific input on which the program does not work [28]. The program inputs and outputs are typically texts or numbers. The primary skills involved are algorithmic thinking skills (sequences, loops, conditions, variables, functions), being incremental and testing. Advanced problem sets may also require data representation, abstraction, decomposition, and overall integration of many computational thinking skills.

Typical cases of this type correspond directly to the above-given description, varying in their choice of language, specific environment, and scope of programming concepts addressed (PF 1). For some types of programming constructs (e.g., working with files), it is possible to use a less interactive version where students only submit an answer (PF 2).

There are also other potential variations and transitions to other problem classes. When using scaffolding code, which students only slightly modify, there is a smooth transition toward code interpretation problem class. A specific example is the Parson puzzle (PF 4), where the goal is to recover the correct ordering of program lines. Interactive querying (PF 5) is at the transition toward data analysis class; it has a similar mode of interaction as standard text-based programming problems but targets different skills. Text-based programming can be combined with microworlds (e.g., PF 3). The combination with microworlds can also lead to more open-ended activities, e.g., competitions of virtual robots [58].

4.2 Block-Based Programming in Microworlds

Text-based programming in a standard, high-level programming language is typically the final aim of teaching programming. For novices, it is, however, quite intimidating. Consequently, there have been many attempts at lowering the barriers to programming [32]. A common approach is the use of block-based programming, which significantly simplifies syntactical aspects of programming and lets students focus on algorithmic skills [4]. Block-based programming is often used in combination with microworlds; [47] provide an overview specifically of the intersection of block-based programming, microworlds, and well-structured problems.

A microworld is a simplified representation of some domain [49]. Microworlds are useful not just for learning programming but also for other domains (e.g., physics microworlds). In the context of programming, the most common microworld consists of a 2D world with a movable agent and very simple rules, e.g., a robot on a grid with obstacles or an agent which draws pictures by movement (turtle graphics). A problem is typically specified as a goal state for the agent or the world (e.g., reach a target square or draw a figure).

As in the case of text-based programming, the realization is typically interactive: students use blocks to construct a code and repeatedly execute their code until they find a solution. As opposed to text-based programming, in microworlds, there is usually a single initial configuration that is used for the purpose of automatic evaluation of solution correctness[4]. This means that nearly any problem can be solved by a long sequence of elementary actions without a need to use constructs like loops or variables. This can be prevented by the use of limits on the length of code. The main skills practiced by this problem class are algorithmic thinking skills (particularly sequences,

[4]In principle, it would be possible to use multiple input states in a similar fashion to text-based programming. In practice, it is difficult to realize a clear user interface for this approach.

conditions, and loops), planning, and testing. Secondary skills are pattern detection and description, decomposition.

This problem class is relevant to a broad audience. By tuning the complexity of the microworld and available instructions, it is possible to achieve a wide range of problem difficulties. With simple commands and graphical blocks, it is possible to design problems that are solvable even by children who cannot read. On the other hand, it is possible to design microworld puzzles that are challenging even for university students [47].

Typical members feature a robot on a grid (PF 6, PF 7) or drawing microworld (PF 8, PF 9). By using sequences of arrows instead of blocks with text, there is a smooth transition toward the problem-solving class. [36] describe several types of scaffolding in block-based environments; with the use of these scaffoldings, we obtain a smooth transition toward the code interpretation class. Block-based programming in microworlds also has close connections outside computational thinking (e.g., geometry skills) and natural transition toward open-ended activities, e.g., the programming of multimedia stories (Scratch, ScratchJr, Alice) or physical robots (e.g., Lego Mindstorms, Lego WeDo, Ozobot, Beebot, Dash&Dot).

4.3 Code Interpretation and Understanding

This problem class is still focused on programming. The task, however, is not to produce code but rather to correctly interpret provided code. Students are given a fragment of code and have to answer a question testing their understanding of the code. A typical question is "What is the output of the code?" There are also many other options, e.g., "Which input leads to such and such output?", "What is the purpose of this code?", "Is the syntax correct?" (see e.g., [22, 64]). These problems are very useful for quick assessment of students' understanding; several studies have explored the validity of assessments using these types of problems [23, 51]. They are also useful as a tool for learning: [40] argue that it might be better to start teaching CS1 with a focus on comprehension of program execution and only later writing code; [54] describe and evaluate PRIMM (Predict, Run, Investigate, Modify, Make) approach to teaching programming, where the first step is code interpretation.

Problems in this class usually have low interactivity: either a selected response format (multiple-choice question, drag and drop) or a short written answer. The skills practiced by them are mainly algorithmic thinking skills. Compared to code writing, these problems do not involve incrementality, testing, debugging, or planning. However, they allow more targeted addressing of common misconceptions and difficult concepts. See [59] for the most common misconceptions in block-based introductory programming and [48] for a comprehensive review of misconceptions and difficulties in text-based introductory programming.

In the case of producing code, there is a non-trivial step between block-based programming and text-based programming. In the case of code interpretation, the transition can be much smoother. The same code can often be expressed in a high-level programming language, block-based notation, flowchart, or even slightly structured natural language, and the transition between these formats can be rather smooth. These problems are relevant to a broad audience since they can naturally cover the whole range of difficulties from trivial (`print "Hi"; print "Hello"`) to tricky (e.g., understanding nuance of Python operations with lists).

Although the transition is smooth, it is possible to distinguish several main problem families: simple sequences of actions in microworlds (PF 10), code interpretation with block-based notation and focus on control flow structure (PF 11), code interpretation with a high-level programming language, which also considers syntactical aspects of the code (PF 12), and evaluation of formulas and queries (PF 13). The first three families are concerned with sequences of actions, whereas the last one is concerned with a single expression.

4.4 Problem Solving and Logic Puzzles

Problems in this class involve clearly specified rules and a desired target state or situation; the goal is to find a path to the goal state or satisfy all provided constraints. They often involve microworlds. The main difference from previous problem classes is that the problems here do not feature explicitly written programs. Instead, students perform actions directly within the given microworld. This class is at the boundary between computational thinking and mathematics since the focus is on general problem-solving skills and logical thinking (specifically planning and logical deductions).

This problem class is quite rich and contains problems with a variety of interaction modes. It would be possible to create subclassification, e.g., based on the existing taxonomy of puzzles by [26]. However, this class is not completely specific to computational thinking; the skills practiced by problems in this class are closely relevant to mathematics, physics, and other domains. Therefore, we have included in the catalog only several problem families representative of the main directions.

Typical puzzles relevant to computational thinking are mazes and sliding block puzzles (PF 14). They can be naturally used as a preparation for block-based programming in microworlds since they often use similar kinds of microworlds and rules. A particularly interesting variant is the case where the puzzle is solved by placing commands (represented by symbols) directly into the microworld(PF 15).

Other relevant puzzles focus on logical reasoning and deduction. These typically involve minimal interaction: a student often iteratively fills in some information (often in a grid), but feedback is provided only after the solution is completed. For these reasons, they are sometimes called "pencil puzzles" [8]. Specific examples are abstract logic puzzles based on constraint satisfaction (PF 17); a well-known instance is Sudoku. Near the boundary with the data representation class is the construction of logic circuits (PF 18).

There are also puzzles dominated by natural language text (PF 19, PF 16). To solve these problems, students need to correctly understand and interpret the text. Due to the importance of text, this type of problem has close connections to reading comprehension skills — similarly to word problems in mathematics, for which the interplay between numerical and reading comprehension skills is clearly documented [14].

4.5 Non-interactive Pattern Recognition and Abstraction

The main characteristic of this problem class is that it involves a hidden rule; the student needs to determine the rule and apply it (without necessarily explicitly formulating it), e.g., given a sequence of numbers, the goal is to find the next one (which follows according to the sequence rule). Strictly speaking, this type of problem is not completely well-structured since a student may find alternative solutions and argue that they are also acceptable. However, it is possible to create problems where the correct solution is clearly better than any alternative, and thus the problem becomes sufficiently well-structured.

Problems of this type are non-interactive or with very simple interaction: typically multiple-choice questions or simple drag-and-drop exercises. The skills practiced by this type of problem are dominantly pattern detection and abstraction. This type of problem is often used beyond the computational thinking context, particularly in testing general intelligence and abstract reasoning [10].

A typical example of a problem from this class is "complete a sequence" (PF 22): given a series (or matrix) of symbols, the goal is to fill in a missing one. Other commonly used problems are word problems like "word analogy" and "odd one out" (PF 26). The solution of these problems requires the use of abstraction. However, this problem family is on the boundary of computational thinking

and language skills and its relation to other computational thinking problems is not completely clear; future research is needed to clarify this.

We have also collected or created problems specifically for computational thinking context: data abstraction (PF 24), behavior abstraction (PF 24), and analysis of images (PF 25).

4.6 Interactive Hypothesis Testing

This problem class is also based on the "hidden rule" principle. The difference is that in the previous case, all the information for solving the problem was provided statically, whereas in this case, the student interactively proposes testing inputs that help to determine the rule. A key part of the problem-solving aspect is thus the choice of suitable inputs.

The realization of problems in this class is necessarily interactive: the student chooses some inputs and the environment provides the answer for the experiment. The goal is to use as few attempts as possible to determine the hidden rule. The main skills practiced by these problems are forming hypotheses and efficient testing. Other skills depend on a specific problem; they typically involve abstraction, pattern recognition, and logical thinking.

Problems in this class do not have a standard name or classification; there are denoted by several different names, e.g., "inductive reasoning game" or "black box puzzle." To organize these problems, we have collected various specific examples (e.g., Mastermind, Zoombinis, Eleusis, Zendo, Mao, Laser black box) and classified them by what is hidden: a specific configuration (PF 27), an acceptance rule (PF 28), or a mapping between states (PF 29).

4.7 Data Representation and Analysis

Problems in this class concern methods for data representation and analysis. This class is related to pattern recognition and abstraction class – representations of data are typically closely connected with recognizing and describing patterns. In the case of this class, the focus is on the explicit description of patterns. The description can be in some special-purpose notation (describing embroidery patterns using arrows) or using a standard data representation scheme (describing color images using RGB codes). Solving these types of problems may require some specific knowledge behind the data representation scheme. The main focus, however, is typically on computational thinking principles: recognizing patterns and constructing their algorithmic descriptions. This problem type forms a bridge between basic pattern recognition (where the rule is not verbalized) and general-purpose programming (which can be used, as one of its applications, to describe patterns). Typical examples for data representation focus on binary numbers (PF 30), image representation (PF 32), or text encoding (PF 33). Description of patterns can be done using regular expressions (PF 31).

Data representation is interconnected with data analysis and data visualizations. These topics are more suitable for open-ended exploration. Nevertheless, it is possible to devise well-structured problems that practice basic analysis and visualization skills, e.g., questions asking about an interpretation of a provided visualization or a choice of a suitable analysis approach (PF 34).

5 DISCUSSION

In this work, we provide an extensive catalog of problem sets for the practice and assessment of computational thinking and describe the properties of these problem sets. Based on the analysis of these properties, we propose a taxonomy that provides an organization of computational thinking problems. To conclude, we now discuss the potential uses of the presented catalog and taxonomy.

5.1 Current Use of Computational Thinking Problems

To explore the current usage of the identified problem classes, we analyzed research papers in a recent systematic review on the assessment of computational thinking [60]. This review paper used a systematic procedure to identify computational thinking assessment papers and classified assessment types as traditional, portfolio, or other. We analyzed all papers identified as using a "traditional" assessment. Within these, we found 20 papers that used well-structured problems and included sufficiently detailed information to determine what kind of problems was used.

Within this sample, the clearly dominant problem class is "code interpretation." This class was used in some form in nearly all of these papers. Other repeatedly used classes are "block-based programming in microworlds" (6 papers) and "problem solving and logic puzzles" (3 papers). Other problem classes were used only very sparsely. The dominance of "code interpretation" has several good reasons: problems from this class are easy to realize and evaluate (they can be mostly realized using general multiple-choice questions), and understanding code offers possibilities to test many computational thinking skills; it is a natural form specifically for evaluation of algorithmic thinking skills, which are the core computational thinking skills.

However, it is possible that the dominant usage of code interpretation is also due to the combination of historical and practical reasons and that there are important facets of computational thinking which are not well-assessed using these types of problems and which could be assessed using different problem classes. We believe that the usage of other problem classes should be seriously considered in future research on computational thinking assessment.

5.2 Design of Computational Thinking Problems

We provide a large collection of problem sets that are systematically organized. It is thus possible to relatively easily find inspiration for a specific purpose. A possible future extension is to construct a database of specific instances of problems that can be directly used. The described features of problems can be used for recommendations or adaptations in personalized learning environments.

The provided map and feature encodings provide inspiration for the design of new problems by focusing attention on combinations of features that are not yet sufficiently covered. The above-given analysis of problem usage in the current assessment clearly shows the absence of problems focusing on abstraction, decomposition, and pattern recognition. In our catalog, we have provided several problem sets focusing on these topics. These are, however, less standard and mature than problems focusing on algorithmic thinking skills, and they could be clearly further developed and improved.

Specifically, we consider as a useful direction attempts to construct "bridges" between existing problem classes. As a specific illustration, we have identified the following gap. For abstraction and decomposition skills, there are available relatively simple activities with images. On the other hand, these skills are used in programming problems, where their usage is complex and mixed with other skills. Can we create some intermediate problems that will help students more smoothly practice their abstraction and decomposition skills?

5.3 Research on Computational Thinking

One of the criticisms of the computational thinking movement is the use of unsupported claims about the transfer and generalizability of skills [61]. The presented taxonomy can be used for a clearer formulation of the scope of research studies and for the study of research questions concerning the validity of assessment and of learning transfer.

Our taxonomy implicitly makes predictions about convergent validity and transfer of learning. The taxonomy indirectly specifies how close two problems are: if they are in the same problem set, they are very close; if they are in different classes, they are far away. The natural hypothesis is

"the convergent validity and level of transfer should be high for problems that are close and low for problems that are far." This is a specific hypothesis that can be experimentally evaluated; such evaluation is, however, a major undertaking since we have discussed a large set of problems (note that current research of this type is scarce and typically involves just two types of problems).

Our main aim is to propose the overall approach to the organization of the landscape of computational thinking problems. The specific taxonomy and specific catalog of problem sets constitute our best effort at the current situation, not a proposal for a final product. We suppose that future research will find a violation of the hypotheses of the above-given form and that the presented taxonomy will be improved by revisions. We believe that the proposed version is a useful start.

REFERENCES

[1] Charoula Angeli, Joke Voogt, Andrew Fluck, Mary Webb, Margaret Cox, Joyce Malyn-Smith, and Jason Zagami. 2016. A K-6 computational thinking curriculum framework: Implications for teacher knowledge. *Journal of Educational Technology & Society* 19, 3 (2016), 47–57.

[2] Tiffany Barnes. 2005. The Q-matrix method: Mining student response data for knowledge. In *American Association for Artificial Intelligence 2005 Educational Data Mining Workshop*. Pittsburgh, PA: AAAI Press, 1–8.

[3] Valerie Barr and Chris Stephenson. 2011. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *Acm Inroads* 2, 1 (2011), 48–54.

[4] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Commun. ACM* 60, 6 (2017), 72–80.

[5] Petr Boroš, Juraj Nižnan, Radek Pelánek, and Jiří Řihák. 2013. Automatic detection of concepts from problem solving times. In *International Conference on Artificial Intelligence in Education*. Springer, 595–598.

[6] Acey Boyce and Tiffany Barnes. 2010. BeadLoom Game: using game elements to increase motivation and learning. In *Proceedings of the fifth international conference on the foundations of digital games*. 25–31.

[7] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*, Vol. 1. 25.

[8] Zack Butler, Ivona Bezáková, and Kimberly Fluet. 2017. Pencil puzzles for introductory computer science: An experience- and gender-neutral context. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 93–98.

[9] Ünal Çakıroğlu, İsak Çevik, Engin Köşeli, and Merve Aydın. 2021. Understanding students' abstractions in block-based programming environments: A performance based evaluation. *Thinking Skills and Creativity* 41 (2021), 100888.

[10] Patricia A Carpenter, Marcel A Just, and Peter Shell. 1990. What one intelligence test measures: a theoretical account of the processing in the Raven Progressive Matrices Test. *Psychological review* 97, 3 (1990), 404.

[11] John Crabtree and Xihui Zhang. 2015. Recognizing and Managing Complexity: Teaching Advanced Programming Concepts and Techniques Using the Zebra Puzzle. *Journal of Information Technology Education: Innovations in Practice* 14 (2015), 171–189.

[12] Valentina Dagienė and Gerald Futschek. 2008. Bebras international contest on informatics and computer literacy: Criteria for good tasks. In *International conference on informatics in secondary schools-evolution and perspectives*. Springer, 19–30.

[13] Valentina Dagienė and Sue Sentance. 2016. It's computational thinking! Bebras tasks in the curriculum. In *International conference on informatics in schools: Situation, evolution, and perspectives*. Springer, 28–39.

[14] Gabriella Daroczy, Magdalena Wolska, Walt Detmar Meurers, and Hans-Christoph Nuerk. 2015. Word problems: a review of linguistic and numerical factors contributing to their difficulty. *Frontiers in psychology* 6 (2015), 348.

[15] Javier del Olmo-Muñoz, Ramón Cózar-Gutiérrez, and José Antonio González-Calero. 2020. Computational thinking through unplugged activities in early years of Primary Education. *Computers & Education* 150 (2020), 103832.

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

[17] Ruwan Devasurendra. 2020. Bebras Australia Computational Thinking Challenge.

[18] Thomas M Fiore, Alexander Lang, and Antonella Perucca. 2018. Tactile Tools for Teaching: Implementing Knuth's Algorithm for Mastering Mastermind. *The College Mathematics Journal* 49, 4 (2018), 278–286.

[19] Lindsey Ann Gouws, Karen Bradshaw, and Peter Wentworth. 2013. Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 10–15.

[20] Shuchi Grover and Roy Pea. 2018. Computational Thinking: A competency whose time has come. *Computer science education: Perspectives on teaching and learning in school* 19 (2018).

[21] Mariluz Guenaga, Andoni Eguíluz, Pablo Garaizar, and Juanjo Gibaja. 2021. How do students develop computational thinking? Assessing early programmers in a maze-based online game. *Computer Science Education* 31, 2 (2021), 259–289.

[22] Orit Hazzan, Tami Lapidot, and Noa Ragonis. 2014. *Guide to teaching computer science.* Springer.

[23] Sally AM Hogenboom, Felienne FJ Hermans, and Han LJ Van der Maas. 2021. Computerized adaptive assessment of understanding of programming concepts in primary school children. *Computer Science Education* (2021), 1–30.

[24] Juraj Hromkovič, Tobias Kohn, Dennis Komm, and Giovanni Serafini. 2016. Combining the power of python with the simplicity of logo for a sustainable computer science education. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer, 155–166.

[25] Ting-Chia Hsu, Shao-Chen Chang, and Yu-Ting Hung. 2018. How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education* 126 (2018), 296–310.

[26] Lianne V Hufkens and Cameron Browne. 2019. A Functional Taxonomy of Logic Puzzles. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–4.

[27] Nicholas Ichien, Hongjing Lu, and Keith J Holyoak. 2020. Verbal analogy problem sets: An inventory of testing materials. *Behavior research methods* 52, 5 (2020), 1803–1816.

[28] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[29] Petr Jarušek and Radek Pelánek. 2011. What Determines Difficulty of Transport Puzzles?. In *Twenty-Fourth International FLAIRS Conference*.

[30] Filiz Kalelioglu, Yasemin Gülbahar, and Volkan Kukul. 2016. A Framework for Computational Thinking Based on a Systematic Research Review. *Baltic Journal of Modern Computing* 4, 3 (2016), 583.

[31] Jason Zagami Karsten Schulz, Sarah Hobson. 2016. Bebras Australia Computational Thinking Challenge.

[32] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 83–137.

[33] Yuhan Lin and David Weintrop. 2021. The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67 (2021), 101075.

[34] Abraham S Luchins. 1942. Mechanization in problem solving: The effect of Einstellung. *Psychological monographs* 54, 6 (1942), i.

[35] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 55–106.

[36] Nicholas Lytle, Yihuan Dong, Veronica Cateté, Alex Milliken, Amy Isvik, and Tiffany Barnes. 2019. Position: Scaffolded coding activities afforded by block-based environments. In *2019 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 5–7.

[37] Thomas W Malone and Mark R Lepper. 2021. Making learning fun: A taxonomy of intrinsic motivations for learning. In *Aptitude, learning, and instruction*. Routledge, 223–254.

[38] Linda Mannila, Valentina Dagiene, Barbara Demo, Natasa Grgurina, Claudio Mirolo, Lennart Rolandsson, and Amber Settle. 2014. Computational thinking in K-9 education. In *Proceedings of the working group reports of the 2014 on innovation & technology in computer science education conference*. 1–29.

[39] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. 2015. Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia* 46 (2015), 1–23.

[40] Greg L Nelson, Benjamin Xie, and Andrew J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 2–11.

[41] Seymour A Papert. 2020. *Mindstorms: Children, computers, and powerful ideas*. Basic books.

[42] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.

[43] Richard Pattis, J Roberts, and M Stehlik. 1981. Karel the robot. *A gentle introduction to the Art of Programming* (1981).

[44] Radek Pelánek. 2017. Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques. *User Modeling and User-Adapted Interaction* 27, 3 (2017), 313–350.

[45] Radek Pelánek. 2020. A classification framework for practice exercises in adaptive learning systems. *IEEE Transactions on Learning Technologies* 13, 4 (2020), 734–747.

[46] Radek Pelánek. 2021. Adaptive, Intelligent, and Personalized: Navigating the Terminological Maze Behind Educational Technology. *International Journal of Artificial Intelligence in Education* (2021), 1–23.

[47] Radek Pelánek and Tomáš Effenberger. 2020. Design and analysis of microworlds and puzzles for block-based programming. *Computer Science Education* (2020), 1–39.

[48] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.
[49] Lloyd P Rieber. 1996. Seriously considering play: Designing interactive learning environments based on the blending of microworlds, simulations, and games. *Educational technology research and development* 44, 2 (1996), 43–58.
[50] Marcos Román-González, Jesús Moreno-León, and Gregorio Robles. 2019. Combining assessment tools for a comprehensive evaluation of computational thinking interventions. In *Computational thinking education*. Springer, Singapore, 79–98.
[51] Marcos Román-González, Juan-Carlos Pérez-González, and Carmen Jiménez-Fernández. 2017. Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in human behavior* 72 (2017), 678–691.
[52] Elizabeth Rowe, Ma Victoria Almeda, Jodi Asbell-Clarke, Richard Scruggs, Ryan Baker, Erin Bardar, and Santiago Gasca. 2021. Assessing implicit computational thinking in Zoombinis puzzle gameplay. *Computers in Human Behavior* 120 (2021), 106707.
[53] Linda Seiter and Brendan Foreman. 2013. Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 59–66.
[54] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education* 29, 2-3 (2019), 136–176.
[55] Valerie J Shute, Chen Sun, and Jodi Asbell-Clarke. 2017. Demystifying computational thinking. *Educational Research Review* 22 (2017), 142–158.
[56] Helmut Simonis. 2005. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, Vol. 12. 13–27.
[57] Raymond Smullyan. 1986. *What is the Name of this Book?* Touchstone Books.
[58] Suwasono Suwasono, Dwi Prihanto, Irawan Dwi Wahyono, and Andrew Nafalski. 2017. Virtual Laboratory for Line Follower Robot Competition. *International Journal of Electrical and Computer Engineering* 7, 4 (2017), 2253.
[59] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 151–159.
[60] Xiaodan Tang, Yue Yin, Qiao Lin, Roxana Hadad, and Xiaoming Zhai. 2020. Assessing computational thinking: A systematic review of empirical studies. *Computers & Education* 148 (2020), 103798.
[61] Matti Tedre and Peter J Denning. 2016. The long quest for computational thinking. In *Proceedings of the 16th Koli Calling international conference on computing education research*. 120–129.
[62] Peter C Wason. 1960. On the failure to eliminate hypotheses in a conceptual task. *Quarterly journal of experimental psychology* 12, 3 (1960), 129–140.
[63] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2016. Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology* 25, 1 (2016), 127–147.
[64] Greg Wilson. 2019. *Teaching Tech Together: How to Make your lessons work and build a teaching community around them*. CRC Press.
[65] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.
[66] Jessica Yauney, Scott R Bartholomew, and Peter Rich. 2022. A systematic review of "Hour of Code" research. *Computer Science Education* (2022), 1–33.
[67] LeChen Zhang and Jalal Nouri. 2019. A systematic review of learning computational thinking through Scratch in K-9. *Computers & Education* 141 (2019), 103607.

A SKILLS: LITERATURE MAPPING

For the analysis of the treatment of computational thinking skills in literature, we have analyzed a sample of papers that are recent (published in the last 10 years) and highly cited (at least 100 citations in Google Scholar): [1, 3, 7, 20, 25, 30, 38, 53, 55, 63, 67].

Comparison of covered skills across papers is not straightforward because authors differ in the used terminology and in the granularity in which they discuss skills (e.g., "algorithmic thinking" versus "variables"). We have created a compromise skill list and for each paper classified its coverage of the skill with ternary classification: 0 = not covered, 0.5 = partially covered, 1 = covered. The value 0.5 is used in cases where the terminology or granularity does not fit exactly ours, and thus the coverage of the skill may be only implicitly implied. Table 4 provides the results.

Table 4. Overview of skills considered in highly-cited overview papers about computational thinking

	Barr, Stephenson, 2011	Brennan, Resnick, 2012	Seiter, Foreman, 2013	Mannila et al. 2014	Weintrop et al. 2016	Angeli et al., 2016	Kalelioglu et al., 2016	Shute et al., 2017	Hsu et al., 2018	Grover, Pea, 2018	Zhang, Nouri, 2019
data abstraction	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑
functions (behavior abstraction)	●	◑	◑	◑	●	◑	◑	◑	◑	◑	◑
decomposition (modularity)	●	●	●	●	●	●	◑	●	●	●	●
algorithmic thinking: sequences	◑	●	●	◑	●	●	◑	◑	◑	●	●
algorithmic thinking: loops	◑	●	●	◑	●	●	◑	◑	◑	●	●
algorithmic thinking: conditions	◑	●	●	◑	●	●	◑	◑	◑	●	●
variables, basic data types (int, str)	◑	●	●	◑	●	●	◑	◑	◑	◑	●
parallelism	●	●	●	●			◑	●	●		●
data representation	●		●	●	●		◑		●		
data analysis	●			●	●		◑	●	●		
pattern recognition							●	●	●	●	
pattern description (pattern generalization)								◑	●	◑	
logical reasoning	◑						◑		◑	●	◑
planning, problem solving					●		●		◑	◑	
iterative approach (being incremental)		●			◑	◑	◑	●	◑	●	●
testing	●	●			●	◑	◑	●	◑	●	●
automation	●			●			●		●	●	
modeling and simulation	●			●	●	●	●	●	●		
generalization (transfer to different fields)						●		●	●		

Note that a similar analysis was recently independently reported by [9]. They use a slightly smaller sample of reviewed papers but reach very similar conclusions about the covered skills as we do.

B CATALOG OF PROBLEM SETS

This appendix presents a catalog of problem sets grouped into problem families. Problem families are denoted by numbers; problem sets within a family are denoted by letters. The catalog does not aim at completeness but rather wide coverage. Specifically, the choice of problem sets within families is illustrative: in most cases, it would be possible to mention more problem sets within a family or to organize them in a slightly different manner. For each problem family, we also include an example. Due to limited space, these examples are not completely specified problems but only illustrations that highlight specific aspects of the problem family. The examples are either abstracted versions of commonly used problems or problems that we developed for the Umíme informatiku learning environment (https://www.umimeinformatiku.cz/).

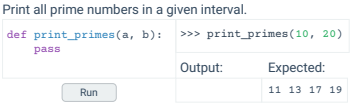
The matrix of problem set features is available as an electronic supplement to the paper (file `ct-catalog-features.csv`).

B.1 Programming in a Text-Based Language

PF 1 Interactive programming

The goal is to write a program in a high-level language (e.g., Python) to solve a given (parametrized) problem. A program is automatically evaluated using hidden testing data. This is a standard setting in introductory programming (usually called autograder), realized in many tools (see [35] for references).

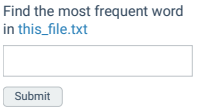
(A) *Expressions and conditions.* Simple problems with numbers and text messages that require only expressions and conditional execution.
(B) *Loops.* Problems using loops (for, while), typically simple computations with numbers (e.g., sums, divisibility).
(C) *Strings and lists.* Problems that require the use of strings and lists.



PF 2 Find the answer

The goal is to write a program for a specific task and submit only the result of the computation on a specific input. This is useful, for example, for situations where the autograder approach may be more technically difficult to realize.

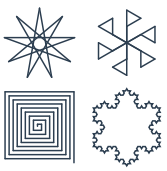
(A) *Text file analysis.* The goal is to compute requested information for a given text file (e.g., the text of a book).
(B) *Bitmap image processing.* The goal is to recover information hidden in a provided bitmap file (e.g., using minor color differences which are not perceptible by the eye).
(C) *Math problems.* Problems with numbers that cannot be easily solved without programming, e.g., "find the sum of all natural numbers below 1000, for which the digit sum is 13." A well-known collection of such problems (but mostly too advanced for novices) is Project Euler (projecteuler.net).
(D) *Tabular data analysis.* The goal is to compute requested information based on a given tabular data (e.g., Olympic medal table).



PF 3 Turtle graphics with a text-based language

The goal is to draw a specified image in a turtle graphics microworld (vector graphics with a relative cursor). Historically, turtle graphics was used with the Logo language [41]; nowadays, it is commonly used with Python [24].

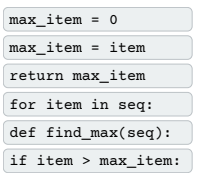
(A) *Loops.* Drawing polygons, stars, and other simple regular shapes using loops.
(B) *Variables.* Drawing spirals and "growing patterns" using variables and simple functions.
(C) *Recursion.* Drawing fractals using recursive functions.



PF 4 Parson puzzles

The goal is to recover the correct ordering of program lines for a specified problem (the input is a sample solution with randomly ordered lines) [42].

(A) *Syntax and semantics.* Simple programs with loops and variables, where the focus is on the basic syntax and semantics of programming constructs (nesting, importance of command order).
(B) *Algorithms.* Simple algorithms that can be written in a few lines of code (e.g., sorting, binary search).



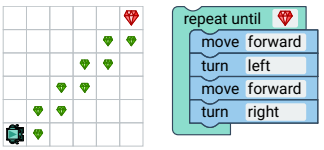
PF 5 Interactive querying
Given data and a query formulated in a natural language, the goal is to write the query in a formal language (e.g., SQL queries over database tables, XPath queries over XML documents, or spreadsheet queries). The interface is interactive, i.e., students can write queries and see results. Similarly to PF 1, it is possible to use evaluation on hidden testing data.

(A) *Basic queries.* Queries over simple data using logical operators and several attributes.
(B) *Multiple data sources.* Queries over data from several sources (e.g., multiple tables in database).
(C) *Advanced queries.* Queries using more advanced concepts (e.g., "group by").

B.2 Block-Based Programming in Microworlds

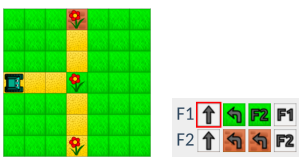
PF 6 Robot on a grid (blocks)
In a microworld with a robot on a 2D grid, the basic goal is to get the robot to a target location using movement actions (forward, turn left, turn right). The microworld may contain specific elements that make the goal more complex, e.g., obstacles to be avoided, objects to be collected, colored cells in the grid (used for conditional movement), enemies. A classic version of this microworld is Karel the Robot [43], which was used with text-based programming. Nowadays, this type of microworld is commonly used with block-based programming. [47] provides a discussion of design issues and [21] a specific analysis.

(A) *Basic movement.* Programs are simple sequences of basic actions.
(B) *Loops and conditions.* Programs utilize loops or conditional movement (e.g., based on the color of cells or the presence of obstacles).
(C) *Parallelism.* There are multiple robots in the microworld; programs have to address coordination issues.



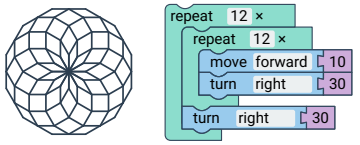
PF 7 Robot on a grid (arrows)
The same basic type of microworld as PF 6, but instead of blocks, programming is done with a linear sequence of arrows. This approach makes it intuitive and simple to start with. At the same time, this setting allows complex problems by the use of functions. Each line of commands defines a function; these functions can be repeatedly called. Specific examples are LightBot [19] or RoboZZle (robozzle.com/).

(A) *Basic movement.* Simple sequences of basic actions.
(B) *Loops and conditions.* Programs utilize conditional execution (e.g., with respect to the cell colors) and looping realized by simple tail recursion.
(C) *Functions and recursion.* By utilizing recursion, it is possible to express complex behavior even with the very simple notation.



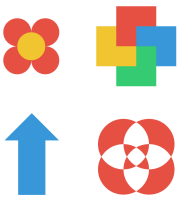
PF 8 Turtle graphics with block-based language
The same turtle graphics microworld with interactive interface and automatic evaluation as in PF 3, but with block-based programming interface and a selection of problems aimed at younger students.

(A) *Basic movement.* Program is specified with blocks using a few simple commands (forward, left, right).
(B) *Loops.* Drawing polygons, stars, and flowers using loops.
(C) *Variables and spirals.* Drawing spirals and "growing patterns" using variables.



PF 9 Shape drawing
This is another variation on the drawing microworld—similar to PF 8 but with a different focus. In this case, the agent moves to specific coordinates or in an absolute direction (as opposed to the relative movement used in turtle graphics). The agent is capable of drawing several elementary shapes (circle, square, triangle) of specified size and color. The goal is to produce a given shape by suitably combining the elementary ones [6, 47].

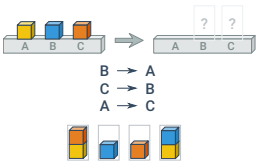
(A) *Basic drawing.* Short sequences of movement and drawing steps.
(B) *Combined and repeated shapes.* Use of loops to draw patterns with simple structure; construction of combined shapes by a suitable ordering of drawing actions.
(C) *Debugging and functions.* Problems feature ready-made code for a given shape, but with a minor bug; the goal is to find and correct the mistake. Problems may also include prepared functions (draw a house), which have to be used.



B.3 Code Interpretation and Understanding

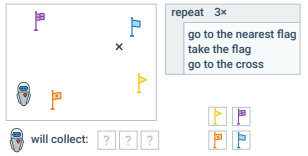
PF 10 Interpreting sequence of actions
Given a simple microworld, the goal is to interpret the effect of a sequence of actions or to fill in missing actions into a provided sequence. The focus is on the ability to mentally imagine the effect of actions and on the understanding of the impact of their ordering.

(A) *Movement in a maze.* A maze microworld, where actions correspond to movement (cardinal directions or movement toward objects in the maze).
(B) *Cube stacking.* A microworld with several stacks of cubes and actions for moving one stack on top of another.



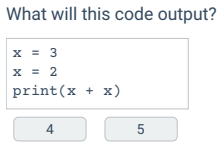
PF 11 Code interpretation (block-based)
Given a code in block-based programming (or semi-structured text or control flow diagram), the goal is to determine the output or answer multiple-choice questions about the behavior of the code. The code can manipulate numbers, texts, or control an agent in a microworld. [23] describe a large collection of these problems.

(A) *Loops.* The code is a sequence of commands, a loop, or two nested loops. Loops use a fixed number of iterations.
(B) *Conditions and while loops.* The code contains conditional execution.
(C) *Variables.* The code manipulates variables.



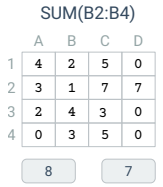
PF 12 Code interpretation (text-based)
Given a code in a high-level programming language (e.g., Python), the goal is to answer a multiple-choice question about its behavior.

(A) *Basic constructs.* Determine the output of a short code fragment; primarily useful for practicing basic constructs (expressions, conditions, loops).
(B) *The behavior of worked-out examples.* Questions about the behavior of a worked-out example, e.g., a function for computing digit sum. Primarily useful for practicing functions and data structures.



PF 13 Evaluating formulas and queries
Given a formula or query in a specific formalism, the goal is to determine the result of the formula (either as a selected response or a constructed short answer).

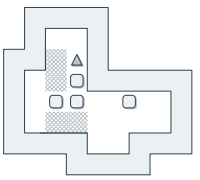
(A) *Logic expressions.* For a given logic expression (using standard operators like and, or, not), determine its truth value.
(B) *Spreadsheets formulas.* For a given spreadsheet table and a formula (e.g., `SUM(A3:A7)`), determine its value or interpretation.
(C) *SQL queries.* For a given database table and SQL query, determine the result of the query or its interpretation.
(D) *Regular expressions.* Decide whether a regular expression matches a given word.



B.4 Problem Solving and Logic Puzzles

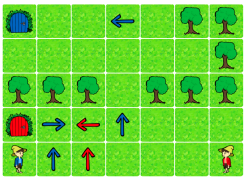
PF 14 Mazes and sliding blocks
The goal is to find a path to a goal state in a 2D grid microworld or a maze. The problem is interactive, i.e., the student controls an agent that moves in the maze. [29] present an analysis of difficulty factors for this type of puzzle.

(A) *Maze with conditions.* Basic pathfinding in a maze; the maze can have special conditions, e.g., no left turn, keys and doors, 3D maze with ladders and holes.
(B) *Maze with optimization.* The goal is to find a path through a maze while optimizing a certain criterion, e.g., the number of coins collected or fuel-efficiency.
(C) *Sliding block puzzles.* Given a 2D grid with several pieces, the goal is to slide the pieces to a goal configuration. Well-known specific examples are 15 puzzle, Rush hour, and Klotski.
(D) *Pushing agent.* Similar to sliding block puzzles, but with a single agent that pushes pieces. A well-known example is the Sokoban game, more modern and complex examples are Stephen's Sausage Roll or Baba Is You.



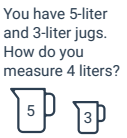
PF 15 Commands in the microworld
This problem has a similar setting as PF 6, PF 7, or PF 14 but students neither control agents directly nor do they write a standalone program. Instead, the commands for agents are placed directly into the microworld and executed when the agent reaches them. A specific example is the Arrow game described by [47]; other related examples are OzoBot (physical robot capable of reading commands drawn as sequences of colors) or The Incredible Machine game.

(A) *Basics.* Basic movement of a single agent, e.g., using arrows for a change of direction and jumps for overcoming obstacles.
(B) *Multiple agents.* Setting with multiple agents, where it is necessary to address coordination issues (avoiding collisions).
(C) *Changing state.* Problems where it is necessary to take into account changing state of the environment (e.g., disappearing arrows, explosions).



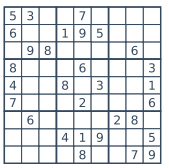
PF 16 Planning scenario
A problem is given as a textual description of a planning situation with several constraints. The goal is to find a solution (sequence of steps) that satisfies all requirements.

(A) *Transport puzzles.* The goal is to get some objects to target locations according to given rules. Well-known examples of this type are river crossing puzzles (e.g., jealous husbands, missionaries and cannibals).
(B) *Scheduling.* The goal is to plan a sequence of actions while satisfying given time constraints, e.g., Aircraft Scheduling in [17].
(C) *Water pouring puzzles.* The goal is to measure a specified volume of water using several water jugs with a known integer capacity. Well-known research use of these puzzles is for the illustration of the Einstellung effect [34].



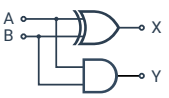
PF 17 Constraint satisfaction puzzle
This type of puzzle is a specific instance of a general constraint satisfaction problem [56]. The problem takes the form of a grid that is partially filled with numbers, lines, or similar objects. The goal is to fill the rest of the grid so that all conditions are satisfied.

(A) *Symbol placement.* The goal is to find the position of symbols within a grid. Examples: Sudoku, Tents, Easy as ABC, Skyscrapers.
(B) *Find the path.* The goal is to find a path (or several paths) through a grid, satisfying both local constraints and some global condition (e.g., non-branching, visiting all squares). Examples: Slitherlink, Number links, Nurikabe.



PF 18 Logic circuits
The goal is to build a specified functionality using elementary logic gates (and, or, not). The problem is typically presented in interactive form using standard computer science notation. It is also possible to present these problems using non-computer metaphors, e.g., the flow of water, wooden machine, or bottle recycling [17].

(A) *Combinational circuits.* Circuits using only the basic logic gates (and, or, not), e.g., the addition of two two-bit numbers.
(B) *Sequential circuits.* Circuits that use memory, e.g., a counter.



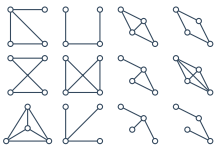
PF 19 Textual logic puzzles
Given a set of statements in a natural language, the goal is to determine specific information using logical deduction.

(A) *Interpreting statements.* Given a situation (e.g., described by an image) and a sentence, determine whether it is valid or what is the consequence of the action. Specific example (bomb deactivation): Given a schema of a bomb and a deactivation instruction, the goal is to choose the correct action. A complex, multiplayer version of this problem is the popular game Keep Talking and Nobody Explodes.
(B) *Knights and knaves.* This is a type of logic puzzle, popularized by [57], where some characters can only answer questions truthfully and others only falsely. Given a set of statements by several characters, the goal is to determine who is a knight and who is a knave.
(C) *Logic grid puzzle.* Given a set of statements about several persons, the goal is to determine features of individual persons (similarly to PF 17, this is a special case of constraint satisfaction problem). The most well-known instance of this type of puzzle is called Zebra Puzzle or Einstein's Riddle [11]).

B.5 Non-interactive Pattern Recognition and Abstraction

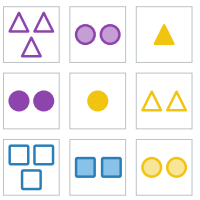
PF 20 The same pattern
Given a set of images, the goal is to find pairs that show the same pattern.

(A) *Simple identity.* The images are completely identical. The problem is non-trivial in cases where the set contains images with closely similar patterns (e.g., various snowflakes).
(B) *Rotations and reflections.* The images are identical but may be reflected or rotated.
(C) *Graph isomorphism.* The images show graphs (nodes and edges); the goal is to match isomorphic graphs. The problem can be presented without graph terminology, e.g., as matching of animals made of sticks and nuts [31].



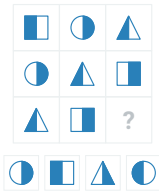
PF 21 Detect the pattern
In these problems, there is an explicitly formulated rule for a pattern, and the goal is to find or create the pattern in a given data.

(A) *Select the pattern.* Given a set of objects, the goal is to select a subset that corresponds to the given rules. A well-known example is the "Set game", which is played with cards that vary in four features (number, shape, color, shading) across three possibilities (e.g., circle, triangle, and square for shapes). The goal is to find a triplet of cards, such that for each one of the four features, the three cards must display that feature as either all the same or all different.
(B) *Create the pattern.* Given a set of objects, the goal is to manipulate them (according to some rules) so that the desired pattern is created. This mechanism is used in tile-matching video games like Bejeweled or Candy Crush Saga.



PF 22 Complete the sequence

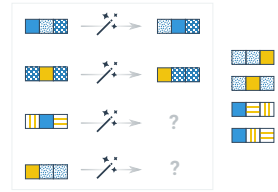
Given a sequence of objects that obey a hidden rule, the goal is to determine the rule and deduce the next object in the sequence. Interaction is typically realized as a multiple-choice question. This problem is commonly used in general intelligence testing [10].



(A) *Number and letter sequences.* The principles used in rules include repetitions, arithmetic and geometric sequences, and modular arithmetics.
(B) *Geometric sequences.* The principles used in rules include rotations, axial symmetry, movement in space, and permutations of colors and fill patterns.
(C) *Raven's progressive matrices.* Geometric pattern organized in a a matrix (the original formulation used 2×2 matrices, a common form uses 3×3 matrices).

PF 23 Behavior abstraction

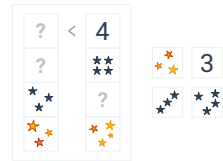
Given several examples of function behavior, the goal is to deduce the behavior and apply it in another setting.



(A) *Image transformation.* The behavior is a transformation of images (e.g., rotation, reflection, permutation of colors, change of size, and their combinations).
(B) *Word transformation.* The behavior is a transformation of a word (e.g., addition or removal of a letter, permutation).

PF 24 Data abstraction

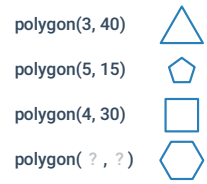
Given several images, the goal is to organize them with the use of abstraction.



(A) *Abstraction ladder.* The problem is given as a pattern of images with several missing ones. The goal is to fill in the missing images from a selection in such a way that each column of images represents an abstraction ladder (going from the most concrete to the most abstract).
(B) *Graph neighbors.* The goal is to find a correspondence between "maps" (images with several regions) and their graph representation (dual graphs), see e.g., Encoded Treasure Map in [17].

PF 25 Image analysis

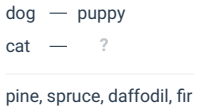
Given several images, the goal is to assign them a correct description. The images and descriptions are prepared in such a way that the assignment requires the use of abstraction or decomposition.



(A) *Image decomposition.* Given several images, the goal is to decompose them into several constituent parts that are used repeatedly in these images.
(B) *Image codes.* Given several images and their compact codes, the goal is to determine the principle behind the codes and determine codes for several other images. The codes are cryptic sequences of letters or numbers, which denote some properties of images.
(C) *Function headers.* The descriptions are function calls with parameter values.

PF 26 Word relations
Given a set of words, the goal is to choose one of them or to find a new word by reasoning about their relations and properties [27].

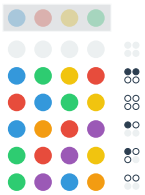
(A) *Word analogies.* Word analogy is a common verbal reasoning problem that can be seen as a practice of abstraction. The illustration shows a common form of this problem, i.e., given an example pair, the goal is to fill in the second word in a second pair in such a way that the relationship is the same.
(B) *Odd one out.* Given a set of words, the goal is to determine what they have in common and select a word that does not belong to the group.



B.6 Interactive Hypothesis Testing

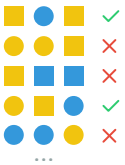
PF 27 Find a hidden configuration
The goal is to determine a hidden configuration of some objects using a limited number of queries. The rules of a specific problem specify possible configurations, queries, and the form of the answer to queries.

(A) *Categorical objects.* A well-known example is the Mastermind game [18], where the hidden configuration is a sequence of colored pegs. The query is also a sequence of pegs; the response is information on the number of correctly placed pegs and the number of correctly colored (but misplaced) pegs. Another example is the Pizza pass problem in Zoombinis game [52].
(B) *Position in a grid.* The goal is to determine the position of objects in a grid. For example, the configuration may be the position of mirrors in a black box and a query corresponds to pointing a laser into the black box and observing where the beam leaves the box.



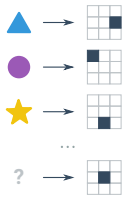
PF 28 Find a hidden acceptance rule
The goal is to determine a hidden acceptance rule in a limited number of attempts. In each attempt, a student provides a specific object and gets a binary response (accepted or not accepted).

(A) *Object features.* The object is composed of several features; the acceptance rule is based on these features. Example: the Alergic cliffs puzzle in the Zoombinis game [52].
(B) *Multiple object configurations.* The rule concerns configurations of objects, e.g., colorful building blocks (Zendo game) or number sequences. A well-known example is Wason's 2-4-6 task [62].
(C) *Move sequences.* The rule concerns the acceptance of moves within a game, e.g., the movement of figures on a grid (Penultima game) or card sequences (Eleusis game, Mao game).



PF 29 Find a hidden mapping

The goal is to determine a hidden mapping that transforms an input state into an output state. The student's attempt consists of choosing an input state and observing an output state. The goal is typically formulated as an output state for which the student has to find a corresponding input state (in a limited number of attempts).



(A) *Memory-less mapping.* There exists a direct systematic correspondence between inputs and outputs, e.g., Mudball wall in the Zoombinis game [52].

(B) *Mapping with memory.* The output depends not only on the current input but also on previous ones, e.g., the mapping corresponds to a finite-state machine.

B.7 Data Representation and Analysis

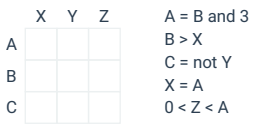
PF 30 Binary puzzles

Given a set of clues, the goal is to fill a grid with binary values (0, 1) so that all clues are satisfied. This problem family can be seen as a special case of constraint satisfaction puzzles PF 17 that is tailored specifically toward the use of binary values. [5] provide specific examples and analysis.

(A) *Binary numbers.* The clues concern the binary representation of numbers and arithmetic operations over them (addition, subtraction).

(B) *Logic operations.* The clues concern logical operations (and, or, not), which are performed in a bitwise manner over rows and columns.

(C) *Crosswords.* The clues may involve both binary numbers and logic operations and contain cyclical references. Solving a problem thus requires planning and logical reasoning.



PF 31 Regexp golf

Given a set of positive and negative examples (words), the goal is to write a regular expression that matches only the positive examples. The problem is interactive: a student writes a regular expression and is shown which words it matches.

(A) *Basic expressions.* Problems requiring only the basic usage of regular expression syntax (wildcards, letter groups, repetition).

(B) *Advanced expressions.* Problems requiring more complex syntax (word boundaries, backreference) and problems requiring a compact description of patterns due to a short limit on the length of a regular expression.

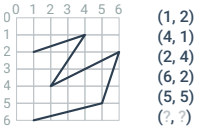
PF 32 Image representation

Given an image and its representation (as a text or a list of symbols), the goal is to fill missing parts (parts of the description or parts of the image) or choose a matching image and description.

(A) *Arrow sequences.* Repetitive line images (e.g., simple embroidery patterns) represented as a sequence of arrows describing moves.

(B) *Bitmaps and vectors.* Images represented in a simplified bitmap or vector format (e.g., list of zeros and ones for individual pixels, lines represented by coordinates).

(C) *Color codes.* Matching colors and their RGB codes.



PF 33 Text encoding

The goal is to transform a text using a given procedure that modifies its representation. Simple problems require just a straightforward application of provided transformation rules; more complex problems require pattern recognition and logic to deduce used rules.

(A) *Coding.* Representation of text using common coding schemes (e.g., ASCII table, Morse code).
(B) *Ciphers.* Transformation of text using simple ciphers (e.g., transposition, monoalphabetic substitution).
(C) *Compression.* Text compression using simple algorithms (e.g., run-length encoding, dictionary methods, delta encoding).

2354 → R I D E
6412 → P E A R
2764 → R O P E
2415 → ? ? ? ?

PF 34 Data visualization understanding

Given visualization of data, the goal is to answer questions that test the understanding of the visualization.

(A) *Matching tables and graphs.* The goal is to match a data table and a corresponding or two different visualizations of the same data.
(B) *Interpretation.* For the visualization of some real-world data, the goal is to answer multiple-choice questions about its interpretation.