Steps Before Syntax: Helping Novice Programmers Solve Problems using the PCDIT Framework

Oka Kurniawan[1], Cyrille Jégourel[1], Norman Tiong Seng Lee[1], Matthieu De Mari[1], and Christopher M. Poskitt[2]
[1]Singapore University of Technology and Design, Singapore
[2]Singapore Management University, Singapore
{oka_kurniawan, norman_lee, cyrille_jegourel, matthieu_demari}@sutd.edu.sg, cposkitt@smu.edu.sg

Abstract

Novice programmers often struggle with problem solving due to the high cognitive loads they face. Furthermore, many introductory programming courses do not explicitly teach it, assuming that problem solving skills are acquired along the way. In this paper, we present 'PCDIT', a non-linear problem solving framework that provides scaffolding to guide novice programmers through the process of transforming a problem specification into an implemented and tested solution for an imperative programming language. A key distinction of PCDIT is its focus on developing concrete cases for the problem early without actually writing test code: students are instead encouraged to think about the abstract steps from inputs to outputs before mapping anything down to syntax. We reflect on our experience of teaching an introductory programming course using PCDIT, and report the results of a survey that suggests it helped students to break down challenging problems, organise their thoughts, and reach working solutions.

1. Introduction

Learning programming from scratch is understood to be challenging [1]. This is mainly due to the high cognitive load involved in typical introductory courses: novice programmers must learn language syntax, IDEs, engineering principles (e.g. abstraction, modularity), and computational thinking all for the first time. Until their mental models have fully developed, novices can struggle to get started on a harder programming problem, sometimes taking a 'syntax-first' approach of coding something—*anything*—before even analysing how to properly solve it. This is compounded by the fact that many introductory courses do not explicitly teach *problem solving* skills and strategies, assuming instead that they are picked up along the way [2].

Several authors have developed approaches for guiding novice programmers through the process of problem solving. For example, in their Python textbooks, Dierbach [3] and Liang [4] proposed frameworks based on the steps of the software development life cycle, i.e. analysis/requirements, design, implementation, and testing. Students are encouraged to think about the input/output requirements of the problem, "design a process for obtaining the output from the input" [4], implement it, then finally "test the program on a selected set of problem instances" [3]. As another example, Loksa et al. [5] proposed a framework that expands upon the design phase by encouraging students to search for analogous problems and solutions that can be applied to the one they are tackling.

These approaches share a typical characteristic in that testing is at the end of the process. Actually being able to get to the end, however, depends on the student's level of metacognitive awareness, i.e. their ability to think on their own about the problem [6]. For students lacking metacognitive skills, previous research has highlighted the benefit of solving concrete cases *before* programming [7], as well as making the problem solving process explicit (e.g. by using an automated assessment tool) and having them reflect on their progress [8, 9]. Controlled experiments in these works revealed that students are more likely to provide a correctly implemented solution to a problem and demonstrate better metacognitive awareness.

It is in this context that we developed 'PCDIT', a problem solving framework for novice programmers that encourages them to design/solve concrete cases before programming, and to regularly reflect using the five eponymous phases of the process (Figure 1): **P**roblem Definition, **C**ases, **D**esign of Algorithm, **I**mplementation, and **T**esting. A key characteristic of the framework is the 'C' phase, which focuses on developing concrete cases for the problem early without actually writing any test code: students instead think about the steps at an abstract level, only mapping them down to program syntax in later phases. Another characteristic is its non-linearity: students are

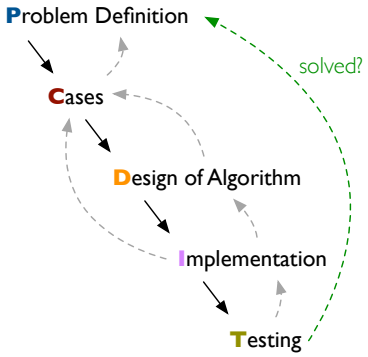


Figure 1. Steps of the PCDIT framework, including some possible non-linear flows between them

encouraged to engage in a reflective, case-driven, and iterative process that may feel more productive and encouraging for novices [10]. The process keeps on going until the programming task has been solved satisfactorily, meaning that the proposed answer fulfills all the requirements described in the problem statement, and can operate correctly on all test cases. Ultimately, the scaffolding of PCDIT makes the process of problem solving that we naturally follow—but many novices do not—explicit.

In this experience report, we introduce the PCDIT framework, and describe how we used it in a first-year undergraduate programming course, where it was explicitly taught as a way to help solve more challenging coding exercises. We present the results of a post-use student survey that suggests it was successful in helping students to organise their thoughts, break down problems, and arrive at working solutions. We critically reflect on our experiences—as software engineering lecturers—of using it in teaching, and make a number of recommendations on how to do so more effectively. Finally, we share a PCDIT worksheet and a number of examples that can be used by other practitioners.

2. Related Work

In the mathematical problem solving framework described in Pólya's book, "How to Solve It" [6], the reader is told to: (1) understand the problem, (2) devise a plan, (3) carry out the plan, and (4) look back (i.e. evaluate the solution). In the second stage, the reader is asked "Have you seen it before? Do you know a related problem?", implicitly suggesting that successful problem solving requires an adequate knowledge base or cognitive resources. On top of this, according to Schoenfeld [11], successful problem solving requires three other aspects in the learner: (1) heuristics (strategies for making progress in unfamiliar situations), (2) control (making decisions about strategies and resources), and (3) beliefs (i.e. about the subject).

Some introductory Python programming textbooks [3, 4] introduce problem solving using a framework that is based on the software development life cycle and is analogous to Pólya's, namely: (1) requirements and analysis, (2) design, (3) implementation, and (4) testing. In the first stage, Liang [4] tells the reader to use the requirements identified to determine the input and output data for the problem. Dierbach [3] points out that the reader also needs to consider how the input data can be represented in the program itself. For the remaining stages, both authors then tell the reader to write the algorithm, implement it in code, and then evaluate the solution using test cases. While both books use worked examples to illustrate concepts, Dierbach's book also has specific sections that illustrate the problem solving process throughout the chapters.

Other authors adopt a framework that is similar to Pólya's, but with emphasis on different aspects. McCracken et al. [12] highlights a stepwise refinement strategy, citing the need to decompose the problem into sub-problems with the following framework: (1) abstract the problem from its description, (2) generate sub-problems, (3) transform sub-problems into sub-solutions, (4) recompose, and finally, (5) evaluate and iterate. Loksa et al.'s framework [5] tells the reader to draw upon a knowledge base of existing problems in stages (2)–(4): (1) reinterpret problem prompt, (2) search for analogous problems, (3) search for solutions, (4) evaluation of a potential solution, (5) implement a solution, and (6) evaluate implemented solution.

Some authors propose that, at the problem analysis stage, examples should be written to illustrate the purpose of the program. Riley [13] describes requiring students to write a problem definition which contains a problem description, input and output specifications, error conditions, and specific examples that illustrate the input and outputs. In the six-stage process proposed in "How to Design Programs" [14], the reader is told to provide some functional examples in the initial stages of problem analysis, prior to carrying out any design. This involves providing specific pairs of input and output values in order to illustrate the purpose of the function to be written.

Studies have been done on the problem solving ability of students in first-year programming courses. While McCracken et al. [12] suggested that novice learners of programming have no problem solving

ability, subsequent studies have concluded that reducing students' cognitive load might have a positive effect on problem solving performance. In a study by McCartney et al. [15], participants were provided partially-completed code and allowed access to external materials, thus reducing their cognitive load. Many participants demonstrated the ability to program incrementally. Utting et al. [16] showed that participants who were provided with a test harness had better success in completing an object-oriented programming task, compared to those who had to rely solely on the program description.

It has been pointed out that novice learners of computing have "fragile knowledge" [17]. This was illustrated in a study by Lister et al. [18], where novice learners were shown to have a weak ability to read and trace computer programs, lacking the prerequisite skills for problem solving. Thus, pedagogical strategies have been proposed to help novice programmers develop their problem solving skills.

One pedagogical strategy has been to teach students explicitly algorithmic solutions to known problems, e.g. a counter-controlled loop, termed as "goals and plans" by Soloway [19] and "algorithmic patterns" by other authors [20, 21]. Soloway points out that these must be explicitly taught to students. In the problem solving process, students must apply a stepwise refinement process and break down a problem into sub-problems in such a way that these known patterns can be used [19]. Clancy and Linn [22] point out that regular classroom instruction must continually focus on helping students to learn and apply these patterns.

A curriculum that used Soloway's goal/plan framework and explicitly taught 18 algorithmic patterns was reported by de Raadt [23]. The study showed that students successfully used these strategies more often, compared to previous iterations of the course that only exposed students to these patterns implicitly.

Specific instruction in class on problem solving has also been used as a pedagogical strategy. Arshad [24] used recitation sessions where teaching assistants demonstrated solutions to programming problems and articulated their thoughts aloud at the same time. Students reported that this aspect of the course was most useful for them.

Falkner and Palmer [25] described a class with three interventions: live demonstrations of programming examples, lessons that specifically discussed problem solving, and encouraging cooperative problem solving in students.

Loksa et al.'s study [5] seemed to show that interventions designed to improve problem solving skills and encourage metacognition had a positive effect on novice learners. Participants in the study were explicitly taught the problem solving framework. As the participants worked on the programming tasks and encountered difficulties, they were prompted to think about which stage of the problem solving framework they were in. The IDE used by the participants had an "idea garden", which was a set of problem solving strategy hints which students could use.

Prather et al. [8] described metacognitive difficulties that students faced, noting that many students misunderstood the problem prompt and formed the wrong conceptual model, began coding straightway without designing the solution, and were not able to respond correctly to feedback from error messages and failed test cases. They also pointed out that current automated assessment tools (AAT) did not have features that facilitated students through Loksa et al.'s problem solving framework.

Subsequently, an AAT was modified to require students to solve a randomly generated test case after being presented with the problem [7, 9]. Only upon solving this test case are they allowed to begin implementing their solution. This was based on the hypothesis that this would facilitate students' metacognitive awareness in the earlier problem solving stages. It was shown that students who worked on the test cases before their implementation showed better problem solving outcomes compared to a control group.

There have also been AATs designed to require students to input test cases together with their solution, forcing them to demonstrate the correctness of their solution. In the AAT used by Edwards [26], students are graded both on their test suite and their solution. Wrenn and Krishnamurthi [27, 28] augmented their AAT with a tool for students to write test cases and evaluate their thoroughness and completeness, which is done prior to implementing the solution to the problem. There was evidence that students used this tool prior to implementation even when not required, suggesting a change in student behaviour as a result of this tool.

3. Context

The motivation to develop a problem solving framework for novice programmers came from teaching *"Digital World"*, a first-year undergraduate programming course at the Singapore University of Technology and Design. Our institution primarily offers four-year engineering degree programmes, which students apply to after completing post-secondary education. A unique characteristic of our programmes is that they share a common first year, meaning that *every* student is required to take our course, regardless

of what they ultimately major in.

Our course covers the fundamentals of computational thinking and programming using Python, and is designed to be accessible to students without any prior experience in the language, taking them from the basics (variables, types, conditionals) through to some introductory object-orientation material. It is delivered using elements of the flipped classroom [29, 30, 31]: before coming to class, students are expected to read some instructor-prepared materials and complete a reading quiz, so that they can be primed and ready to focus on exercises and deeper technical discussions during class. Our materials are hosted on Google Colab [32] (a hosted Jupyter Notebook [33] interface), which allows for code snippets to be run in the browser and thus makes examples more interactive.

As the course is taken by every undergraduate student at our institution, it caters to a wide range of abilities and backgrounds in programming—including none. In previous years, we found that novice programmers were often struggling with where to start on any exercise that went beyond the very basics. Novice programmers would often take a 'syntax-first' approach, where they would type *some* code that they saw earlier without really thinking about what it does, or whether it takes any steps forward towards the solution, eventually leading to tangled-up code that has them demotivated and hitting a brick wall. Observing this pattern year after year led us to design the problem solving framework of this paper: we wanted to encourage such students to reflect, organise their thoughts, think through the problem at an abstract level, and only look for the right syntax/constructs once the problem and solution are clear.

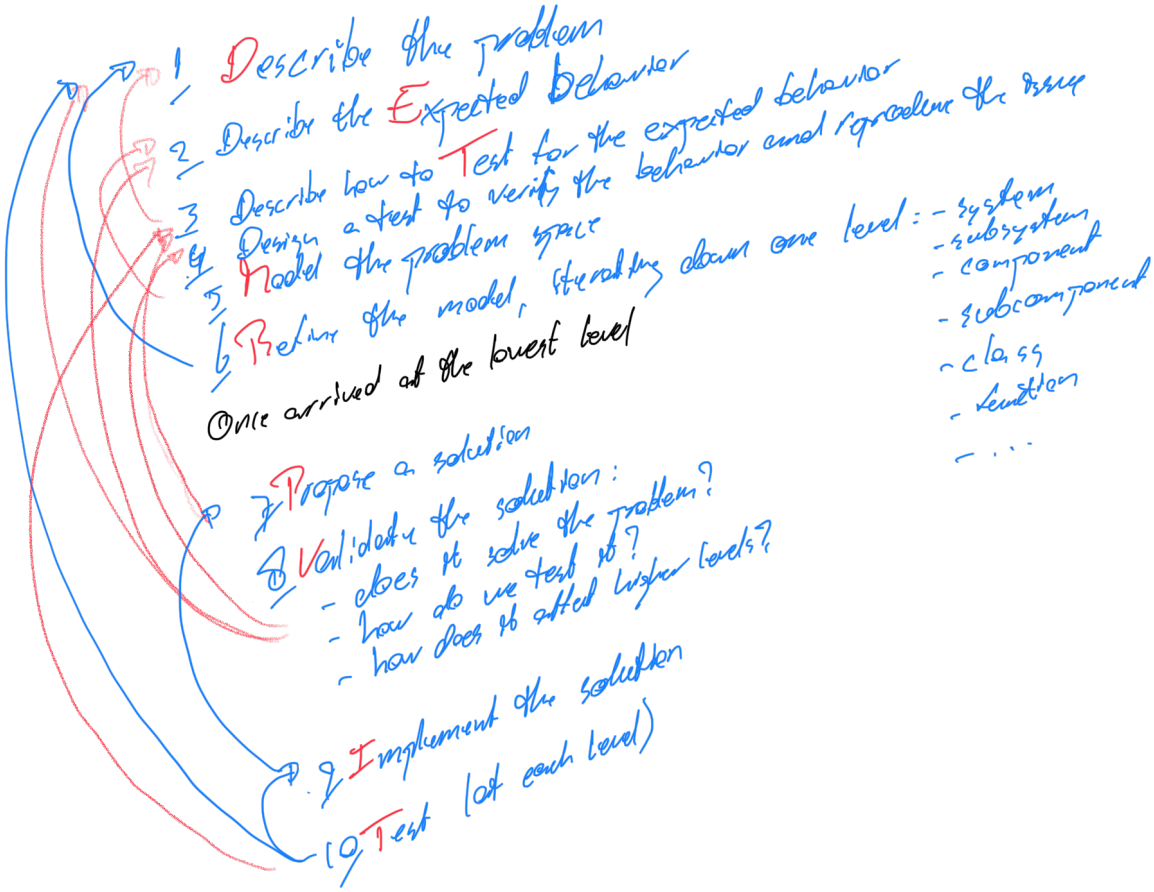
4. Our Intervention: PCDIT

PCDIT Framework. Our pedagogical intervention was the design of the PCDIT framework and the decision to explicitly teach it in our introductory programming course. The five key steps of the framework are given in Figure 1, and are intended to capture the problem solving process that we (as instructors) naturally follow, but that novice programmers are not yet familiar with. By naming the steps and providing an acronym, we make it easier for students to reflect on where they are in the problem solving process, potentially increasing their metacognitive awareness [8, 9]. It is important to note that instructors and students are encouraged to go iterate between the steps as their thinking brings clarity.

In general, the process begins with forming a **P**roblem Definition: students are asked to identify the types of inputs and outputs and summarise in natural language what it is that needs to be solved (e.g. "take a single string value as input, then return the reverse of that string as output"). This step is common to many problem solving frameworks in understanding the problem and formulating it. Similar to Dierbach [3], students are encouraged to provide more detail on the kind of data involved in both the input and the output and how it can be represented in the program. This step also requires students to summarise the problem in a single statement, similar to Riley's 'general description' [13].

The second phase asks students to develop concrete **C**ases, i.e. before even thinking about the algorithm. The intention is for students to conceptualise the abstract steps from concrete inputs to outputs, helping them to generalise to an algorithm more easily in later parts of the framework. The **C**ases step is similar to the functional examples in "How to Design Programs" [14]. As students work on various concrete cases, they can also step back and revise their problem definitions, e.g. adding additional information about the required data types. This step is crucial for novice programmers, many of whom do not have any existing algorithmic patterns or schemas: it may be difficult for such students to search for analogous problems as in Loksa's framework [5]. They need to build the solution from the bottom up and this concrete **C**ases step provides a bridge to figure out the algorithmic solution in the next step. As discussed in Section 2, working out specific concrete cases helps students to understand the problem better, and there is evidence it helps them in implementing their solutions. We highlight that while some frameworks encourage students to write cases as part of their testing code, in PCDIT, they focus on *working out the abstract steps* (e.g. on paper) from concrete inputs to outputs.

Once students have worked on these concrete cases, they can begin the **D**esign of Algorithm phase: for each concrete input/output, we ask them to enumerate the steps they did in working out the concrete **C**ases. They are asked to look back on how they arrive at the output starting from the input. We then ask them to identify patterns in those steps and generalise them to computational steps. These steps can be written in a mix of pseudo-code and (precise) natural language—whatever the student is currently more comfortable with. This part can be iterated several times, starting with more coarse subgoals/descriptions, before refining them over the iterations, e.g. by employing specific *key words/phrases*, such as "for every element in…", "as long as…", or "compare if…". Using specific keywords that sound similar to programming language syntax eases the transition from pseudo-code to actual code later. Figures 2–3 illustrate how one can take some concrete **C**ases for a problem


1/ Describe the problem
2/ Describe the Expected behavior
3/ Describe how to Test for the expected behavior
4/ Design a test to verify the behavior and reproduce the issue
5/ Model the problem space
6/ Refine the model, iterating down one level: - system
- subsystem
- component
- subcomponent
- class
- function
- ...
Once arrived at the lowest level
7/ Propose a solution
8/ Validate the solution:
- does it solve the problem?
- how do we test it?
- how does it affect higher levels?
9/ Implement the solution
10/ Test (at each level)

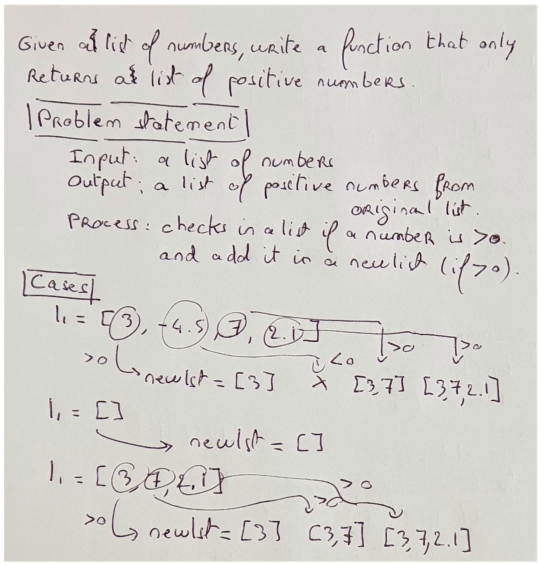


Figure 2. 'P' and 'C' steps examples for list of positive numbers

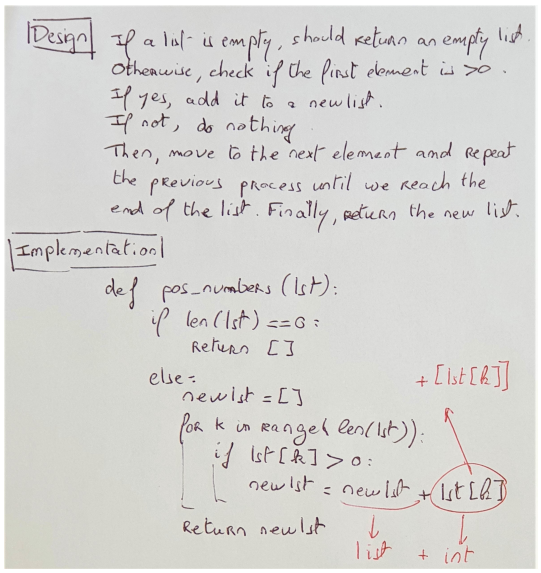


Figure 3. 'D' and 'I' steps examples for list of positive numbers. The initial 'I' step in this example contains an error, discovered in the 'T' step

and start to sketch an algorithm Design in an intuitive (but not yet fully refined) way. (Further examples are available in Examples 1–3 of [34].)

In the subsequent steps, students start to map the pseudo-code of their solution down to concrete Python in the Implementation phase. In our teaching, we iterate this part of the framework with the Testing phase, ensuring that students are regularly testing their programs after completing every few lines of mapping. This helps to ensure novices feel motivated and productive by tackling smaller/feasible sub-problems one-by-one. In testing their code, students can use some of the Cases identified earlier, or propose new ones that potentially highlight the need to go back and improve other aspects of the algorithm further. The Implementation step in Figure 3 actually contains a syntax error in the list operation, illustrating the importance of going back and revising the initial implementation after the Testing step.

We created a PCDIT worksheet that we share with students when teaching the framework [34]. Among our supplementary materials, we include fully worked examples (matrix multiplication, extraction of positive numbers of a list, etc.) using our PCDIT worksheet [34].

Implementation. We now describe how the framework was taught in our first-year programming course (Section 3), and how we elicited the reflections of the students using an optional survey.

The instructors explicitly introduced the PCDIT framework in the third or fourth week of the course, when students began to see more complicated problems, e.g. dealing with loops and a combination of conditionals and iterations, together with the string, list, and dictionary data types. The problems that the instructors used to demonstrate the PCDIT framework involved the synthesis of several concepts such as: (1) manipulating strings using loops and conditionals, and (2) operations involving dictionaries or lists. Fully worked examples of these problems can be found in our supplementary material [34].

In the lesson segment involving the PCDIT framework, after a brief introduction, instructors would show the problem to the students, and show how the PCDIT framework should be used by filling up the worksheet and displaying the process on the projector. Students were reminded that writing the code would only begin *after* some iterations through the first three stages had been satisfactorily completed. The segment then ended with a live coding demonstration showing how the completed PCDIT worksheet could be used in conjunction with the code that was written.

At the end of the lesson segment, an instructor from a different class came to invite students to fill in a survey on their experience of applying PCDIT. The survey was done online and a follow-up interview was available for those who were willing to take part. The survey asked them about their confidence in problem solving and their

computing self-efficacy [35, 36]. We also asked whether the PCDIT framework helped them in their problem solving process and in writing Python code.

5. Survey Results

Our optional survey elicited 47 responses from three classes, indicating a response rate of about 35%. Of these, 25 students were female and 22 were male. Before joining our university, the majority of respondents (66%) had studied at junior colleges, i.e. institutions that offer pre-university courses such as the GCE Advanced Levels or the International Baccalaureate Diplomas. Other students had studied at polytechnics (12%) and international schools (20%). A number of students (38%) had previously written more than 500 lines of code, whereas 40% had written between 50-500 lines, 18% had written between 1-50, and 3% had never written any. Only eight students explicitly reported using Python (the language of our course) in the past; other students reported some experience in C++, Java, and web programming languages (e.g. HTML, JavaScript). Finally, 49% of the students indicated that they were interested in majoring in our information systems or computer science programme, i.e. likely indicating an explicit interest in programming.

The results in Figure 4 indicate that the survey participants show positive interest towards problem solving. The scores in Figure 4(a) tend to be above 3.0 (on a 5-point Likert scale). The results also correlate on the effort that they would put in when solving a problem. Figure 4(b) strongly suggests that most participants put in effort when faced with problem solving tasks. On the other hand, the survey result on their confidence level in solving a problem is not as high as their interest and effort. It can be seen from Figure 4(c) that the overall Likert average score is only slightly above 3 (3.4) for the positive statement "I am sure I can solve a problem". The negative statement "I lose self confidence if I cannot solve a problem" also results in a score of 3.3 which shows the ambivalence of the respondents' self-confidence in solving problems.

Figure 4(d) suggests agreement that the PCDIT framework, after being explicitly taught in class, helped the respondents to solve programming problems and write code. Both of the average Likert scores were about 3.7 out of 5.0. We did not see any respondents choosing score 1 (Strongly Disagree) and saw only one student choosing score 2 (Disagree). This shows that majority of respondents (62%) find the framework helpful in their problem solving and in mapping the solution to code.

In Figure 5, we break down the results further by plotting the PCDIT responses against the respondents' programming backgrounds (i.e. lines of code written before the course). There was only one student who had written 0 lines in the background survey, and that student chose "Undecided" for whether the framework is helpful. Other respondents leaned towards agreement or strong agreement. We also observed an interesting result in that there were those who had written more than 500 lines and yet still continued to strongly agree that the framework helps them in solving programming problems and writing the code.

We also asked students what they found most useful from the framework. A number of students commented on its systematic nature and step-by-step approach. They also mentioned that the framework helps their thinking or thought process. In fact, the pause it forces before writing the code actually helps them in solving the problem itself. Below are a few quotes from the students' comments:

> *When you write out the thinking process it helps to solve the problem.*

> *The pause it forces me to take before solving something; sometimes I can be too anxious and jump to conclusions without having thought carefully about the problem.*

Another point that we see from the comments is related to **T**esting. Quite a number of comments indicated that the framework helps them to better test the code, as shown by the following quotes.

> *Check potential problems.*

> *Testing codes using different test cases.*

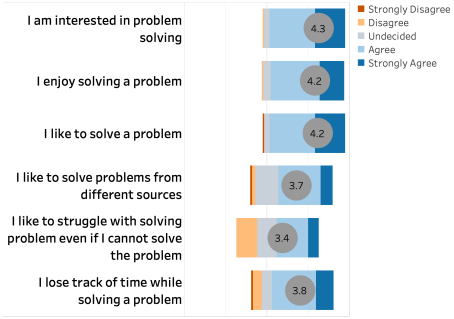
We also asked what they found to be least helpful with regards to this framework. Most of the respondents indicated that it is time consuming. One response explicitly said that such a step may be good for beginners but not for experienced programmers.

> *It's good for beginners, but may be time consuming for experienced programmers.*

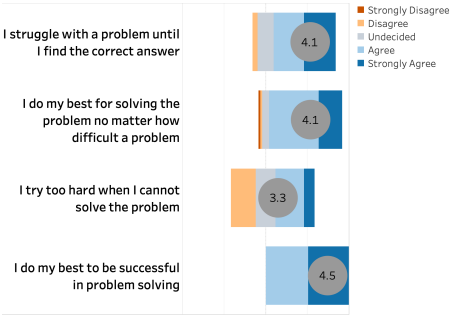
One response commented that they do not know the difference between the **C**ases step and the **T**esting step as seen from the quote below.

> *C:Test cases. Is this not also a part of T:Testing?*

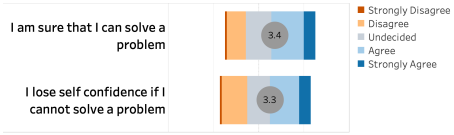
We will take up the survey results, students' comments as well as instructors' reflection in the following section for our reflection and discussion.



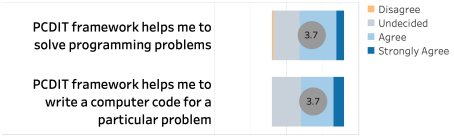
(a) Questions on interest in problem solving



(b) Questions on effort to solve problems



(c) Questions on confidence to solve problems



(d) Questions on the helpfulness of the PCDIT framework

Figure 4. Results (5-point Likert scales) from our post-use student survey on problem solving and PCDIT

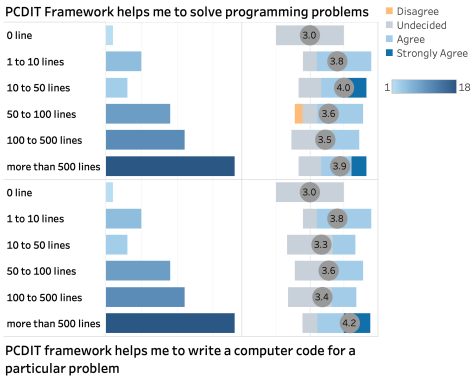


Figure 5. Survey results (5-point Likert) regarding PCDIT plotted against pre-course coding experience

6. Critical Reflections

In this section, we reflect and discuss our experiences of using the PCDIT framework to teach students problem solving in our programming course. First, we reflect on the viewpoints of students who used the framework. Next, we discuss the reflections of instructors who facilitated it during class.

Student Reflections. The survey respondents had varied programming backgrounds. Quite a number of them had actually written some code in the past, with only 20% having written 50 lines or fewer. This likely explains the strong positive scores for interest and effort in our questions. Despite this, we noticed some ambivalence with respect to confidence in solving problems. This means that even though the students tend to enjoy solving problem and have some experience in writing computer code (more than 30% had written more than 500 lines of code), their confidence level in solving a problem is not high. This shows awareness from the students that being able to solve problems is not the same thing as simply writing code.

The breakdown shown in Figure 5 gives some insight into how students' programming background may affect their perception of the PCDIT framework. As mentioned, no student chose 'Strongly Disagree' and only one student chose 'Disagree' when asked whether the framework helps them. In general, the majority of students agreed or strongly agreed that the framework helped them. More students chose 'Undecided' as a response for the question on whether the framework helps them to write computer code as compared to whether the framework helps them in solving programming problems. This suggests that the framework helps more on thinking through the problem solving process. This also could mean, that for most students, they need more help on the problem solving process rather than on finding the right syntax to use.

What is interesting, however, is that the majority of students who had written more than 500 lines of code found the framework helpful. These are students with some programming experience and who may have learnt Python syntax previously. This again agrees with the

observation that the framework's main contribution is on the problem solving process. For these students, mapping a solution to Python code seems not to be one of their main challenges. This also could be the reason why more of them voted favourably about the framework.

These conclusions are strengthened further when taking the students' written comments into account. Most students indicated that the framework helps them in organising their thoughts, in the process of thinking through the problem, and in making their thinking process more systematic. None of them mentioned that it helps them to actually write the Python code. So it is more on the process of problem solving that the framework appears to contribute.

On the other hand, the very same advantage provided by the framework can also be considered one of its disadvantages: the thinking process and the stages of the framework *consume more time*. In the context of a synchronous class or exam, students may be reluctant to use such a method. One way to speed up the application of PCDIT would be to write both the **P**roblem Definition and **D**esign of Algorithm steps in the editor or AAT itself, e.g. as part of the code's comments.

One of the students who volunteered for a follow-up interview gave further insight on how students with experience in programming use this framework. The student recognised that these steps are some of the things he subconsciously does when solving a problem. The framework helps to make the thinking process more explicit and clear. Moreover, he also stated that he found it useful for more complicated problems where the solution is not a one-liner program. This agrees with the purpose of the framework which is to help with the problem solving difficulties that students face when solving more complex exercises. Another interesting point from the interview was that he actually went through the PCDIT process when doing his *group* programming assignment. They started by formulating the problems. Then, they tried to identify the functions by trying to work out the different cases. He explicitly said that the framework was used to facilitate a kind of group discussion or brainstorming process for solutions. This shows that the framework can be used not only by individuals but also as a group in a collaborative work. Further study on how this framework can be applied in a group setting should be explored.

Instructor Reflections. Three instructors involved in teaching the PCDIT framework were asked to share their reflections. All three found that the framework is natural and makes explicit what it is they naturally do when teaching programming. For example, one of them said:

> *I realized that many parts of PCDIT are implicitly built into the problem sets that I designed.*

Another mentioned that the PCDIT structures what it is he does when solving problems himself:

> *The framework is well structured and the acronym seems easy enough to be remembered by the students. I have also realised that the PCDIT approach is actually more or less the approach I am using in practice.*

With respect to the difficulties in teaching using the framework, most of them identified the second step—working on the **C**ases—to be one that students find challenging, and that more time should be spent during class to cover it. One of them stated that most students do not see the difference between the **C**ases and **T**esting stages and more emphasis on their difference should be done in the teaching. One example on how **C**ases and **T**esting stages differ can be found in Example 3 in [34]. In this example, the **T**esting step helps to identify an error when adding an item into a list which is particularly dependent on the programming language syntax and features. On the other hand, the **C**ases step guides the design of algorithm and would not be able to detect such mistakes. Example 4 in [34] shows how **C**ases can be related to the **T**esting. In this example, the same test case is used. Observing the identical output of the print statements and the previously worked **C**ases allows the students to relate the **T**esting of their implementation to their previous **C**ases step.

Two of the instructors highlighted students' reluctance to spend time planning their solutions on paper. One instructor wrote the following reflection:

> *When it comes to the problem analysis, I see very few students trying to figure out the problem on paper first. They often try their luck in code and use a trial-and-error approach, basically trying to figure what could be the code that will satisfy the given test cases. This is not the right approach to coding, and I often have to force students to go back to analyzing the problem on paper.*

Instead of **C**ases, one of the instructors found the **D**esign of Algorithm and **I**mplementation stages to be the main difficulty. He also highlighted that students were unwilling to spend time to plan their solutions before actually implementing them. This agrees with students' comments in the sense that students found the framework to be time consuming. But it is exactly this time spent in planning the solutions that enables students to successfully solve the problem. As one of the instructors put it:

> *I genuinely think that the framework is not only helpful for novice programmers but for any programmer trying to solve a complex task. We actually got a lot of positive feedback for it. However, it is only useful for students who want to use it! The main difficulty consists of convincing the students to work on paper first. Moreover, many of them think it takes too much time doing so.*

The instructors noticed a few pitfalls. In particular, the framework is demonstrated over an example. From a particular case, the instructor explains how it helps in deriving a rough draft of a pseudo-algorithm. For example, to explain how to write a Python program calculating the product of two matrices, after the problem statement, the instructor chooses two simple matrices, multiplies them on the board while orally decomposing the work process. This example helps in drafting a pseudo-algorithm and defining a first test case. However, some students tend to believe that testing this particular case is enough. Instructors should facilitate from the particular case to more general steps by discussing different **C**ases. Moreover, instructors must explicitly highlight the differences between the 'C' and the 'T'. The students should be reminded that the final phase is not only performed to catch the syntactic bugs of their programs or check whether the inputs corresponding to the particular case lead to the expected result. It is also a step to test the limits of their programs and see if it works with a more general case beyond what they did in the **C**ases step. For example, what happens if a list of lists (a typical matrix representation in Python) contains an empty sub-list? What if the values are not all numerical? What if the sub-lists do not have the same length? Encouraging the students to be 'malicious' with their own functions seemed to have an impact on some students who then tried to build 'unhackable' programs.

Moreover, it was recognised that the framework does not reduce the need to know the programming language and its syntax. In fact, an instructor stated that some students still struggled with the **D**esign of Algorithm and **I**mplementation stages. The reason for this difficulty is that many students are not familiar with the syntax. Therefore, as important as the problem solving steps are, the exercises and training on the language itself remain necessary for the whole PCDIT framework to be applied. This agrees with Linn and Clancey [37] who argued that both competencies are necessary.

Open Questions. We are encouraged by the positive results in the survey, and the reflections from our students and instructors, which increase our confidence in the effectiveness of using the PCDIT problem solving framework in teaching. These results and reflections have prompted us to ask a number of further questions, many of which could be addressed in formal studies.

For example: must the framework be facilitated by a human instructor? Can the framework be used by students independently without any help from any instructor or can it only be used with some facilitation? Can the role of the instructor be replaced through other means in order to facilitate students' use of the framework? For example, can a more guided and detailed worksheet be used in this case? Can such facilitation be automated?

Another question of interest is regarding how we should implement and integrate this framework in our lessons? One of the highlighted issues is the perception that the PCDIT framework consumes a lot of time. How should the framework be implemented with such constraints and perceptions? Can we implement the framework in such a way that both students and instructors find the time taken is well spent? Would incorporating it into an AAT address this? How early should the framework be taught?

Lastly, we found that PCDIT might be useful not only for individual approaches to problem solving but also for group discussions when approaching a programming assignment. Further study should be done on how such framework can be used in a group setting and whether modifications are needed in the framework when collaborative learning is in place.

7. Conclusion

In this experience report, we presented PCDIT, a problem solving framework for novice programmers, designed to incorporate recommendations from the literature on improving metacognitive awareness. In particular, it encourages students to design/solve concrete cases well before any programming, and promotes regular reflections across the five main phases of the the process. We described our implementation of it for an introductory programming course, and showed how the framework's **C**ases step and non-linearity help novices to increase their confidence and better manage their cognitive load. In a post-use student survey, 62% agreed or strongly agreed that it helped them to solve programming problems, with several highlighting that PCDIT helped them to organise their thoughts, solve problems systematically, and avoid jumping to conclusions without thinking through the problem first. Our instructors also reflected positively, highlighting a number of benefits, including PCDIT's clearly named steps (helping students reflect on where they are in the problem solving process), and the fact that its scaffolding explicitly mirrors the problem solving process we naturally follow (but that many novices don't). In future work, we plan to explore the use of the framework in the context of collaborative learning, and how its steps can be integrated into our automated assessments tools (similar to [9]).

References

[1] C. Kelleher and R. Pausch, "Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers," *Science*, vol. 37, no. 2, pp. 83–137, 2005.

[2] M. De Raadt, M. Toleman, and R. Watson, "Training strategic problem solvers," *SIGCSE Bull.*, vol. 36, no. 2, pp. 48–51, 2004.

[3] C. Dierbach, *Introduction to computer science using Python: a computational problem-solving focus*. Hoboken: John Wiley & Sons, 2013.

[4] Y. D. Liang, *Introduction to Programming Using Python*. Pearson, 2012.

[5] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," in *CHI*, pp. 1449–1461, 2016.

[6] G. Pólya, *How to Solve It*. Princeton University Press, 1945.

[7] P. Denny, J. Prather, B. A. Becker, Z. Albrecht, D. Loksa, and R. Pettit, "A closer look at metacognitive scaffolding: Solving test cases before programming," *Koli Calling*, no. 1, 2019.

[8] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, and M. Cohen, "Metacognitive difficulties faced by novice programmers in automated assessment tools," in *ICER*, pp. 41–50, 2018.

[9] J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci, "First things first: Providing metacognitive scaffolding for interpreting problem prompts," in *SIGCSE*, pp. 531–537, ACM, 2019.

[10] R. S. Rist, "Schema creation in programming," *Cognitive Science*, vol. 13, no. 3, pp. 389–414, 1989.

[11] A. H. Schoenfeld, *Mathematical Problem Solving*. Academic Press, Inc., 1985.

[12] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," in *ITiCSE-WGR*, p. 125–180, 2001.

[13] D. D. Riley, "Teaching Problem Solving in an Introductory Computer Science Class," in *SIGCSE*, pp. 244–251, 1981.

[14] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018.

[15] R. McCartney, J. Boustedt, A. Eckerdal, K. Sanders, and C. Zander, "Can first-year students program yet? A study revisited," in *ICER*, p. 91–98, 2013.

[16] I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B.-D. Kolikant, J. Sorva, and T. Wilusz, "A fresh look at novice programmers' performance and their teachers' expectations," in *ITiCSE-WGR*, p. 15–32, 2013.

[17] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Workshop on Empirical Studies of Programmers*, p. 213–229, Ablex Publishing Corp., 1986.

[18] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," in *ITiCSE-WGR*, p. 119–150, 2004.

[19] E. Soloway, "Learning to program = learning to construct mechanisms and explanations," *Commun. ACM*, vol. 29, no. 9, p. 850–858, 1986.

[20] O. Muller, "Pattern oriented instruction and the enhancement of analogical reasoning," *ICER*, pp. 57–67, 2005.

[21] D. Ginat, "Algorithmic patterns and the case of the sliding delta," *SIGCSE Bull.*, vol. 36, no. 2, p. 29–33, 2004.

[22] M. J. Clancy and M. C. Linn, "Patterns and pedagogy," in *SIGCSE*, p. 37–42, 1999.

[23] M. de Raadt, R. Watson, and M. Toleman, "Teaching and assessing programming strategies explicitly," in *ACE*, pp. 45–54, 2009.

[24] N. Arshad, "Teaching programming and problem solving to CS2 students using think-alouds," in *SIGCSE*, p. 372–376, 2009.

[25] K. Falkner and E. Palmer, "Developing authentic problem solving skills in introductory computing classes," *SIGCSE*, pp. 4–8, 2009.

[26] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," in *SIGCSE*, p. 26–30, 2004.

[27] J. Wrenn and S. Krishnamurthi, "Executable examples for programming problem comprehension," *ICER*, pp. 131–139, 2019.

[28] J. Wrenn and S. Krishnamurthi, "Will Students Write Tests Early without Coercion?," *Koli Calling*, 2020.

[29] M. J. Lage, G. J. Platt, and M. Treglia, "Inverting the classroom: A gateway to creating an inclusive learning environment," *The Journal of Economic Education*, vol. 31, no. 1, pp. 30–43, 2000.

[30] H. N. Mok, "*Teaching Tip*: The flipped classroom," *Journal of Information Systems Education*, vol. 25, no. 1, pp. 7–12, 2014.

[31] M. L. Maher, C. Latulipe, H. R. Lipford, and A. Rorrer, "Flipped classroom strategies for CS education," in *SIGCSE*, pp. 218–223, 2015.

[32] M. J. Nelson and A. K. Hoover, "Notes on Using Google Colaboratory in AI Education," in *ITiCSE*, pp. 533–534, 2020.

[33] A. Willis, P. Charlton, and T. Hirst, "Developing students' written communication skills with Jupyter notebooks," in *SIGCSE*, pp. 1089–1095, 2020.

[34] O. Kurniawan, *PCDIT Worksheets*, 2021. `https://github.com/kurniawano/pcdit`.

[35] T. Gok, "Development of Problem Solving Strategy Steps Scale: Study of Validation and Reliability," *Asia-Pacific Education Researcher*, vol. 20, p. 1, 2011.

[36] H. Kolar, A. R. Carberry, and A. Amresh, "Measuring Computing Self-Efficacy," *ASEE Annual Conference and Exposition*, pp. 1–7, 2013.

[37] M. C. Linn and M. J. Clancy, "The Case for Case Studies of Programming Problems," *Communications of the ACM*, vol. 35, no. 3, pp. 121–132, 1992.