Chapter 2 & Chapter 3

(This version is not the final version. It can be useful material for writing the final report. The order of material is meaningful.)

**[Classical programming learning ways (edit from the proposal)]**

Writing code, undoubtedly, is one of the core parts of the Science in Computing programme, and consequently, it is also the toughest part for learners, especially beginners. Students are not only required to have the ability to analyze the solving logic of problems, but also they need to be proficient in different coding syntaxes. In this case, students need to make many attempts to solve the programming problems, and they may easily give up during this period. To ameliorate this situation, two classical programming learning methods are introduced into the courses to improve students’ code-writing skills. Specifically, one of the methods is a preparation action for code writing, Coached Program Planning. This method guides students to analyze the problems and design the logical procedures with native-language style pseudocode for solving these problems [2, 3] so that the logical parts of codes can be partially separated from the syntax parts reducing the cognitive burden during coding to some extent. Another method is code tracing, which is to track the changes of variables by hand during the execution of codes [4]. Although this method does not have a direct effect on code writing and cannot be applied to every programming question because of its cumbersome procedure, it, through the accumulation of experience in code reading, still can provide some kind of auxiliary help in improving code writing skills.

**[Previous Parsons problem application]**

Besides the above two programming learning methods, a new way, Parsons problem, was created to prepare students for writing code [5]. Instead of letting students directly write code, Parsons problem provides a set of code fragments – including the solutions to the questions and some distractors (some common errors) - for students to choose from and reorder by dragging and dropping [5]. During this period, students can get some instruction feedback for their reordered answers, and they need to repeat reordering until their answers are one hundred percent correct [5]. This method provides notable help in introductory programming study. To be more specific, the puzzle-like game-style Parson problem can improve students’ engagement and motivation in learning programming [5]. And with prepared code fragments and instant feedback, the levels of difficulty of the questions are reduced, and students are more likely to persist in programming instead of giving up halfway. Besides, Parsons problem can be of use to reduce cognitive load since students are only required to reorder the prepared code fragments instead of writing code directly [1]. And some context (fixed code) also can be provided to students to reduce cognitive load further [6]. In addition, Parsons problem integrates the respective advantages of both Coached Program Planning method and the code tracing method. Specifically, Parsons and Haden picked up an idea to include activity diagrams in the questions’ descriptions to help students to understand the solution logic of problems [5], which has a similar function to Coached Program Planning method. And since Parsons problem also requires students to read and understand the meaning of every code fragment, it also takes advantage of code reading just like the code tracing method. In other words, Parsons problem provides magnificent solution examples for students to learn from, giving them some reference material to think about solution steps when meeting some similar questions. Finally, Parsons problem also has an effect on helping students to cultivate good coding habits. For example, the distractors in Parsons problem can be used to show some improper variable names, which assists students to distinguish good names from bad names and train the habits of using meaningful and conforming naming rules names [5].

**[The benefits of using Parsons problem in Data Structures and Algorithms courses]**

Although Parsons problem makes great success in teaching programming, it is limited to only being used in introductory programming courses, and it has not been expanded in middle-level programming courses, for example, Data Structures and Algorithms courses. ~~From my~~ ~~perspective~~, Parsons problem still can demonstrate its superiority in Data Structures and Algorithms courses. Admittedly, students taking intermediate-level programming courses should be able to write code instead of just rearranging the order of provided answer blocks. However, because of the abstractness and universality of programming in this course, it also is a challenging task for students to write it directly (for example, recursion problems). Thus, it is of the essence to introduce Parsons problem to build a “bridge” for students to grow their capability to write code directly by themselves. But, since there are some differences between introductory programming courses with Data Structures and Algorithms courses, the previous Parsons problem in introductory programming courses does not fit the situation in Data Structures and Algorithms courses, and it is not suitable to apply the previous Parsons problem directly. Consequently, some new ideas should be introduced to Parsons problem. The detailed difference between the two courses and the limitation of the previous Parsons problem will be discussed in the following paragraphs.

**[The difference between two courses]**

There are some major differences between introductory programming courses and Data Structures and Algorithms courses. To be more specific, the programming in introductory courses is simple and concrete. It only requires students to understand codes line by line. And most of the parts in this course are separated and not related so it is almost unneeded for students to compare the differences. By contrast, the programming in Data Structures and Algorithms courses is complex and abstract. It focuses on large block code understanding instead of line code understanding. Students are required to not only understand every line of code but also the specific data structures, the classes with their methods (Object Oriented Programming), the recursion methods for general use, and the algorithms with the corresponding analysis. Besides, there are more similar concepts needed to be compared, for example, stacks and queues, different types of trees, and different algorithms for the same purposes (like bubble sort and selection sort for sorting, and breadth-first search, depth-first-search for searching).

**[New elements involved (need to add a new paragraph)]**

**[The specific question in Data Structures and Algorithms, the limitation of previous Parsons problem and new ideas to solve these limitations]**

(Need to change the description, since the writing logic has been changed)

The differences between introductory programming courses and Data Structures and Algorithms courses lead to the limitation of using the previous types of Parsons problem directly in the Data Structures and Algorithms courses.

To show the limitation in detail, five types of questions in Data Structures and Algorithms courses are summarized from the reference book Data Structures and Algorithms in Python

To show the limitation, we have analyzed the exercises in the book and identified five types of common questions.

To show the limitation, the exercises in the book were analyzed and five types of … were identified.

[7].

**[Object-oriented programming type of questions]**

(Maybe divide this paragraph into two paragraphs, one for showing the question types, one for showing the limitation, and then it is natural to show the new ideas separately)

Maybe need to change the example or add an example of a data structure.

Explain the example in the text

Not unique answer and can not use cut in block to have a unique answer

Change the paragraph to focus the unique answer, Use another example to show that change order of codes is still ok to use

Object-oriented programming is a programming model based on objects, which includes attributes and methods [8]. In CS1, since all the programming parts are Function Oriented Programming, the object-oriented programming concept has never been involved. However, in CS2, this concept becomes one of the focal points, since it is the foundation to build different data structures. Because of introducing the object-oriented programming concept, the previous types of Parsons problem cannot work as well as before. This is because the codes in a class look like tools in a tool chest, which means they are only used when doing specific work and the order of placing tools does not have a big effect on using tools. For this reason, the reordering function of previous types of Parsons problem is significantly weakened. Taking an example (Figure 1) [7], the methods in this CreditCard class are the tools to achieve some functions of a credit card, which means that these methods are only called when needed instead of being executed in sequence. Hence, changing the order of methods does not affect the program execution. ~~And in this example, some methods only need one line to return some attributes so it is a piece of cake to guess the corresponding contents of the methods according to the names of the methods. Thus, in this code, only 12 lines instead of 52 lines are valuable to use Parson problem to reorder. It is evident from this example that the utility of the previous type of Parsons problem is watered down.~~

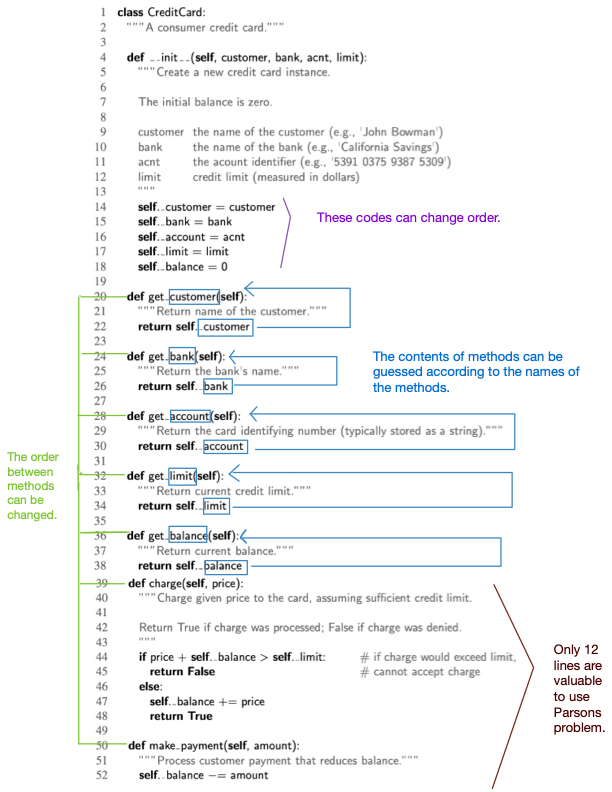


Figure 1 The object-oriented programming example code

[**Object-oriented programming type of questions: new ideas**]

To have a unique answer, some paragraphs describing the functions of the methods should be listed in a specific order in the question description, and students are asked to build methods one by one in a matching order. In this way, a unique order of answers is defined by design. Thus, Parsons problem can not only fit the object-oriented programming type of questions but also keep some of its reordering features at the same time. More importantly, this new design idea enhances students’ understanding of the whole method instead of just every individual line. Further, by this design, students also can have a general idea about what the features of the class used in the question should have, which is of great help when students learn the features of different data structures, in other words, what functions a specific abstract data type (ADT) should support [7]. For example, in figure 2, the left part shows the description of functions of methods of an abstract data type stack in a specific order, and the corresponding methods are implemented in a matching order in the right part.

Graphical user interface

Description automatically generated with low confidence

Figure 2 The description of methods in a stack (left) and corresponding codes (right)

[Add the matching way by using comments to show the method function, not check the whole order, only check matching or not and the order of blocks]

[Find the reference of cutting in block units, and add more information before]

Another way to handle this problem is to check the order of codes inside methods instead of checking the order of method blocks. It looks like the opposite way of cutting in block units. Specifically, cutting in block units prevents multiple correct answers by combining the lines can be ordered in different ways (like a=1; b=2 or b=2; a=1). In this way, it ignores the smaller parts (liens), and focuses on the bigger one (blocks). By contrast, the new design ignores the bigger parts (order between different methods) and focuses on the smaller parts (lines inside the methods). To make this idea more interesting, the description of the function of a method also can be introduced as comments, and let students match the description with the corresponding method codes. (Not talking the same thing)

**[Algorithm analysis type of questions]**

(Maybe divide this paragraph into two paragraphs, one for showing the question types, one for showing the limitation, and then it is natural to show the new ideas separately)

Algorithm analysis, which is a core part of CS2, is to use running time to evaluate whether a data structure or an algorithm is efficient or not [7]. Since the growth rate of running time as a function of the input size n is vital for algorithm analysis, big O, which shows the major parts affecting the growth, is introduced in algorithm analysis [7]. There are seven basic functions in big O (the constant function, the logarithm function, the linear function, the N-log-N function, the quadratic function, the cubic function and other polynomials, and the exponential function) [7] so it is strenuous to ascertain which functions are the correct functions to describe the efficiency of the specific data structures and algorithms. By contrast, the algorithm analysis in CS1 is much more straightforward since only three functions (the contrast function, the quadratic function, and the cubic function and other polynomials) are used and the functions used are only lying on whether there are loop structures or nested loop structures. For this reason, the algorithm analysis parts are always ignored in CS1 so the Parsons problem for CS1 certainty cannot handle this algorithm analysis questions, which is another drawback for using previous types of Parsons problem directly in CS2.

There are seven common big O classes in CS2. Sequential search is O(n)

[**Algorithm analysis type of questions new ideas**]

To involve the algorithm analysis, only operating the code cannot satisfy the situation anymore. Thus, a new component – comment – is introduced in Parsons problem. To be more specific, the comments are used to provide different big O classes for choosing. Students need to select the correct corresponding big O class comments and insert them into codes to do algorithm analysis of the whole code. Unlike the previous reordering questions, this question is more like a multiple-choice question since students can put the comment in wherever they like. Furthermore, this new idea also can be expanded in two ways. One way is to provide not only big O but also the described reasons to get big O. In this way, students can have a deeper understanding of algorithm analysis instead of just guessing big O classes according to their feelings. Another way is to let students match different big O classes with different parts in codes. This way can be used to track the change points of big O classes (like outer for loop and inner for loop) or it can be used to compare different methods in a data structure and let students know the characteristics of a data structure (For example, a heap is quick to get the min value but it is slow to add nodes, shown in Figure 3).

Table

Description automatically generated

Figure 3 Performance of a heap-based priority queue

**[Recursion type of question]**

(Maybe divide this paragraph into two paragraphs, one for showing the question types, one for showing the limitation, and then it is natural to show the new ideas separately)

Recursion is to solve a problem by solving a subproblem that has the same structures as the original problem. In programming, it is achieved by calling the function itself, which means that the same code in a recursion question can solve the same problem but with different values of input parameters. This situation, one code holding different cases, is not common in CS1. For this reason, it leads to a dramatic increase in the difficulty of problems and the previous type of Parsons problem cannot have a significant effect on reducing complexity. Concretely, the previous type of Parsons problem reduces the level of difficulty by providing some code fragments for students to read. However, in the recursion type of question, it is arduous to understand the meaning of every line of code even if the whole code is given in order (like the recursion example code of the Tower of Hanoi in Figure 2 [9]). Thus, the assistance coming from code reading of Parsons problem is notably shrunk, and it is almost like letting students write code directly. That is why the previous type of Parsons problem needs to be improved to give students more hints in solving recursion questions.

Def fib (n):

If (n==1, n==2) :

Else:

Return fib(n-1) + fib(n-2)

|  |  |
| --- | --- |
| Def fib3(n): # n>2  A = fib(n-1)  B = fib(n-2)  Return A+B | A = fib(0)  A = fib(n)  A = fib(n-1)  B = …  Return a+b. return a-b |
| Def fib1(n): # n<=2 |  |

Def fib (n)

For ( ) {

Ans = previous fib + previous previous fib

}

Double(1); double(x)

Text

Description automatically generated

Figure 2 The recursion example code of the Tower of Hanoi

**[The recursion type of questions new ideas]**

According to the suggestions for designing recursive algorithms provided by the textbook, it is beneficial to find a few specific subproblems which have the same structures as the original problems [7]. To achieve the suggested way, a new pre-scaffold way – organizing in several steps – is introduced into Parsons problem. To be more specific, one recursion question is divided into some steps. The first step is to let students reorder the code of base cases (the end point of every recursion chain [7]). Subsequently, change the values of parameters used in this recursion algorithm to have other subproblems. And in this step, the previous base cases should also be involved. In other words, students need to build these subproblems by calling the method in the base cases. In the following step, students should choose the correct method headers with proper signatures. In the final step, students are required to reorder the real recursion codes with all the previous codes as references. This new pre-scaffold way can be partially helpful for students to solve the recursion problems. In this new way, students are provided with more hints (like subproblems) than directly solving recursion questions. Thus, they are less like to be all at sea.

An example should be added here to show the steps (Fibonacci sequence)

In addition, a code-tracing-like way can be used in Parsons problem to do the recursion trace. Specifically, students can track the recursion by ordering the comments of the results of the recursion calls. This method also can be used in the recursion questions performed on the tree data structures since an additional requirement (like showing the tree in preorder traversal or postorder traversal) can be used in the questions. In this way, students can have a clear picture of the flow of recursion calls and they also can have a better understanding of how to manipulate a tree data structure (like the sift-down algorithm in heap).

An example should be added here (about sift-down algorithm)

**[The data structure and algorithm types of questions]**

(Maybe divide this paragraph into two paragraphs, one for showing the question types, one for showing the limitation, and then it is natural to show the new ideas separately)

One of the characteristics of code in CS2 is similar to each other. For example, the codes can have the same data structures but are implemented in different ways (like the array-based stack and linked-list-based stack in Figure 3) or different solving algorithms for the same problems (like bubble sort and selection sort for sorting numbers problem). Although it is ok to use the previous types of Parsons problem individually for each code, it would be more worthwhile to help students to compare these similar codes and consolidate the difference and similarity between these codes. Therefore the students can distinguish them and have a better understanding.



Figure 3 The array-based stack and the linked-list-based stack

**[Comparing type of questions]**

To compare the same data structures implemented in different ways, the transforming learning method is used. In other words, two different codes with some common parts are provided, and students need to choose the common parts from one of the codes using these common parts to build another one. In this way, students can understand new knowledge by transforming what they have learned before. Taking the two implementation ways of a stack as an example, students can drag and drop the method headers in an array-based stack as some components to build a linked-list-based stack. In such a manner, students can have a general idea about what the common parts are (the methods supported in a stack abstract data type) and what differences are (implementation details caused by array structure or linked list structure). Besides, the completed ordered array-based stack codes also can be hints to support students to build the linked-list-based stack.

An example should be added here

To compare different algorithms of the same question, two algorithms are mixed by using distractors, and students are supposed to distinguish and order each of them from the mixing code pool. To be more specific, unlike previous distractors, which are all used to show some incorrect or improper code [5], a group of correct complete codes of an algorithm is introduced as distractors in order to mix with another algorithm. In other words, the jumbled code fragment pool includes two different correct algorithms, and students should pick up, rearrange, and submit the code fragments in these two algorithms separately. This setting method is significantly applicable in the CS2 since there are a lot of questions having this characteristic – one question with several solving algorithms, for example, bubble sort, selection sort, insertion sort for sorting numbers, and breadth-first search, depth-first search for searching. In this way, it can help students to distinguish similar algorithms in the same categories mentioned before, which is worthwhile when students have learned more than one algorithm and begin to use them motley because of blurry memory.

An example should be added here

**[Balance the degree of difficulty]**

The key function of Parsons problem is to add a preparation stage before students actually begin to write codes. This preparation stage is achieved mainly by reducing the degree of difficulty caused by directly writing codes. In other words, one of the key issues of Parsons problem is to find a balance point in the degree of difficulty, which is easier than writing codes but still valuable to practice. To fulfill this demand, three different methods (selecting difficulty level, pre-scaffold, and content) are introduced in the following paragraph.

**[Selecting difficulty level]**

In the previous Parsons problem, no matter whether it is pre-scaffold or student-scaffold, the problem always is predefined with a fixed difficulty level, which means that students cannot ask for further help with additional hints if they meet problems in solving this Parsons problem. Since the codes in CS1 are quite simple, this problem may not happen frequently, and it may not affect students solving problems. However, due to the improvement in the complexity of the codes in CS2, this problem begins to become a huge issue. To handle this issue, multiple difficulty levels are set, and students are allowed to switch to an easier version of this question halfway. For example, in Figure [], three different difficulty levels are set. Specifically, Level 0 is to just cut codes in lines and allow students to reorder them. Subsequently, in Level 1, the headers of methods and predefined description comments are placed in the correct positions for hints. In Level 2, the codes are grouped by methods so that students can only reorder smaller parts of codes for every method instead of selecting from the whole mixing codes pool. In Level 3, some codes in methods are fixed as context and only some core parts need to be reordered. If students find the present version of Parsons problem too difficult, they can change to an easier version.

[An example needs to be added]

**[Pre-scaffold]**

Pre-scaffolding is to give students some structures for hints. This kind of method to handle the difficulty has already been used in the existing Parsons problem, like providing loop structures. This method is also used in CS2 courses but with some extension. To be more specific, pre-scaffolding can be achieved by separating codes by different methods like the example shown before. In addition, pre-scaffolding also can be achieved by separating codes into different steps like the example shown in recursion questions. Finally, this method also can be achieved by separating codes according to different subgoals helping students to divide a huge, complex problem into smaller simple questions.

[An example needs to be added]

**[Context]**

Context is used to provide some ordered codes as information and let students only reorder some smaller parts in the middle of these ordered codes. Context can be done a step further, which means that students only need to insert some key code fragments into the right location of the whole code file. And to keep the question valuable, some wrong codes are also provided to confuse students. This kind of question is extra useful in iterator and recursion questions.

[An example should be added]

**[Game-like experience]**

To make the procedure of solving questions more interesting, some game design ideas are added to this project. Not only game-design elements like colorful icons and rewards are included, but also some design model liking freezing time before asking another feedback are used. Like the following example…

[An example should be added]

**[References]**

[1] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Research on Parsons Problems. In Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE’20), February 3–7, 2020, Melbourne, VIC, Australia. ACM, New Y ork, NY , USA, 8 pages. https://doi.org/10.1145/3373165.3373187

[2] Rita Garcia, Katrina Falkner, and Rebecca Vivian. 2018. Scaffolding the Design Process Using Parsons Problems. In Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling ’18). ACM, New York, NY, USA, Article 26, 2 pages. https://doi.org/10.1145/3279720.3279746

[3] H. Chad Lane and Kurt VanLehn. 2003. Coached program planning: dialogue-based support for novice program design. SIGCSE Bull. 35, 1 (January 2003), 148–152. https://doi.org/10.1145/792548.611955

[4] Code Tracing. Retrieved September 7, 2022, from https://microcredentials.digitalpromise.org/explore/code- tracing

[5] 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 (ACE ’06). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. <http://dl.acm.org.ezproxy.auckland.ac.nz/citation.cfm?id=1151869.1151890>

[6] Garner, Stuart. “An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming.” Proceedings of the 2007 InSITE Conference, 2007, <https://doi.org/10.28945/3127>.

[7] Goodrich, Michael T., et al. Data Structures and Algorithms in Python. Wiley, 2018.

[8] “Object-Oriented Programming.” Wikipedia, Wikimedia Foundation, 16 Oct. 2022, <https://en.wikipedia.org/wiki/Object-oriented_programming>.

[9] “Python Program for Tower of Hanoi.” GeeksforGeeks, 16 June 2022, <https://www.geeksforgeeks.org/python-program-for-tower-of-hanoi/>.

Search() returns 0

Def search(nums, target):

For pos in range(len(nums)):

If nums[pos]==target:

return pos

Return -1

Nums = [1,2,3,2,1]. Target 1. Count() returns 2

Def count (nums, target):

Ans = 0

For pos in range(len(nums)):

If nums[pos]==target:

Ans =ans+1

Return ans

Block 1, 2, 3, 4,5,

1: print(n)

2: n=n+1

Q1-solution.

“[ 1,2,5,10 ]”