**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)

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.

[**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

**[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)

**[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