CodeTree: A System for Learnersourcing Subgoal Hierarchies in Code Examples

HYOUNGWOOK JIN, School of Computing, KAIST, Republic of Korea
JUHO KIM, School of Computing, KAIST, Republic of Korea

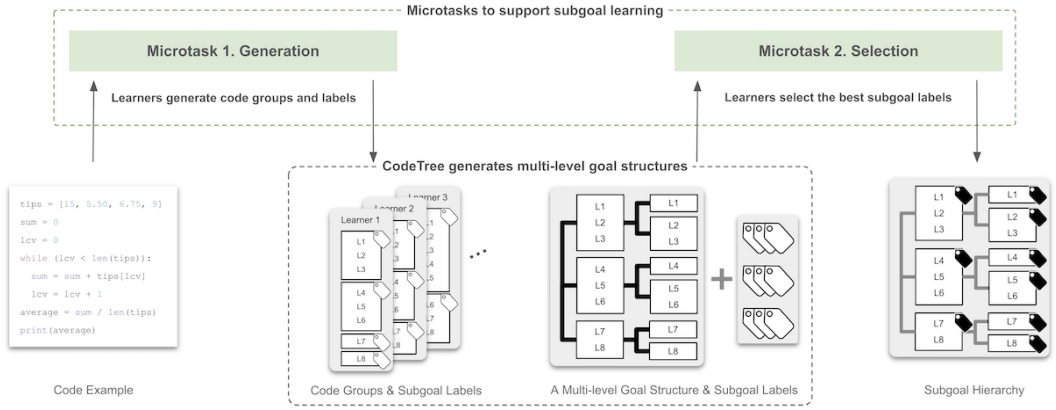


Fig. 1. Our learnersourcing workflow for generating subgoal hierarchies. For each code example, learners in the *Generation* task generate the code groups and subgoal labels that can constitute a subgoal hierarchy. Our hierarchy generation algorithm aggregates the code groups into a multi-level goal structure. Learners in the *Selection* task receive the goal structure and pre-populated subgoal labels to vote for the best subgoal labels for each subgoal in the hierarchy.

Subgoal-labeled code examples help learners understand code patterns and apply them to different problem contexts. Subgoal labels are multi-level in nature and based on goal structures that define the hierarchical functional units in code. Data-driven methods and experts can supply the goal structures, but they do not work in environments with scarce data and limited availability of experts. Previous research has shown that learnersourcing is effective for sourcing high-quality subgoal labels of given goal structures. We extend this research by learnersourcing goal structures themselves, thereby making the generation of subgoal-labeled materials fully learner-driven. We introduce CodeTree, a system that generates multi-level goal structures by aggregating learner-generated subgoals from two subgoal learning activities—*Generation* and *Selection*. In a between-subjects study, 45 novices studied three code examples with either CodeTree or code explanations alone. The results showed that CodeTree could learnersource high-quality goal structures and subgoal labels for all three examples with just five learners. Learners reported a significantly higher learning gain and satisfaction compared to the baseline.

Authors' addresses: Hyoungwook Jin, jinhw@kaist.ac.kr, School of Computing, KAIST, Daejeon, Republic of Korea; Juho Kim, juhokim@kaist.ac.kr, School of Computing, KAIST, Daejeon, Republic of Korea.

XXXX-XXXX/2023/9-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts: • **Human-centered computing → Collaborative and social computing systems and tools**.

Additional Key Words and Phrases: crowdsourcing, education/learning, artifact or system, quantitative methods

ACM Reference Format:
Hyoungwook Jin and Juho Kim. 2023. CodeTree: A System for Learnersourcing Subgoal Hierarchies in Code Examples. 1, 1 (September 2023), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Programming is becoming one of the most valuable skills to learn. As the programming population grows, online resources such as documentation, how-to videos, and Q&A websites have become popular for learning and help-seeking [1]. Code examples are typical materials used in programming learning resources. Code examples are short code snippets that demonstrate instantiations of code patterns under specific problem contexts (*e.g., calculating the average value of coins in a pocket*). Since the exemplified problem contexts are often different from the diverse problem contexts that programmers face in practice (*e.g., calculating the average of positive values in an array*) [63], practitioners and learners need to spot and modify parts of code examples to adapt them to their problem contexts. Hence, the educational purpose of code examples is not to have learners simply copy code, but to reduce their cognitive load and provide the means to learn and transfer code patterns to novel problem contexts [16].

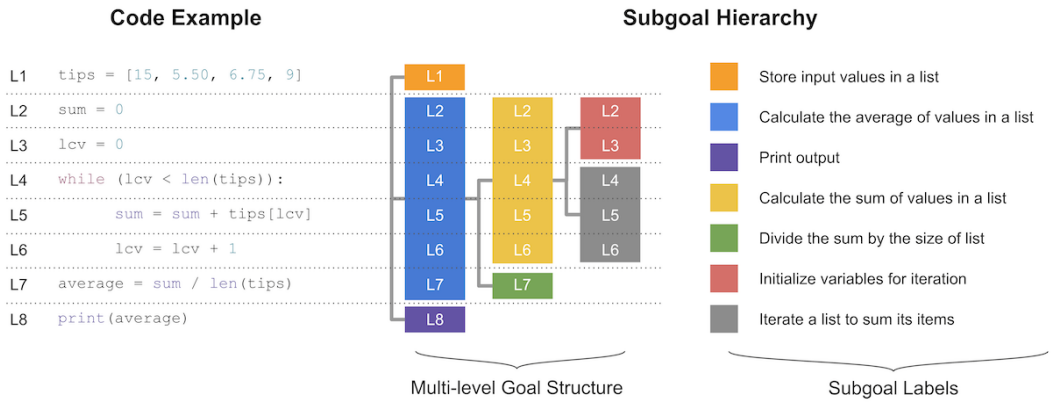


Fig. 2. A subgoal can group a set of related code lines by their function. The subgoals on the left of the hierarchy are coarse-grained goals that explain high-level functions that span multiple lower-level subgoals (or code lines). The subgoals on the right are fine-grained goals that explain the code line by line.

Education research has shown that learning subgoals in code examples can effectively scaffold learners' transfer to novel problems [6, 7, 20]. Subgoals are functional units that divide code into smaller pieces and help learners navigate code to find the part to modify for different problem contexts. Subgoals often form a hierarchy with high-level strategic goals and low-level constituent goals comprising the high-level goals. A subgoal hierarchy refers to a hierarchical organization of subgoals in code and consists of a goal structure and subgoal labels (Fig. 2). Subgoal hierarchies are used to explain the function of each part of code [37, 45], to create the materials for diverse learning activities [15, 36], and to generate adaptive explanations [26]. Goal structures are a fundamental

[1] https://insights.stackoverflow.com/survey/2018

component of subgoal hierarchies to facilitate these learning supports as they set the frameworks for organizing and mapping subgoals to code.

The generation of goal structures has been dependent on expertsourcing and data-driven methods. Conventionally, instructors and domain experts take an iterative process to generate subgoals [10], but this is limited in terms of scalability as the process is time-consuming. To overcome the limitation, data-driven methods have been applied to learning environments where large code datasets are available (e.g., Scratch) [5, 41, 49]. These methods compare learners' code submissions for a problem at scale and identify common code patterns as subgoals of the problem. However, data-driven methods do not work in learning environments where code data per code example is scarce. For instance, although StackOverflow has many code examples, more than half of their questions are answered with less than ten code examples, which is not a feasible scale to adopt data-driven approaches [3].

To support the subgoal learning of code examples in ubiquitous environments, we propose using learnersourcing to generate goal structures without dedicated experts or large-scale code data. Learnersourcing is a scalable crowdsourcing technique that leverages learners' creativity and knowledge to create learning resources for future learners [52, 61]. Prior research has shown that learnersourcing could effectively reduce experts' effort in subgoal label generation by offloading certain tasks to learners [15, 61]. We extend this line of research further by proposing an approach to also offload the effort of generating goal structures to learners. Compared to learnersourcing of subgoal labels, generating goal structures is new and challenging because it has to build the structure from scratch while label-sourcing works on top of a given structure. Despite the challenge, learnersourcing of goal structure is fundamental to prior subgoal label generation approaches for making the generation of subgoal hierarchies completely learner-driven and scalable in data-scarce environments.

We built a prototype system, CodeTree, to investigate the feasibility of learnersourcing the generation of goal structures without expert intervention. CodeTree is a tool for studying code examples by carrying out two learning activities that ask learners to either 1) group code lines into subgoals or 2) vote for the best explanations as the subgoal of a given code. These activities elicit self-explanation of goal structures as learners should make sense of the relationship between code and the subgoals they make or vote on. CodeTree leverages learners' responses to these activities to generate and control the quality of multi-level goal structures and subgoal labels. After enough learners' responses are populated, CodeTree algorithmically aggregates learners' partial but complementary subgoals to generate comprehensive multi-level goal structures (Fig. 2). Although our main aim is to construct goal structures for code examples, we added the features for collecting subgoal labels to the system to support end-to-end subgoal hierarchy generation.

We evaluated the feasibility of CodeTree for learnersourcing subgoal hierarchies. Our evaluation study with 45 Python novices showed that 1) CodeTree could learnersource correct subgoal hierarchies for three code examples with just five learners, 2) studying code examples with CodeTree resulted in a higher learning gain in code tracing skills (with standardized effect sizes of around 0.7) and higher satisfaction than with explanations of code alone, and 3) the user interfaces and visualizations in CodeTree helped learners understand and generate subgoal hierarchies by enhancing the visibility of the mapping between subgoals and code, and by providing an overview of the subgoal hierarchies.

Our primary contributions are summarized as follows:

- A learnersourcing workflow and algorithm for generating multi-level goal structures of code examples.

- CodeTree, a system that embeds the workflow and provides user interfaces that visualize subgoal hierarchies to scaffold generating and learning of subgoals in code examples.
- Empirical evidence that CodeTree can populate high-quality subgoal hierarchies with just five learners while improving their code tracing skills and learning satisfaction.

2 RELATED WORK

This research aims to support the adoption of subgoal learning on code examples at scale. We use learnersourcing as an approach to generate goal structures needed for subgoal learning in learning environments where experts and data are scarce. This section reviews previous literature on 1) subgoal learning and its practices, and 2) learnersourcing systems of different forms.

2.1 Subgoal Learning

A subgoal refers to a conceptual action or state that is found in the process of achieving a higher level (sub)goal in problem-solving [11]. Subgoals organized in hierarchies provide a useful mental model of task structures for decomposing tasks into subtasks [6]. Subgoal learning is a pedagogy that teaches learners to discern task structures and constituent subgoals in worked examples so that learners can modify and apply the task structures to novel problems [9].

Early research on subgoal learning focused on finding effective presentations of subgoals in mathematics and physics worked examples. Catrambone showed that subgoal labeling, which adds subgoal labels to worked examples, can scaffold learners' transfer from worked examples to novel problems [6, 9, 11]. Catrambone also tested different variants and found that visual isolation of steps and problem-independent subgoal labels can elicit the transfer even further [7, 8].

Later subgoal learning research primarily took place in the programming domain and looked into the different learning effects in depth. Margulieux and Catrambone found that subgoal-oriented instructional materials in programming improve learners' problem-solving performance [35, 37]. Margulieux et al. applied subgoal learning frameworks to introductory programming courses for a semester and found that subgoal-labeled materials also increase learners' retention of knowledge and courses [39, 40]. They also looked into the usefulness of learners' self-explanations of subgoals during initial problem-solving and observed that self-explanations are as useful as expert-generated labels [34]. Ericson et al. used subgoal labels in Parsons problems to make them more effective for testing learners' performance and understanding of code structure [19].

Researchers also studied subgoal learning in different activity types. Based on Chi's active-constructive-interactive framework of learning [13], the constructive method of subgoal learning, in which learners learn by creating subgoal labels by themselves, has been proposed and investigated. Compared to the passive and active methods that give learners expert-generated subgoal labels, the constructive method can help learners acquire more transferrable knowledge by promoting creative thinking and self-explanation. Extensive research has shown that the learners who practiced the constructive method outperform learners with either passive or active methods for basic programming and app inventing tasks [36, 44, 45].

Our work is founded on the findings in the previous research. Since we want our learnersourcing tasks to be pedagogically meaningful, we followed the constructive and active methods of subgoal learning to design our tasks. Hence, one of our evaluation metrics is how much our system replicates the previous pedagogical effects. By learnersourcing goal structures, we envision that our system will enable the application and investigation of subgoal learning in broader environments.

2.2 Active Learnersourcing

Learnersourcing is a type of crowdsourcing that leverages learners' responses in their learning activities to generate meaningful data for future learners [27]. Learnersourcing has advantages over

expertsourcing in terms of scalability because it can draw a workforce from large-scale learning environments, such as MOOCs and Q&A websites, and learners are often motivated to participate in learning activities without monetary rewards. In return, learnersourcing requires reliable quality control mechanisms on learner-generated data because learners are inherently less knowledgeable than experts. For effective quality control, learnersourcing often accompanies majority voting [1, 29] and automated-methods [48].

There are largely two types of learnersourcing—passive and active—depending on how data are generated. Passive learnersourcing uses readily acquirable data from natural learning processes, such as learners' interaction logs and code submissions [25, 28]. Since the data size is often large, automated methods are used to analyze and create meaningful learning supports [49, 60]. When target data is not readily available, active forms of learnersourcing are used. Active learnersourcing adds new learning activities to conventional learning processes to ask learners to generate specific data [23, 42, 62, 64]. One of the challenges in active learnersourcing is to design the activities to be pedagogically meaningful and easy to attempt to encourage the voluntary participation of learners. The activity designs often follow well-defined pedagogies and microtask workflow to improve the learning experience and reduce learners' workload.

There has been research on using active learnersourcing to generate subgoal labels for how-to-videos, mathematics, and algorithmic problem-solving. Crowdy [61] learnersourced subgoal labels for how-to videos by periodically asking learners to self-explain the goals of video sections while watching. Crowdy used a generate-evaluate-proofread workflow to divide the label generation process into manageable microtasks and to ensure the quality of subgoal labels through multiple checks. SolveDeep [26] gathered solution graphs of mathematics problems by asking learners to group the steps in their solutions by subgoals and explain them. Collected solution graphs are then used to generate feedback on subgoals that future learners make. AlgoSolve [15] used active learnersourcing to collect subgoal labels of code for algorithmic problem-solving. AlgoSolve used a vote-label workflow to familiarize learners with high-quality subgoal labels first. Collected subgoal labels are used to scaffold future learners to plan their solutions.

Our work extends the line of research on learnersourcing subgoals. Previous learnersourcing systems have focused on generating subgoal labels for goal structures that experts pre-defined. Although these systems successfully reduce experts' burden to complete subgoal hierarchies, their scalability depends on experts creating goal structures because the creation of goal structures needs to precede subgoal labeling. Our work empowers previous label-sourcing systems to be truly scalable with a learnersourcing workflow and coordination algorithms for making the generation of subgoal hierarchies fully learner-driven.

3 DESIGN GOALS

Based on prior work, we set three design goals for learnersourcing goal structures and supporting subgoal learning of code examples. Our design goals touch upon the type of subgoal hierarchies to generate, learnersourcing workflow design, and a versatile visualization to help learners generate and learn subgoal hierarchies.

G1. Generate multi-level goal structures.

Theoretically, a code example can have multiple instances of goal structures. Goal structures may vary in the granularity of constituent subgoals and depth. Among the many possible instances, we specifically aim to generate multi-level goal structures that are useful for subgoal labeling [35], self-explanation activities [15, 36], and feedback generation [26]. Having multi-level goal structures is especially useful for making these use cases more adaptive to learners. For example, learners

with prior knowledge can receive feedback and questions for high-level subgoals, while less-experienced learners can start with low-level subgoals to understand smaller code patterns. Previous studies also showed that single-leveled subgoals hardly fit all learners with diverse background knowledge [7, 8, 17, 50]. Hence, to maximize the sensitivity of the levels and their benefits, we aim to populate subgoals at as many levels as possible and organize them into multi-level goal structures.

G2. Divide the generation task into microtasks while not compromising the learning objective.

Generating multi-level subgoal hierarchies from scratch is a complex task that may frustrate individual learners. In crowdsourcing literature, dividing a complicated task into manageable microtasks is shown to reduce cognitive demand and improve crowd workers' performance [4, 31]. When designing microtasks in learnersourcing, one of the key considerations is to keep the size of the microtasks small enough so that learners can easily attempt it, but at a level that does not compromise the learning objective [52]. For example, one possible microtask design is to assign learners different parts of code to generate multi-leveled subgoals. However, this may compromise the learning aspect of seeing code examples, as learners will not get enough chances to understand the entire code. Hence, we aim to break the hierarchy generation task into manageable units but ensure that each microtask helps learners skim and understand entire code examples.

G3. Provide learners with a visualization for an overview of subgoal hierarchies.

Subgoal hierarchies are complex data structures that connect code, goal structures, and subgoal labels (Fig. 2). Learners are not typically familiar with generating such complex subgoal hierarchies. Previous crowdsourcing research showed that visualizing the overview of worker-generated data improves their performance and efficiency in complex annotations and graph generation [24, 31, 56]. Visualization of subgoal hierarchies during subgoal generation tasks can also alleviate learners' difficulties by raising awareness of the data they generate and possibly improve. The visualization can also aid in learning the organization of the goal structures and how each subgoal instantiates to a specific code. Hence, we aim to add a versatile visualization that can scaffold both generation and learning of subgoal hierarchies to our interface.

4 SYSTEM

We built CodeTree, a learnersourcing system that generates high-quality subgoal hierarchies while supporting subgoal learning of code examples. Learners can use CodeTree to enhance their understanding of existing code examples by either generating subgoals of the code on their own (Fig. 3) or selecting the best descriptions for given subgoals (Fig. 4). After populating enough subgoals from learners, CodeTree algorithmically aggregates the subgoals into comprehensive subgoal hierarchies. The following subsections describe the user workflow, interfaces, and our algorithm in detail and explain how they achieve the three design goals.

4.1 Microtasks for subgoal learning

We divided the hierarchy generation task into two microtasks—*Generation* and *Selection*—taking G2 into account (Fig. 1). In *Generation* task, learners self-explain subgoals of code examples by grouping code lines into functionally meaningful units and describing each unit. In *Selection* task, learners solve multiple choice questions (MCQs) that ask for selecting the best label for each subgoal. We chose *Generation* and *Selection* tasks as our microtasks because each task follows the constructive and active methods of subgoal learning [13, 36]. Each task is complete on its own in terms of helping learners explore the entire code while dividing learners' workload to generate complete subgoal hierarchies from scratch.

4.1.1 Microtask 1: Generate code groups and subgoal labels. In *Generation* task, learners self-explain the functions of each part of code by generating subgoals on their own. Learners first read problem statements and code examples to check problem contexts and solutions. Then, learners use the hierarchy generation interface to generate and organize subgoals. Learners can generate subgoals by 1) creating an empty subgoal either at the root or below other subgoals at Fig. 3 (C), 2) clicking lines of code at Fig. 3 to add to the subgoal as a group (B), and 3) write a subgoal label that explains the group at Fig. 3 (C). Although learners do not receive feedback on their subgoals, *Generation* task can be helpful as learners explicitly self-explain functions and structures of code [34, 54].

Through *Generation* task, CodeTree collects diverse code groups (i.e., groupings of code lines) and subgoal labels. Each learner outputs a list of code groups and subgoal labels. We expect the lists to reflect learners' diverse perspectives [50] and contain subgoals at different levels that can serve as the basis for generating multi-leveled hierarchies (G1).

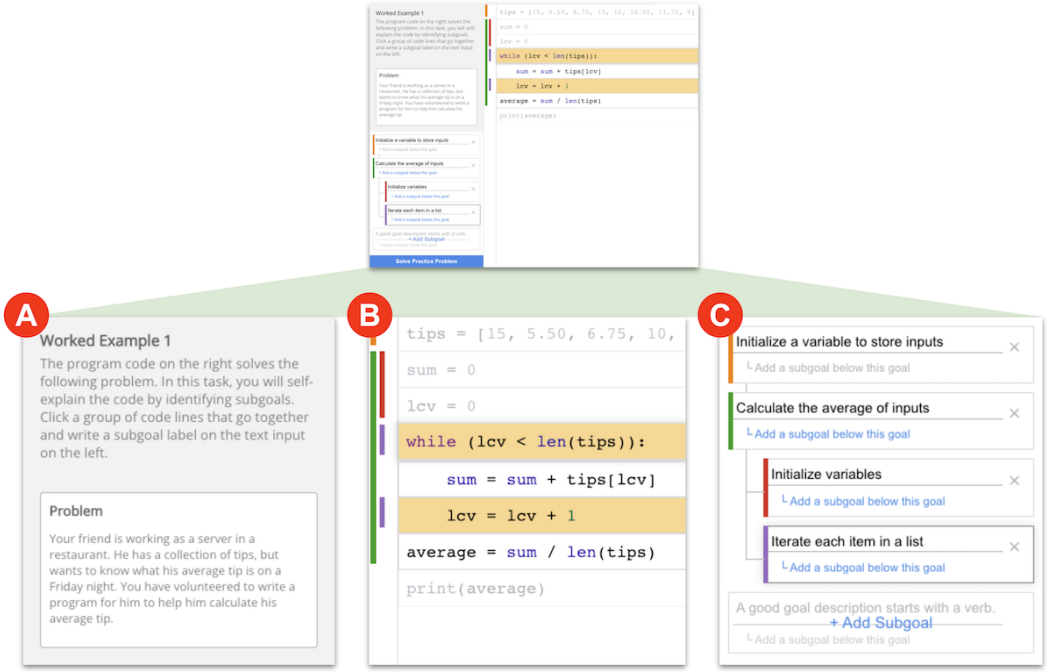


Fig. 3. The user interface for the generation task: (A) Instructions and problem statement, (B) A code example to study. Learners can click and select lines of code to make a code group (currently selected lines are highlighted in orange). Code lines are dimmed to gray and become unselectable if they are either already grouped or outside of parent subgoal scopes, (C) Hierarchy generation interface. Learners can write down subgoal labels for each code group and can add lower-level subgoals.

4.1.2 Microtask 2: Select subgoal labels that best explain constituent code groups. In *Selection* task, learners self-explain the function of each code group in a given goal structure by selecting the best descriptions. Similar to *Generation* task, learners first read problem statements and code examples. Then learners answer a series of MCQs given by CodeTree to check their understanding of the code. Each MCQ has at most three options to choose from, and learners can add a new one if none looks plausible or if there is a better description. After solving each question, learners receive corrective feedback on their answers (Fig. 4 (C)) to confirm their understanding of the code. Again, colored

bars (on Fig. 4 (B)) visualize given subgoal hierarchies and highlight positions of code groups asked by MCQs (G3).

Each MCQ has two answers and a distractor (wrong answer). CodeTree generates MCQs based on the learner-generated subgoal labels from *Generation* task. The answers are chosen from the subgoal labels that previous learners created for the code group being asked. The distractor is also chosen from learner-generated subgoal labels but from another random code group that is mutually exclusive. CodeTree selects distractors based on our heuristic assumption that previous learners would not write interchangeable subgoal labels for two mutually exclusive code groups. We regard choosing two answer options as a multi-armed dueling bandit problem [65] and use a round-robin and greedy algorithm to balance the exploration for good labels and the provision of the best labels known so far. The reward of the problem is defined as whether learners select either of the answer options.

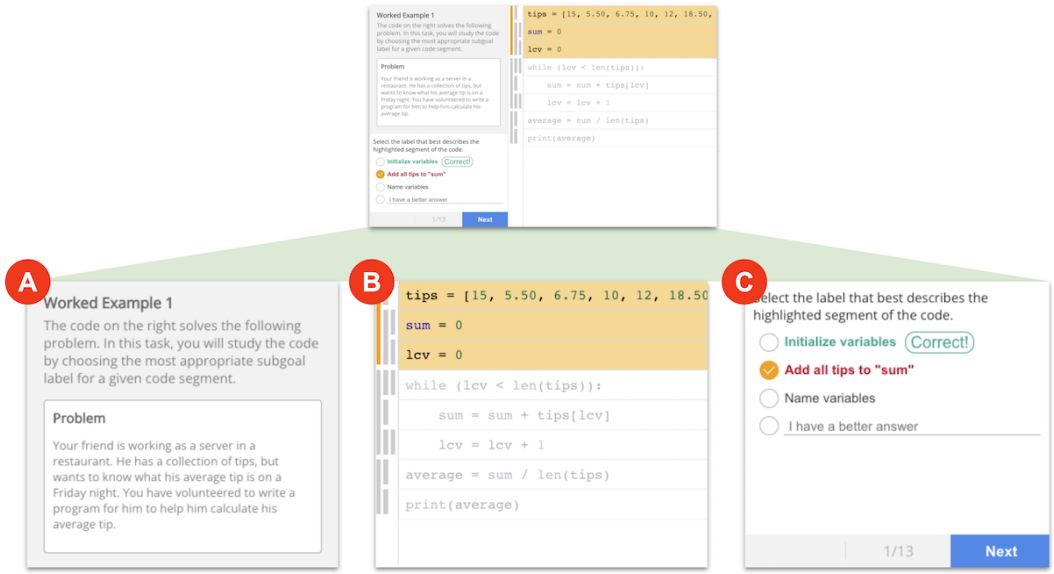


Fig. 4. The user interface for the selection task: (A) Instructions and problem statement, (B) A code example to study. Parts of code being asked in the MCQ are highlighted in orange, (C) A MCQ problem and its options. When learners select options and click the "Next" button, our system provides corrective feedback on their selection.

4.2 Colored bar visualization to overview subgoal hierarchies

Our system also provides a novel visualization of subgoal hierarchies through color-coding of code scope and labels to overview goal structures and adapt flexibly to different code formats and deeply nested structures. We adopted the visualization to the user interfaces of both microtasks (Fig. 3 (B) and Fig. 4 (B)). Each colored bar beside code examples represents a code group. The vertical position of a bar indicates the code lines that it groups. For example, the red bar in Fig. 3 represent a code group for line 2 and 3. The horizontal position of a bar indicates the level it belongs to in a hierarchy. The red bar is at the second level under the green bar that groups line 2 to 8. The color-coding of bars maps each subgoal to a specific part of code examples and reduces the split-attention effect by serving as a visual link between the spatially distant code and subgoal labels [21, 22].

Colored bars collectively outline the goal structures and help learners overview them (G3). In *Generation* task, colored bars support learners in coordinating overall goal structure by informing learners at which position and level they create subgoals. Learners can also check the bars to easily spot which part of code examples they have not annotated with subgoals. In *Selection* task, colored bars serve as a navigator to traverse goal structures deeply nested with many constituent subgoals. Learners explore the goal structure by preorder traversal and receive MCQs that ask about increasingly specific code groups. Learners can also refer to parent or child code groups to select the right level of subgoal explanations in MCQs.

The visualization is designed to present complex mapping between subgoal hierarchies and code. Code examples that are subgoal-labeled at their creation are written in a way that clearly shows goal structures from code. Inline comments or visual isolation can easily present subgoal hierarchies of such code examples. In order to make our visualization applicable to more general code examples beyond the pre-formatted code examples, we made the visualization flexible to work for possibly complex structures and mappings. For instance, lines 4 and 6 in Fig. 3 may form a meaningful code group despite not being contiguous. While the comment-based presentation does not work for these non-contiguous code groups, our visualization can pinpoint code lines and group them.

Algorithm 1 A hierarchy generation algorithm

Input: I: A list of Tuple(code group, subgoal label)
Step 1: merge tuples with identical code groups
$U \leftarrow$ an empty list
for each Tuple(code group G, subgoal label L) in I **do**
  if U has a tuple containing G **do**
    add L to the existing tuple
  else do
    add Tuple(G, L) to U
  end if

Step 2: calculate priority of each code group
for each code group G in U **do**
  $Priority_G \leftarrow$ occurrence number of G in I

Step 3: sort code groups by their priority
sort U by $Priority$

Step 4: populate as many code groups in a hierarchy
$H \leftarrow$ an empty hierarchy
for each Tuple(code group G, subgoal labels L) in U **do**
  if G does not conflict with H **do**
    add G and L to H
  end if
Output: H

4.3 Workflow and algorithm for generating multi-leveled hierarchies

Our learnersourcing workflow is organized by the two microtasks and a hierarchy generation algorithm (Fig. 1). Learner-generated subgoals from *Generation* are the seed for our algorithm to

generate initial subgoal hierarchies. The generated hierarchies are fed to *Selection* to refine subgoal labels.

The hierarchy generation algorithm (Algorithm 1) is based on two assumptions:

A1. most learners can identify correct individual subgoals although they may lack the ability to identify an entire hierarchy.
A2. learners can recognize complementary levels of subgoals so that their collection will be comprehensive enough to make complete subgoal hierarchies.

Based on these assumptions, the algorithm first calculates the priority of each code group by their submission count. Then, the algorithm uses the priority values to decide which subgoal to add to the subgoal hierarchy in case of conflicts (see Fig. 7). The algorithm keeps adding subgoals without conflicts until it achieves complete hierarchies.

Before being fed to *Selection* task, generated subgoal hierarchies undergo post-processing. CodeTree removes the poor subgoal labels that are too lengthy, that start with non-verbs, or that are textually similar to other subgoal labels. For measuring textual similarity, we divided texts into morphemes [30] and computed the Sørensen–Dice coefficient [55] between the morphemes. To make subgoal hierarchies complete, CodeTree also adds code groups to the leaf positions where subgoals are missing from the input. In *Selection* task, learners generate the labels for these new code groups.

5 STUDY DESIGN

To evaluate the feasibility of CodeTree for the three design goals, we recruited 45 programming novices to run a between-subjects study with three conditions—**Baseline**, **Generate**, and **Select**. The conditions differed in the methods of studying code examples. *Generate* condition participants studied code examples by doing *Generation* task; *Select* condition participants studied code examples through *Selection* task; *Baseline* condition participants studied code examples with detailed explanations only. Through this study, we explored three research questions that link to each design goal.

RQ1. Can CodeTree learnersource correct and comprehensive subgoal hierarchies?
RQ2. Do learners find *Generation* and *Selection* tasks helpful for learning and manageable to do?
RQ3. Does the colored bar visualization help understanding and generation of subgoal hierarchies?

5.1 Participants

The target users of CodeTree are programming learners who have learned basic Python syntax but struggle to write code themselves to solve problems. We recruited 45 participants who 1) took an introductory Python programming class only, 2) did not score perfectly in our pre-test, and 3) experienced moderate intrinsic cognitive (below 19 out of 30) during the study. The participants were recruited on campus and from online communities with a compensation of 20,000 KRW (approximately 17 US dollars) for a 90-minute session. The participants were randomly assigned to one of the three conditions (*Baseline*, *Generate*, and *Select*). We confirmed that there was no statistically significant difference in participants' initial knowledge (pre-test scores) between conditions (one-way ANOVA, F=0.89, and p=0.42).

5.2 Procedure and Materials

Throughout the sessions, participants learned the usage of while-loops in Python through three code examples and practice problems isomorphic to the examples. We referred to the study procedures and materials from previous studies on subgoal learning [36, 45]. The code examples, the problems

Table 1. Demographic averages for 45 participants and the correlation of each characteristic with participants' performance score.

	Mean/ proportion	*Std. deviation*	*Pearson's correlation with performance score*	
			r	*p*
Gender	19 female	-	0.14	0.30
Age	21.73	3.84	-0.29	0.15
Year in college	2.82	3.84	0.31	0.13
Comfort with programming (1:Not comfortable at all - 7: Very comfortable)	3.69	1.13	0.01	0.48
Expected difficulty for learning programming (1: Very difficult - 7: Very easy)	3.44	1.02	0.33	0.11

Table 2. The outline of the study and the time allotted to each step.

Step (min.)	**Baseline (15 participants)**	**Generate (15 participants)**	**Select (15 participants)**
1 (8)	Introduction + Informed consent		
2 (3)	Demographic questionnaire		
3 (5)	Pre test		
4 (10)	Analogy training	Subgoal training	
5 (30)	Study 3 code examples with explanations + Solve 3 practice problems	Generation tasks on 3 code examples + Solve 3 practice problems	Selection tasks on 3 code examples + Solve 3 practice problems
6 (3)	Cognitive load measurement		
7 (20)	4 Assessment problems		
8 (5)	1 Parsons problem		
9 (5)	Post test (identical to pre test)		
10 (-)	Post survey		

for pre & post-tests, practice, and assessment were identical to the materials used in the between-subjects study of Margulieux et al. [36], except that the materials were rewritten in Python (see Appendix A.). We chose Python for our programming language because Python was the most popular among our recruit targets. All participants were accessible to review materials that briefly explained basic syntax and concepts of Python. The instructions and the user interface were localized into Korean to avoid confusion or unnecessary difficulties.

The example learning steps (steps 4 and 5 in Table 2) were different by conditions. Participants in *Generate* and *Select* conditions received tutorials about subgoal learning and usage of each feature of our user interface. The tutorial included the learning benefits of subgoal learning, exemplar subgoal labels on simple math equation solving, and subgoal-making exercises with answer labels at the end as corrective feedback (Fig. 5 Right). The example subgoal labels were all problem-independent, implicitly guiding participants to write problem-independent subgoal labels. *Baseline* participants received analogy training, which exerts cognitive load comparable to the subgoal training in other conditions [45]. *Generate* and *Select* participants used respective interfaces in Fig. 3 and Fig. 4 to study code examples. *Baseline* participants used another interface that removed subgoal-related features (Fig. 5 Left).

The code examples in all three conditions were presented with line-level explanations of the code. We added the explanations to help participants understand the code and to simulate typical Q&A websites and documentation where explanations of code are present. We used the latest Codex AI model [12] to generate the explanations in consideration that it is the most readily available method for providing detailed explanations of code at scale [33]. The first author checked the quality of the explanations. The code examples used in the study and the line-level explanations are provided in Appendix A.4.

The transfer distances of the practice problems and assessment problems to code examples were set differently. Right after studying each code example, participants solved a practice problem, which solution was isomorphic [45] to the code example. Participants could run their code and receive feedback on whether they were correct. We chose isomorphic problems so that participants could try out the same code structure they had just studied. For the assessment problems, we chose contextual transfer [45] problems to test how each condition affects participants' performance in modifying learned code examples and transferring them to novel problems. Participants could not run their code during the assessment nor receive feedback.

We conducted all sessions with *Generate* participants before any *Select* participants to populate subgoal hierarchies for *Selection* task. The hierarchy algorithm generated subgoal hierarchies with the *Generate* participants' code groups.

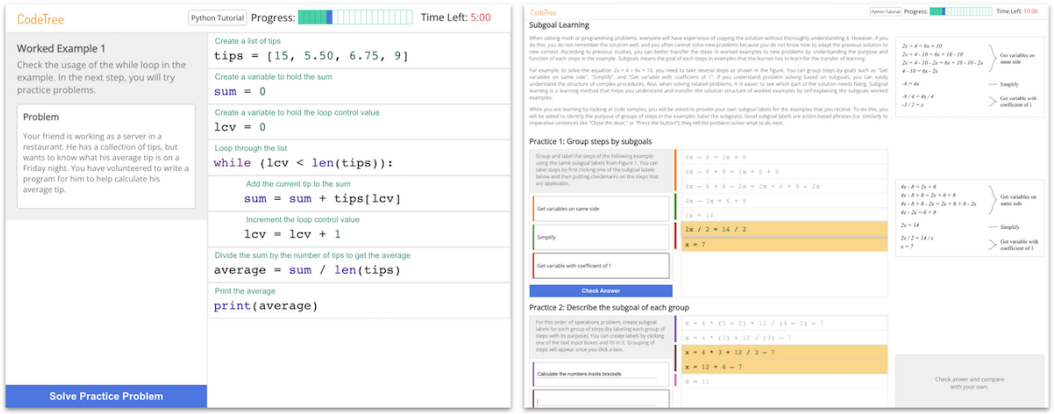


Fig. 5. Left: the user interface for *Baseline* condition. Participants studied the worked example with line-level explanations of the code only, and they could proceed to the next step at any time without requisites. Right: the instruction and practice activities for the subgoal training. Participants grouped and subgoal-labeled math equation-solving steps as practice and then checked the answer.

5.3 Measurements

Our measurements are two-fold. The evaluation of the quality and variety of the subgoals created by participants was taken after the study with external evaluators. The assessment of participants' learning gain and experience was conducted sequentially during the study (Table 2).

Code group quality. The quality of code groups made by *Generate* participants was classified into three types—*incorrect*, *meaningful*, and *core* (see Table. 3). A code group is *meaningful* if its constituent code lines collectively represent any useful subgoal. A code group is *core* if it is *meaningful* and represents one of the subgoals essential to solving a problem. Code groups that are not *meaningful* nor *core* are considered *incorrect*. We recruited two evaluators with four semesters

of TA experience in CS courses. The evaluators assessed the first 30 code groups together and the remaining 39 code groups independently. The inter-rater reliability for the independent assessment was substantial (Cohen's kappa, κ=0.63).

Table 3. The code groups that the evaluators assessed as *incorrect*, *meaningful*, and *core* respectively. The entire code example is in Fig. 2.

Code group quality	Example
Incorrect	L1: tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9] L2: sum = 0 L5: sum = sum + tips[lcv]
Meaningful	L6: lcv = lcv + 1
	L4: while (lcv <len(tips)):
Core	L4: while (lcv <len(tips)): L6: lcv = lcv + 1
	L7: average = sum / len(tips) L8: print(average)

Subgoal label quality. The quality of subgoal labels made by *Generate* participants was classified into three types—*incorrect*, *problem-specific*, and *problem-independent* [36] (see Table. 4). A label is *incorrect* if it simply describes the execution of code or is wrong. A label is *problem-specific* if it correctly describes the function of code but contains information specific to the current problem and cannot be generalized to other isomorphic problems. A label is *problem-independent* if it is correct and generalizable to other isomorphic problems. We recruited two other evaluators with two semesters of TA experience in CS courses. Likewise, they assessed the labels from the first 30 code groups together and the remaining 72 labels independently. The inter-rater reliability for the independent assessment was high (Cohen's kappa, κ=0.68).

Table 4. Subgoal labels that evaluators assessed as *incorrect*, *problem-specific*, and *problem-independent*. The labels described the subgoal for code "sum = sum + tips[lcv]" in Fig. 2.

Subgoal label quality	Example
Incorrect	Add tuple[lcv] to sum
Problem-specific	Add a tip value to total sum
Problem-independent	Add a value to get the total sum

Diversity index and conflict ratio of code groups. One of our assumptions (A2) in algorithm design is that learners will generate diverse code groups that can complement each other. Hence, diversity and complementarity in code groups are important properties that make our learnersourcing workflow effective. We used Simpson's diversity index [51] to measure the diversity of code groups. For the calculation of the index, we treated each code group as an entity and the number of its submissions by participants as its population. We measured the complementarity of code groups by the ratio of conflicting code group pairs in all possible pairs (1.0 is a total conflict). Two code groups are complementary if one is the subset of the other or there are no intersecting code lines; otherwise, they conflict and cannot coexist in the same hierarchy (see Fig. 7). However, there will be existential conflicts between code groups because there can be multiple correct instances of subgoal hierarchies. We measured the existential conflict ratio by the same calculation but with only core code groups.

Holistic evaluation on subgoal hierarchies. Holistic evaluation is meaningful apart from the previous measurements. A hierarchy may not be effective for learning, even though its constituent code groups and labels are correct individually. For example, a subgoal hierarchy may have an imbalanced goal structure or inconsistent subgoal labels for denoting parts of code. The holistic evaluation aims to evaluate code groups and labels as a whole beyond their individual qualities. The assessment focused on 1) the composition of code groups in each layer in hierarchies and 2) the consistency among subgoal labels (see Fig. 6). We assessed the composition by evaluating whether each parent code group is split in a logically even manner by its child code groups. We quantified consistency by the size of the largest set of labels that do not conflict with each other in denoting variables or concepts. The evaluators who assessed the code groups worked together to evaluate the composition and consistency of the three subgoal hierarchies generated for each code example.

Score increase. Participants' code-tracing skills were measured with the multiple-choice questions in pre and post-tests (Step 3 and 9 in Table 2). Pre and post-tests were composed of questions that asked about execution outputs of while loops or the code to print desired outcomes. The questions for both tests were identical, but the order of questions and options was randomized. The score differences between pre and post-tests were calculated for each participant to measure individuals' learning gain in code-tracing skills.

Performance score. Participants' code-writing skills were measured with four assessment problems and a Parsons problem (Step 7 in Table 2). Two external evaluators graded the participants' answers to the assessment problems. The evaluators had two and four semesters of TA experience in an introductory Python programming class. The evaluators followed the grading guidelines from the previous work [36], and the total score for the four problems was 36. For each assessment problem, the two evaluators graded the first 20 participants' answers together to make a specific grading scheme. They graded the remaining 25 answers independently. The inter-rater reliability of the independent grading was high (Pearson's correlation, r=0.94). Participants' answers to the Parsons problem were auto-graded and scored out of 10. Each participant's performance score is the sum of the assessment score and the Parsons score.

Cognitive load. The cognitive load of participants was measured right after the example learning steps to evaluate how each intervention imposes a burden on learners (Step 6 in Table 2). We used ten 10-point Likert-scale questions designed to measure cognitive load for programming tasks [43]. The questions asked about three types of cognitive load—intrinsic, extrinsic, and germane. Participants' ratings for each question were summed up by the types, resulting in a maximum of 30 for intrinsic and extrinsic, and 40 for germane.

Post-survey questions. After the post-test, the participants received survey questions on their learning experience and system usability (Step 10 in Table 2). The survey is composed of two parts. The first part (Fig. 10) had questions about their experience studying code examples and were common to all conditions. The second part (Fig. 11) had questions about the system's usability and helpfulness for conducting condition-specific tasks and were different by condition. Each question had two sub-questions in which participants rate a 7-point Likert scale for a given statement and leave text comments to explain their rating.

6 RESULTS

We report the quantitative results and participants' comments on the evaluation study and answer the three research questions. We organized this section by each research question.

RQ1. Can CodeTree learnersource correct and comprehensive subgoal hierarchies?

Learnersourced subgoal hierarchies were correct and complete. The ratios of correct (*meaningful* + *core*) code groups in total code groups in respective subgoal hierarchies were 13/13, 13/14, and 20/21. Moreover, the subgoal hierarchies contained all the *core* code groups identified by the evaluators. Noting that the average ratios of *core* code groups in each participant's submission were 65%, this result shows that the participants could collectively populate most of the core code groups, although a participant alone could not. In terms of label quality, the ratios of correct (*problem-specific* + *problem-independent*) subgoal labels in total labels were 12/13, 11/14, and 18/21 in each subgoal hierarchy. Among them, 8, 8, and 10 subgoal labels were *problem-independent*. In the holistic evaluation, most compositions of the code groups were correct, and the consistencies between subgoal labels were also high. The ratios of correct parent-child code group relations were 6/6, 5/7, and 8/10. The sizes of the largest consistent subgoal label set were 11/13, 12/14, and 21/21. Hence, we conclude that each generated subgoal hierarchy is correct and consistent as a whole. Detailed evaluation results of subgoal hierarchies for the first and second code examples are presented in Fig. 6.

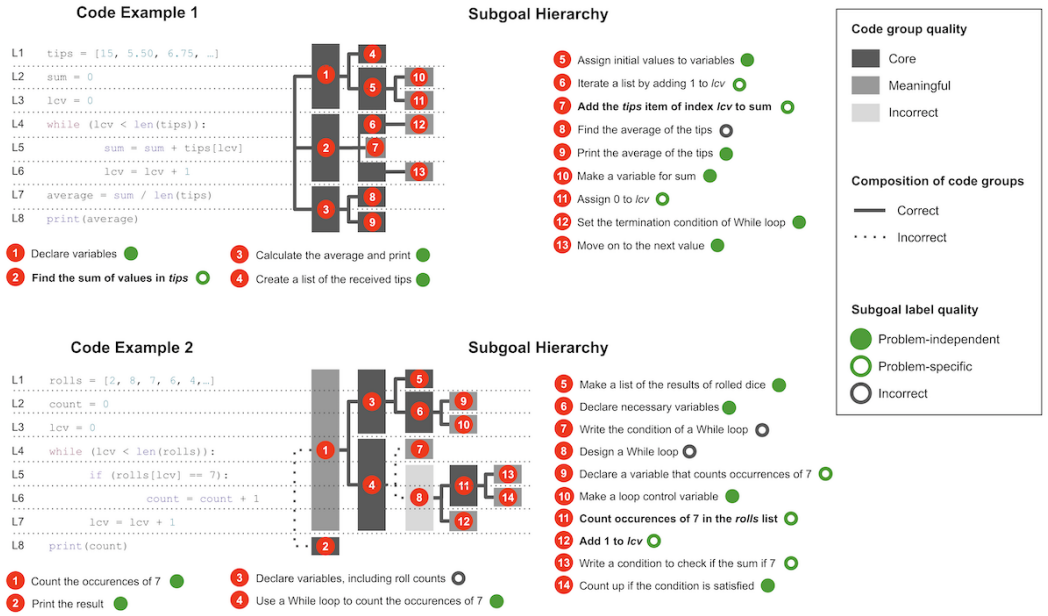


Fig. 6. The subgoal hierarchy generated for code examples 1 and 2 and their evaluation results. We presented the most selected subgoal labels only. The participants wrote subgoal labels in Korean to avoid language barriers in describing good labels, and we translated them into the figure. The labels that are inconsistent with others are bolded. The evaluators judged these labels to be inconsistent in that they use exact variable names (e.g., rolls and lcv) for reference while others explain in words.

The participants generated meaningful code groups and labels. Almost all of the code groups that *Generate* participants made were correct (*meaningful* + *core*). On average, 95% of the code groups from a participant were correct (SD = 6%). On the other hand, the ratios for *core* code groups were only 65% (SD = 17%). For MCQ responses, the participants chose better or equal quality labels most of the time (M = 98%, SD = 3%). These observations collectively verify our assumption A1 to a certain extent that the majority of individual learners can identify correct subgoals but struggle to generate complete hierarchies with all *core* code groups.

The participants generated diverse but somewhat conflicting code groups. The Simpson's diversity indexes of code groups for the three code examples were 0.93, 0.91, and 0.96. Their conflict ratios were 0.54, 0.54, and 0.55, and their existential conflict ratios were 0.53, 0.57, and 0.57. High diversity indexes and moderate conflict ratios indicate that participants recognized diverse subgoals but from different instances of subgoal hierarchies. Although this does not align with our assumption A2 that learners will make complementary subgoals, the quality evaluation showed that the algorithm filtered conflicting code groups and generated hierarchies correctly. We expect that there is room for guiding learners to generate subgoals of specific instances to reduce conflicts and make our workflow more efficient.

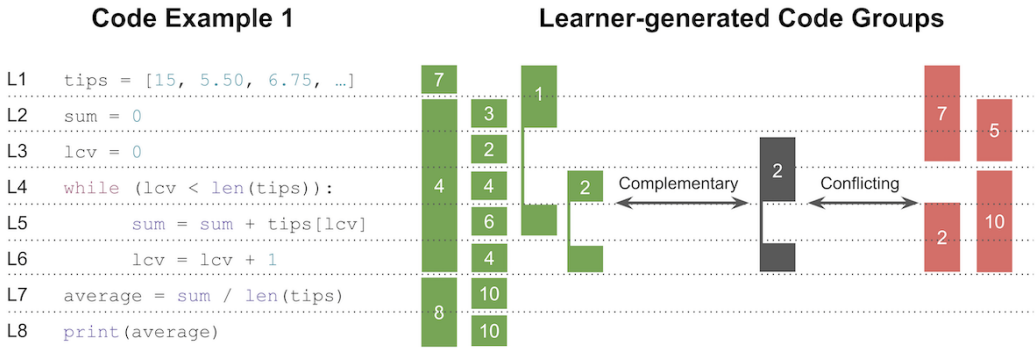


Fig. 7. The code groups that the *Generate* participants generated for code example 1. The number in a code group indicates the number of participants who submitted it. To measure conflicts between code groups, we counted the ratio of conflict relations (colored in red) in all pairs of code groups.

Just a few learner contributions could help generate high-quality subgoal hierarchies. To estimate the number of learners needed for achieving high-quality subgoal hierarchies, we measured the quality of code groups and subgoal labels under a simulation. We simulated 1) the hierarchy generation algorithm and 2) the *Selection* task, each with n number of simulated learners.

To simulate the algorithm with n participants, we randomly sampled n *Generate* participants. We then used the algorithm to generate subgoal hierarchies from the sampled participants' code groups. The random sampling was repeated 15 times for each n. We report their average. We calculated the ratio of correct (*meaningful* + *core*) code groups and the number of *core* code groups in the generated hierarchies and plotted them against the number of simulated learners in Fig. 8 (A) and (B). The ratios of *correct* code groups at the end (n=15) were 0.93, 0.87, and 0.95 for the subgoal hierarchies of each code example. The numbers of *core* code groups at the end were 6.33, 5.00, and 10.00. Both estimates saturate around n=5, albeit the third subgoal hierarchy continued to improve and populated all the *core* code groups at n=15.

To observe how the quality of subgoal labels changes throughout the *Selection* task, we simulated n successive learner's responses in the *Selection* task with a probability that a participant will choose a better subgoal label in an MCQ. We set the probability as 0.98 based on our empirical result. A simulated learner votes for one of the best subgoal labels in given MCQ options or other options according to the probability. For each n, we repeated the simulation 15 times. We report the average ratio of correct (*problem-specific* + *problem-independent*) and *problem-independent* labels among the labels that received the most votes. Fig. 8 (C) and (D) plot these against the number of simulated learners. All hierarchies achieved high correctness at the end (1.00, 0.93, and 0.90) and reached the maximum achievable with the given data. All hierarchies also had high populations of

problem-independent labels (0.82, 0.77, and 0.66) close to their achievable maximums (0.92, 0.79, and 0.71). Both estimates saturated around n=5.

The result implies that just five learners can successfully populate a goal structure and subgoal labels for a code example. Such learner-to-code example ratio (five to one) shows that the generation of subgoal hierarchies for all code examples in typical Q&A websites and MOOCs is feasible with just existing learners in the learning environments. For instance, a question on StackOverflow is viewed by 30 people on average in a month [2]. If five of them are motivated to understand code examples in depth and contribute subgoals to CodeTree, the system will be able to populate subgoal hierarchies for all newly created and old code examples on StackOverflow. However, we also clearly note that our study participants may have a higher level of prior knowledge in programming than typical learners in the environments, and we may not replicate such high efficiency (i.e., five to one) in the wild where the level of code complexity and knowledge of learners vary a lot. Nevertheless, our finding suggests that learnersourcing is a scalable approach for collecting subgoals online. We discuss how we can concretize the findings in future research in Section 8.

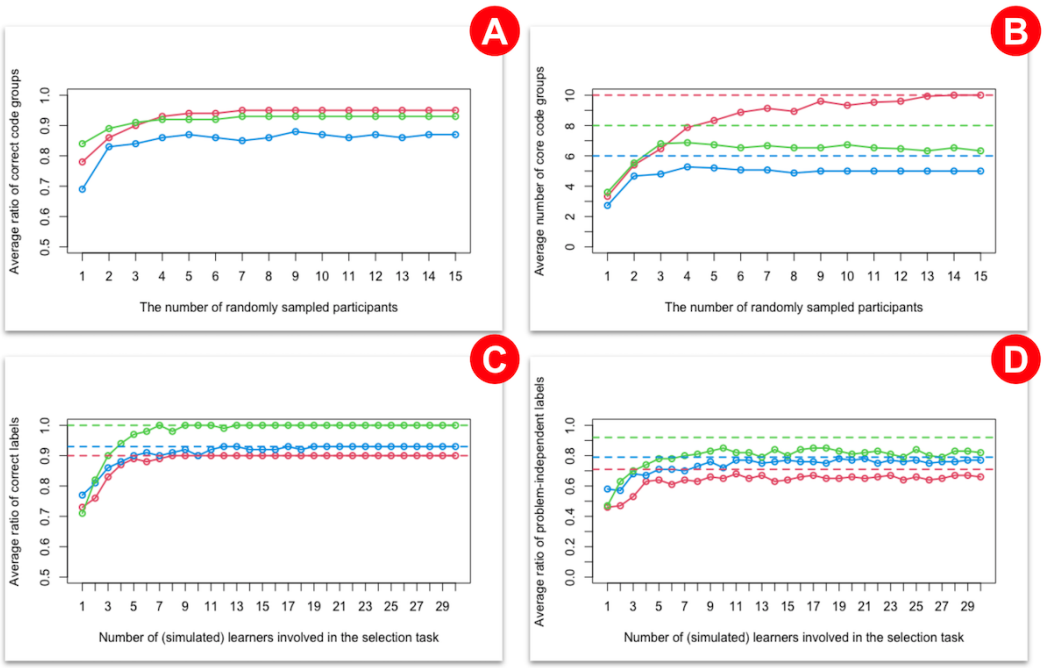


Fig. 8. Each color line denotes subgoal hierarchies for different code examples: green for code example 1, blue for code example 2, and red for code example 3. (A, B) The average ratios of (correct/core) code groups in hierarchies that were generated with the code groups of randomly sampled n participants. The dashed line denotes the total number of core code groups. (C, D) The average ratio of *correct*/*problem-independent* labels at the end of the selection task simulation with n participants. The dashed lines denote the maximum achievable ratios for given datasets. For example, if the subgoal labels for a code group are all *incorrect*, there is no way to improve.

RQ2. Do learners find *Generation* and *Selection* tasks helpful for learning and manageable to do?

[2]https://data.stackexchange.com/stackoverflow/query/213319/average-views-per-question-by-month

The generation task improved code tracing skills. All except one participant scored equal or higher in their post-test. The score increases (Fig. 9 (A)) in *Generate* condition were statistically significantly higher than in *Baseline* condition (one-tailed t-test, $p = 0.03$, $d = 0.72$). The score increases in *Select* condition were higher than in *Baseline* but not statistically significant (one-tailed t-test, $p = 0.15$, $d = 0.38$). The performance scores (see Fig. 9 (B)) in both *Generate* and *Select* conditions were higher than in *Baseline* condition but not statistically significant (one-tailed t-test, $p = 0.15$, $d = 0.38$ between *Baseline* and *Generate*, $p = 0.07$, $d = 0.40$ between *Baseline* and *Select*).

Our results and previous studies on subgoal learning [36, 45] complement each other to a certain extent. The score increases and performance scores measure participants' code tracing and writing skills respectively. Previous studies observed significant improvement in code-writing skills for the active and constructive forms of subgoal learning but weak significance in code-tracing skills. On the other hand, we observed significant improvements in participants' code-tracing skills only. In theory, both code-tracing and writing skills should have improved because code-tracing is a precursor to code writing [32]. We speculate that the difference in the tools for subgoal generation might have elicited different aspects of learning more prominent. Nevertheless, they accord closely with the positive effect of subgoal learning in improving transfer distance.

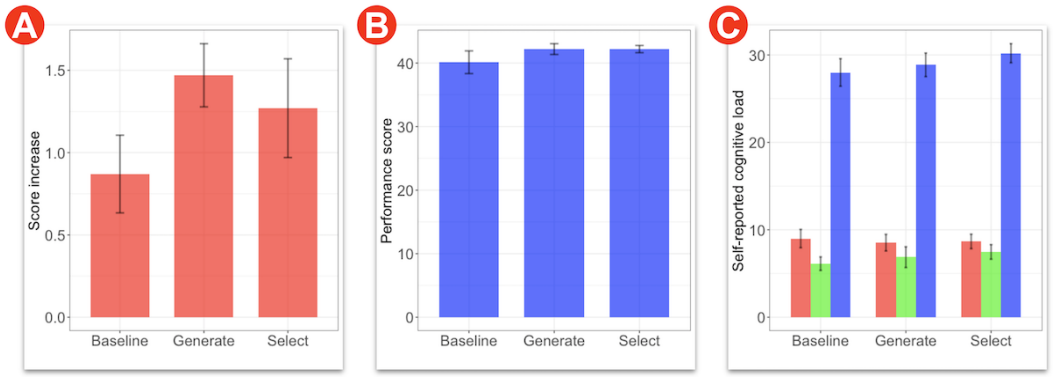


Fig. 9. (A) Average score increases between pre and post-test scores across conditions, (B) Average performance scores across conditions, (C) Average self-reported ratings of intrinsic (red), extrinsic (green), germane (blue) cognitive load across conditions.

Generation and Selection tasks were manageable microtasks. There were no significant differences between conditions for the three types of cognitive load (one-way ANOVA, intrinsic: $F = 0.07$, $p = 0.94$, extrinsic: $F = 0.68$, $p = 0.61$, germane: $F = 0.50$, $p = 0.52$). Little differences among the conditions indicate that our microtasks did not impose additional cognitive load despite being more active than *Baseline*. P26 commented that subgoal learning tasks were manageable because top-down exploration of subgoals helped him digest the code easily: "I could understand long code examples faster by dividing them into smaller units."

Generation and Selection tasks helped improve learners' satisfaction. The learning condition did not affect participants' *Q1: understanding the usage of while loops* and *Q2: comprehension of code examples* (one-way ANOVA, $F = 0.65$, $p = 0.52$ for *Q1*, $F = 2.56$, $p = 0.09$ for *Q2*). However, the *Generate* and *Select* participants perceived that they *Q3: understood the hierarchical structure of code examples* significantly better than the *Baseline* participants (one-tailed t-test, $p = 0.02$, $d = 0.79$ between *Baseline* and *Generate*, $p = 0.04$, $d = 0.69$ between *Baseline* and *Select*). Subgoal learning tasks were preferred for *Q4: future reuse* over the *Baseline* task, but not statistically significantly

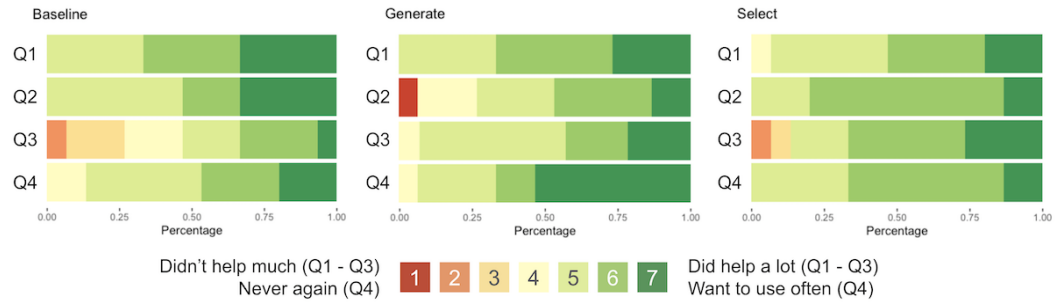


Fig. 10. Likert scale responses for the post-survey questions regarding learning experience. Note that words in parenthesis were changed depending on the answerers' conditions. The survey questions were: *Q1.* How much did (seeing code and explanations/subgoal learning tasks) help to understand the usage of while loops? / *Q2.* How much did (seeing code and explanations/subgoal learning tasks) help to understand code examples? / *Q3.* How much did (seeing code and explanations/subgoal learning tasks) help to understand hierarchical structures of code examples? / *Q4.* How often do you want to use (code examples/subgoal learning tasks) in future programming learning?

(one-tailed t-test, $p = 0.06$, $d = 0.58$ between *Baseline* and *Generate*, $p = 0.31$, $d = 0.31$ between *Baseline* and *Select*).

The participants in *Generate* condition liked the process of explicitly identifying the functions of each part of the code even without corrective feedback. P24 commented, "subgoal learning would be helpful to learn not only the solution specific to example problems but also the general strategies for solving other problems." Another participant P30 said, "[*Generation* task] greatly helped to organize code into small pieces. I used to think programming was difficult, but I could gain confidence by doing the task and solving problems." We speculate that the statistical insignificance resulted from the relatively high satisfaction of *Baseline* participants. *Baseline* participants liked the simplicity of the interface and felt more familiar to use. P3 in the *Baseline* condition noted "the UI was so simple that I could inspect and understand code example well." We may observe a more significant preference between conditions if the study is designed to be within-subjects or more longitudinal to reduce the effect of their initial burden to familiarize novel interfaces.

The participants also perceived the corrective feedback in the *Selection* task as correct and helpful for a check (see Fig. 11 *Q4* and *Q5*). More than half of the *Select* participants rated over 5 for *Q4* and *Q5*. P40 said that the feedback was useful to confirm his understanding of code examples: "I was unsure of my answers many times, but the correct signs helped me confirm that I was doing right." P33 commented the immediacy of feedback also helped: "[the corrective feedback] trained me to self-explain the code with more general and purposeful terms, rather than simply explaining the execution of the code." P44 pointed out that the feedback would be more effective with supplementary explanations for answers. Although most participants thought the answers given by CodeTree were reasonable, some participants doubted the accuracy of the feedback, especially when their responses all turned out to be correct.

RQ3. Does the colored bar visualization help understanding and generation of subgoal hierarchies?

The bar visualization helped generation of subgoals. The *Generate* participants were asked two questions regarding the hierarchy generation interface and the colored bar visualization (see Fig. 11, *Q1* and *Q2*). Both questions were rated over 5 by more than half of the participants. P25 thought the hierarchy generation interface was intuitive, easy to map between code and subgoals, and

effective for representing complex goal structures: "the interface was easy to use and grasp [what I was creating] because subgoals were indented like code in a familiar style." P24 commented, "the interface helped to understand code systematically. I first defined strategies [(i.e., subgoals)] of problems, and then I selected corresponding parts in the actual code." P20, P25, and P29 understood visual notations correctly and liked the idea of using bar lengths and positions to present complex goal structures. P20 noted "the lower-level bars were drawn to the right of, and only within, the upper-level ones. It was intuitive and pleasant to look at with colors."

The bar visualization in Selection task is unnoticeable but is helpful once noticed. The *Select* participants had mixed perceptions of the helpfulness of the bar visualization (see Fig. 11, *Q3*). Half of the participants commented that they did not notice the visualization because its visual changes were too subtle. P31 said "I did not notice the visualization until answering this question. However, if I had looked at it carefully, it would have been helpful to understand the overall hierarchical structures." For those who noticed the visualization, it helped to overview entire hierarchies and check the hierarchical relations between the code asked by MCQs. P41 commented, "I especially liked the area highlighted in orange and the bar on the left showing the structure and scope. I think it helped a lot to understand the overall structure visually. It was also good to show larger scopes of goals first and move to more detailed goals. It would have been better if the bars also showed chosen subgoal labels."

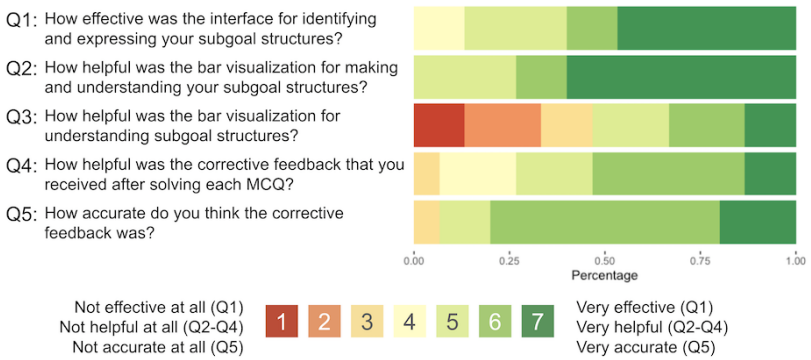


Fig. 11. Likert scale responses for the post-survey questions regarding user interface and the colored bar visualization, and system-generated feedback. *Q1-Q3* were asked to the *Generate* participants, and *Q4-Q5* were asked to the *Select* participants. Images of system features were attached to each question in the survey to indicate each feature clearly.

7 DISCUSSION

In the following subsections, we discuss the implication of our results for generating subgoal hierarchies at scale and generalizing our system design to other relevant domains.

7.1 Efficiency of our workflow for generating hierarchies

One of our notable findings is that just a few learners are needed to generate high-quality goal structures. CodeTree generated correct goal structures with just five programming novices. The participants in the study spent about 9 minutes completing each *Generation* and *Selection* task (6 minutes for *Baseline* task). Considering that *Generate* tasks can run in parallel as they are independent of each other, CodeTree can generate a multi-level goal structure of a code example within 10 minutes in an ideal case. Previous crowdsourcing systems [14, 57] required hundreds

of human intelligence tasks (HITs) for generating taxonomy hierarchies of size 10-20 nodes, a size comparable to our goal structures. Compared with these hierarchy generation methods, our learnersourcing workflow can be deployed in even small classrooms.

We argue that learnersourcing may shed light on reducing the number of workers and time needed for crowdsourcing hierarchically structured data (e.g., concept maps and evaluation criteria). Previous systems that used paid crowdworkers (e.g., MTurk) focused on making each task small so that crowdworkers could contribute without knowing a global hierarchy. However, using fragmented microtasks increased the number of total HITs. In our study, we observed that learners tend to have a good understanding of global hierarchies. For instance, more than half of the *Generate* participants submitted multi-leveled subgoals. Because learners have a good sense of global hierarchies, crowdsourcing systems may empower learners to engage more in the global process of hierarchy generation. Learners may be empowered to directly edit and fix global hierarchies to complement algorithmic coordination.

Indeed, we observed the benefit of empowering learners for higher engagement in our system design iterations. Early *Generation* interface constrained learners from making multi-level subgoals (i.e. adding a subgoal below another subgoal). We designed the system this way because we thought allowing the creation of multi-level subgoals would increase task complexity, and learners would not want to exert additional effort on making more subgoals. However, learners in pilot studies commented that making multi-level subgoals would lower the task complexity as they could make goal structures more flexible and closer to what they picture. After redesigning the *Generation* task to accept multi-level subgoals, we could observe higher learner satisfaction and less confusion for making subgoals. The change also improved our system by collecting more subgoals at different levels with fewer learners. These observations align with previous study [53] in that providing learners a choice to engage in tasks rather than forcing them can improve motivation and the quality of collective output. Although this needs a more thorough investigation in real class settings, it will be worth designing future learnersourcing systems with more flexibility and room for higher learner engagement so that learners can contribute more and better if they want to.

7.2 Generalization of the workflow and interfaces

Hierarchical summaries of videos and articles are often effective for navigation, overviewing, and learning the contents [31, 47, 58, 66]. These hierarchical summaries are conceptually similar to subgoal hierarchies, as they divide contents into meaningful units and have labels that summarize each unit. Ideally, content creators can provide hierarchical summaries at their creation time, but there are millions of content already existing on the web without such summaries. The generation of hierarchical summaries at scale can improve the overall web experience. However, generating hierarchical summaries through current data and expert-driven methods share common challenges with subgoal hierarchy generation. Automatic generation requires large datasets specific to each domain to train models, and domain experts who can generate them are scarce compared to the number of videos.

We argue that our learnersourcing approach can be a viable option for generating hierarchical summaries at scale. Our interface and algorithm can work for diverse content types. The core user interface for grouping contents into meaningful units may apply to other content types. For instance, for videos, viewers can interact with a timeline bar to group sections of videos and label them. Then, our algorithm can identify how the timeline needs to be structured. Since videos are a popular medium of online learning (like how-to's and MOOCs), the tasks can also be designed into pedagogical activities [61].

7.3 Comparison to previous studies on subgoal learning

Despite having many similarities with the settings of previous studies [36, 44, 45], our study did not replicate some of their findings. Particularly, our study's three conditions (*Baseline*, *Generate* and *Select*) are comparable to *No subgoal labels*, *Subgoal labels given* and *Subgoal labels generated* conditions in Morrison et al.'s study [44]. We adopted their apparatus for quantitative measurements, instructional materials, and study procedures. However, we observed significant improvement in code-tracing skills in our *Generate* condition and a mediocre change in code-writing, contrary to Morrison et al.'s study.

Our learning interventions had several differences from that of Morrison et al.'s study. First, we asked participants to group code lines by themselves and make labels, while Morrison et al.'s participants had to make subgoal labels only. Second, the units of subgoal were different between the studies. The subgoals in our study grouped code lines; Morrison's subgoals grouped code writing steps (e.g., 1. determine the termination condition of a loop. 2. invert the termination condition into a continuation condition.). These differences in learning interventions might have elicited different skills (code-tracing vs. code-writing). For example, the grouping activity and code-line-based subgoals might make participants stick to code and self-explain code in detail, improving their code-tracing skills. On the other hand, the subgoals that group code writing steps might have helped learners remember the procedure to write code from scratch.

Our study also did not replicate the findings of Margulieux and Catrambone [36] to a certain extent. Margulieux and Catrambone showed that constructive subgoal learning requires either guidance or corrective feedback to elicit a learning effect. In our study, although *Generate* participants did not receive guidance or feedback, they excelled in the post-test. Our result may suggest that 1) constructive subgoal learning is still better than passive learning even without feedback or 2) participants in our study had high prior knowledge and were less dependent on feedback as they can correct themselves within the constructive activity. In future studies, it will be worth looking into how the learner-driven grouping activity affects the learning experience and how different levels of learners' prior knowledge correlate to the necessity of guidance and feedback in subgoal learning.

8 LIMITATIONS AND FUTURE WORK

We discuss the limitations of our work. First, although our controlled study helped observe the learning benefits of using CodeTree, large-scale studies and deployments can provide stronger empirical evidence in our simulated results for both the algorithm and *Selection* task. Second, we should test code examples with more variety in complexity (e.g., code length and depth of loops) and language to see if our workflow and interface work regardless of these variations. For instance, the number of tasks and learners needed for generating high-quality subgoal hierarchies may not linearly scale with code complexity, or code examples with complex structures might overwhelm learners even with our microtasks. Future research may assess the efficiency of the workflow and interface for generating the subgoals of large code bases (e.g., public repositories on GitHub) that span dozens of lines across different files and complex algorithm code (e.g., solution code on LeetCode) that require high-order skills to understand and decompose steps. It will also be interesting to see how far learners' proficiency in programming affects the number of learners needed for generating high-quality subgoals and whether the contribution from poor learners hurts the quality of code groups in our algorithm.

There is also room for improving CodeTree in the future. Generative AI models have the potential to solve cold start problems by offloading the initial burden of learners [18]. We did not look deeply

into the possibility of using Codex for generating goal structures and subgoal labels because learner-driven subgoal generation has pedagogical value on its own, and Codex often gave incorrect outputs in our attempts. Although we used Codex only for generating explanations of code in this work, it is promising to investigate fine-tuned prompts for goal structure generation.

Although this work puts more weight on generating goal structures, *Selection* task also has room for improvement. While the multi-armed bandit algorithm was used to provide good answers, simple random selection was used to generate distractors. More sophisticated methods to generate pedagogically meaningful distractors [2, 59] will improve the learning gain of *Selection* task. It will also be worth exploring different ordering of MCQs. We design CodeTree to traverse goal structures in preorder and ask MCQs, considering the top-down approach is better for reading code examples. Postorder or BFS-like traversal of goal structures may give a better learning experience and preferences for learners.

In addition to refining CodeTree, future research can focus on deploying CodeTree in the wild to benefit learners and instructors in the real world by leveraging its scalability. One good example is to add CodeTree to Q&A websites so that when learners refer to the code examples, they naturally experience subgoal labeling and voluntarily engage in subgoal learning without an instructor's request in the long run. To that end, CodeTree needs to deal with code examples written for more diverse purposes, such as debugging and improving code styles [46], and we need to reimplement CodeTree with a more general purpose and platform-agnostic technology such as Chrome extension.

Besides, it is necessary to inform instructors of the advantages and teaching methods of subgoal learning so that CodeTree can take root in actual classes and disseminate from them. An immediate actionable item is to adopt CodeTree to the introductory Python class in our institution. Since most study participants took the class, we can expect a replication of significant learning effects from the students. We can ask students to use CodeTree for self-explaining the subgoals of code in their lab sessions, in which students solve practice problems after the lecture. The class deployment will allow us to examine the long-term dynamics of subgoal learning [38] and help students utilize subgoal learning in daily practices with metacognition. Instructors of the class can also use the student-generated subgoal labels as a source to check and evaluate students' understanding.

Despite the difficulty in deploying education systems to real-world learning environments, we believe there is room for spreading subgoal learning with our system as the awareness and trials of new technology adoption increase.

REFERENCES

[1] Solmaz Abdi, Hassan Khosravi, Shazia Sadiq, and Gianluca Demartini. 2021. Evaluating the quality of learning resources: A learnersourcing approach. *IEEE Transactions on Learning Technologies* 14, 1 (2021), 81–92.

[2] Maha Al-Yahya. 2014. Ontology-based multiple choice question generation. *The Scientific World Journal* 2014 (2014).

[3] Ohad Barzilay, Christoph Treude, and Alexey Zagalsky. 2013. Facilitating crowd sourced software engineering via stack overflow. In *Finding Source Code on the Web for Remix and Reuse*. Springer, 289–308.

[4] Michael S Bernstein, Greg Little, Robert C Miller, Björn Hartmann, Mark S Ackerman, David R Karger, David Crowell, and Katrina Panovich. 2010. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. 313–322.

[5] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599.

[6] Richard Catrambone. 1994. Improving examples to improve transfer to novel problems. *Memory & cognition* 22, 5 (1994), 606–615.

[7] Richard Catrambone. 1995. Aiding subgoal learning: Effects on transfer. *Journal of educational psychology* 87, 1 (1995), 5.

[8] Richard Catrambone. 1996. Generalizing solution procedures learned from examples. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 22, 4 (1996), 1020.

[9] Richard Catrambone. 1998. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of experimental psychology: General* 127, 4 (1998), 355.
[10] Richard Catrambone. 2011. Task analysis by problem solving (TAPS): Uncovering expert knowledge to develop high-quality instructional materials and training. In *Learning and Technology Symposium, Columbus, GA*.
[11] Richard Catrambone and Keith J Holyoak. 1990. Learning subgoals and methods for solving probability problems. *Memory & Cognition* 18, 6 (1990), 593–603.
[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[13] Michelene TH Chi. 2009. Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in cognitive science* 1, 1 (2009), 73–105.
[14] Lydia B Chilton, Greg Little, Darren Edge, Daniel S Weld, and James A Landay. 2013. Cascade: Crowdsourcing taxonomy creation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1999–2008.
[15] Kabdo Choi, Hyungyu Shin, Meng Xia, and Juho Kim. 2022. AlgoSolve: Supporting Subgoal Learning in Algorithmic Problem-Solving with Learnersourced Microtasks. In *CHI Conference on Human Factors in Computing Systems*. 1–16.
[16] Graham Cooper and John Sweller. 1987. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of educational psychology* 79, 4 (1987), 347.
[17] Adrienne Decker, Lauren E Margulieux, and Briana B Morrison. 2019. Using the SOLO taxonomy to understand subgoal labels effect in CS1. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 209–217.
[18] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources–Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).
[19] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 20–29.
[20] Bat-Sheva Eylon and F Reif. 1984. Effects of knowledge organization on task performance. *Cognition and instruction* 1, 1 (1984), 5–44.
[21] Mareike Florax and Rolf Ploetzner. 2010. What contributes to the split-attention effect? The role of text segmentation, picture labelling, and spatial proximity. *Learning and instruction* 20, 3 (2010), 216–224.
[22] Paul Ginns. 2006. Integrating information: A meta-analysis of the spatial contiguity and temporal contiguity effects. *Learning and instruction* 16, 6 (2006), 511–525.
[23] Elena L Glassman, Aaron Lin, Carrie J Cai, and Robert C Miller. 2016. Learnersourcing personalized hints. In *Proceedings of the 19th ACM conference on computer-supported cooperative work & social computing*. 1626–1636.
[24] Philipp Helfrich, Elias Rieb, Giuseppe Abrami, Andy Lücking, and Alexander Mehler. 2018. TreeAnnotator: versatile visual annotation of hierarchical text relations. In *Proceedings of the eleventh international conference on language resources and evaluation (LREC 2018)*.
[25] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, Vol. 25. Citeseer.
[26] Hyoungwook Jin, Minsuk Chang, and Juho Kim. 2019. SolveDeep: A System for Supporting Subgoal Learning in Online Math Problem Solving. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–6.
[27] Juho Kim et al. 2015. *Learnersourcing: improving learning with collective learner activity*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[28] Juho Kim, Philip J Guo, Carrie J Cai, Shang-Wen Li, Krzysztof Z Gajos, and Robert C Miller. 2014. Data-driven interaction techniques for improving navigation of educational videos. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 563–572.
[29] Juho Kim, Phu Tran Nguyen, Sarah Weir, Philip J Guo, Robert C Miller, and Krzysztof Z Gajos. 2014. Crowdsourcing step-by-step information extraction to enhance existing how-to videos. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 4017–4026.
[30] Joon-Ho Lim, Yongjin Bae, Hyunki Kim, Yunjeong Kim, and Kyu-Chul Lee. 2015. Korean Dependency Guidelines for Dependency Parsing and Exo-Brain Language Analysis Corpus. In *Annual Conference on Human and Language Technology*. Human and Language Technology, 234–239.
[31] Ching Liu, Juho Kim, and Hao-Chuan Wang. 2018. ConceptScape: Collaborative concept mapping for video learning. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–12.
[32] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.
[33] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International*

Computing Education Research-Volume 2. 37–39.

[34] Lauren Margulieux and Richard Catrambone. 2017. Using learners' self-explanations of subgoals to guide initial problem solving in app inventor. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 21–29.

[35] Lauren E Margulieux and Richard Catrambone. 2016. Improving problem solving with subgoal labels in expository text and worked examples. *Learning and Instruction* 42 (2016), 58–71.

[36] Lauren E Margulieux and Richard Catrambone. 2019. Finding the best types of guidance for constructing self-explanations of subgoals in programming. *Journal of the Learning Sciences* 28, 1 (2019), 108–151.

[37] Lauren E Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research*. 71–78.

[38] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2019. Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 548–554.

[39] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* 7, 1 (2020), 1–16.

[40] Lauren E Margulieux, Briana B Morrison, Baker Franke, and Harivololona Ramilison. 2020. Effect of Implementing Subgoals in Code. org's Intro to Programming Unit in Computer Science Principles. *ACM Transactions on Computing Education (TOCE)* 20, 4 (2020), 1–24.

[41] Samiha Marwan, Yang Shi, Ian Menezes, Min Chi, Tiffany Barnes, and Thomas W Price. 2021. Just a Few Expert Constraints Can Help: Humanizing Data-Driven Subgoal Detection for Novice Programming. *International Educational Data Mining Society* (2021).

[42] Piotr Mitros. 2015. Learnersourcing of complex assessments. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. 317–320.

[43] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. 131–138.

[44] Briana B Morrison, Lauren E Margulieux, and Adrienne Decker. 2020. The curious case of loops. *Computer Science Education* 30, 2 (2020), 127–154.

[45] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*. 21–29.

[46] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 25–34.

[47] Megha Nawhal, Jacqueline B Lang, Greg Mori, and Parmit K Chilana. 2019. VideoWhiz: Non-Linear Interactive Overviews for Recipe Videos.. In *Graphics Interface*. 15–1.

[48] Lin Ni, Qiming Bao, Xiaoxuan Li, Qianqian Qi, Paul Denny, Jim Warren, Michael Witbrock, and Jiamou Liu. 2022. Deepqr: Neural-based quality ratings for learnersourced multiple-choice questions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 12826–12834.

[49] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*. 195–204.

[50] Alexander Renkl. 1997. Learning from worked-out examples: A study on individual differences. *Cognitive science* 21, 1 (1997), 1–29.

[51] Edward H Simpson. 1949. Measurement of diversity. *nature* 163, 4148 (1949), 688–688.

[52] Anjali Singh, Christopher Brooks, and Shayan Doroudi. 2022. Learnersourcing in Theory and Practice: Synthesizing the Literature and Charting the Future. In *Proceedings of the Ninth ACM Conference on Learning@ Scale*. 234–245.

[53] Anjali Singh, Christopher Brooks, Yiwen Lin, and Warren Li. 2021. What's In It for the Learners? Evidence from a Randomized Field Experiment on Learnersourcing Questions in a MOOC. In *Proceedings of the Eighth ACM Conference on Learning@ Scale*. 221–233.

[54] Norman J Slamecka and Peter Graf. 1978. The generation effect: Delineation of a phenomenon. *Journal of experimental Psychology: Human learning and Memory* 4, 6 (1978), 592.

[55] Thorvald A Sorensen. 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. *Biol. Skar.* 5 (1948), 1–34.

[56] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 2012. BRAT: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*. 102–107.

[57] Yuyin Sun, Adish Singla, Dieter Fox, and Andreas Krause. 2015. Building hierarchies of concepts via crowdsourcing. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
[58] Anh Truong, Peggy Chi, David Salesin, Irfan Essa, and Maneesh Agrawala. 2021. Automatic generation of two-level hierarchical tutorials from instructional makeup videos. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.
[59] Ellampallil Venugopal Vinu et al. 2015. A novel approach to generate MCQs from domain ontology: Considering DL semantics and open-world assumption. *Journal of Web Semantics* 34 (2015), 40–54.
[60] Xu Wang, Srinivasa Teja Talluri, Carolyn Rose, and Kenneth Koedinger. 2019. UpGrade: Sourcing student open-ended solutions to create scalable learning opportunities. In *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale*. 1–10.
[61] Sarah Weir, Juho Kim, Krzysztof Z Gajos, and Robert C Miller. 2015. Learnersourcing subgoal labels for how-to videos. In *Proceedings of the 18th ACM conference on computer supported cooperative work & social computing*. 405–416.
[62] Joseph Jay Williams, Juho Kim, Anna Rafferty, Samuel Maldonado, Krzysztof Z Gajos, Walter S Lasecki, and Neil Heffernan. 2016. Axis: Generating explanations at scale with learnersourcing and machine learning. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. 379–388.
[63] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24, 2 (2019), 637–673.
[64] Iman Yeckehzaare, Tirdad Barghi, and Paul Resnick. 2020. QMaps: Engaging Students in Voluntary Question Generation and Linking. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–14.
[65] Yisong Yue, Josef Broder, Robert Kleinberg, and Thorsten Joachims. 2012. The k-armed dueling bandits problem. *J. Comput. System Sci.* 78, 5 (2012), 1538–1556.
[66] Xingquan Zhu, Ahmed K Elmagarmid, Xiangyang Xue, Lide Wu, and Ann Christine Catlin. 2005. InsightVideo: toward hierarchical video content organization for efficient browsing, summarization and retrieval. *IEEE Transactions on Multimedia* 7, 4 (2005), 648–666.

A STUDY MATERIALS

In the appendix, we provide our study materials to help readers understand the exact setting of our study and replicate our study in the future. Since all materials were localized into Korean in the study, we added the original (localized) text along with English translations.

A.1 Pre & Post test Questions

1/5. What are the first and last numbers output by the code segment? (original: 아래 프로그램 코드에서 처음과 마지막으로 출력되는 값은 무엇인가요?)

```
value = 15
while(value < 28):
    print(value)
    value = value + 1
```

(1) 15, 27
(2) 15, 28
(3) 16, 27
(4) 16, 28
(5) I do not know (original: 잘 모르겠음)

Answer: 15, 27

2/5. You are trying to write a program to print the sum of the values in vals. Which of the 3 segments can be entered in the "# missing code" part? (original: vals에 있는 값들의 합을 출력하기 위한 프로그램 코드를 작성하려 합니다. "# missing code" 부분에 들어갈 수 있는 코드는 3개의 Segment 중 어떤 것인가요?)

```
vals = [2,8,7,6,4,7,9,11,8,6,7,4,3,5,7,11,9,7,4,12]
total = 0
# missing code
print(total)

# Segment I.
pos = 0
while(pos < len(vals)):
    totla = total + vals[pos]

# Segment II.
pos = len(vals)
while(pos > 0):
    total = total + vals[pos]
    pos = pos - 1

# Segment III.
pos = 0
while(pos < len(vals)):
    total = total + vals[pos]
    pos = pos + 1
```

(1) I
(2) II
(3) III
(4) I, III (original: I과 III)
(5) II, III (original: II와 III)
(6) I do not know (original: 잘 모르겠음)

Answer: III

3/5. If the value of n is defined as 4, what is the output value of the program code below? (original: 만약 n의 값이 4로 정의되었다면 아래 프로그램 코드의 출력되는 값은 무엇인가요?)

```
outer = 0
while(outer < n):
    inner = 0
    while(inner <= outer):
        print(str(outer) + " ")
        inner = inner + 1
    outer = outer + 1
```

(1) 0 1 2 3
(2) 0 0 1 0 1 2
(3) 0 1 2 2 3 3 3
(4) 0 1 1 2 2 2 3 3 3 3
(5) 0 0 1 0 1 2 0 1 2 3
(6) I do not know (original: 잘 모르겠음)

Answer: 0 1 1 2 2 2 3 3 3 3

4/5. What is the output of the program code below? (original: 아래 프로그램 코드의 출력 결과는 무엇인가요?)

```
# "%" is the remainder operation.
# a % b returns the remainder
# after divinding a by b
# "%"는 나머지 연산입니다.
# a % b는 a를 b로 나눈 뒤
# 나머지를 반환합니다.

a = 24
b = 30
while (b != 0):
    r = a % b
    a = b
    b = r
print(a)
```

(1) 0
(2) 6
(3) 12
(4) 24
(5) 30
(6) I do not know (original: 잘 모르겠음)

Answer: 6

5/5. Which of the segments will print 1 4 7 10 13 16 19? (original: 아래에서 1 4 7 10 13 16 19를 출력하는 프로그램 코드는 무엇인가요?)

```
# Segment I.
k = 1
while(k < 20):
    if(k % 3 == 1):
        print(str(k) + " ")
    k = k + 3

# Segment II.
k = 1
while(k < 20):
    if(k % 3 == 1):
        print(str(k) + " ")
    k = k + 1

# Segment III.
k = 1
while(k < 20):
    print(str(k) + " ")
    k = k + 3
```

(1) I
(2) II
(3) I, II (original: I과 II)
(4) II, III (original: II와 III)
(5) I, II, III (original: I, II, III 모두)
(6) I do not know (original: 잘 모르겠음)

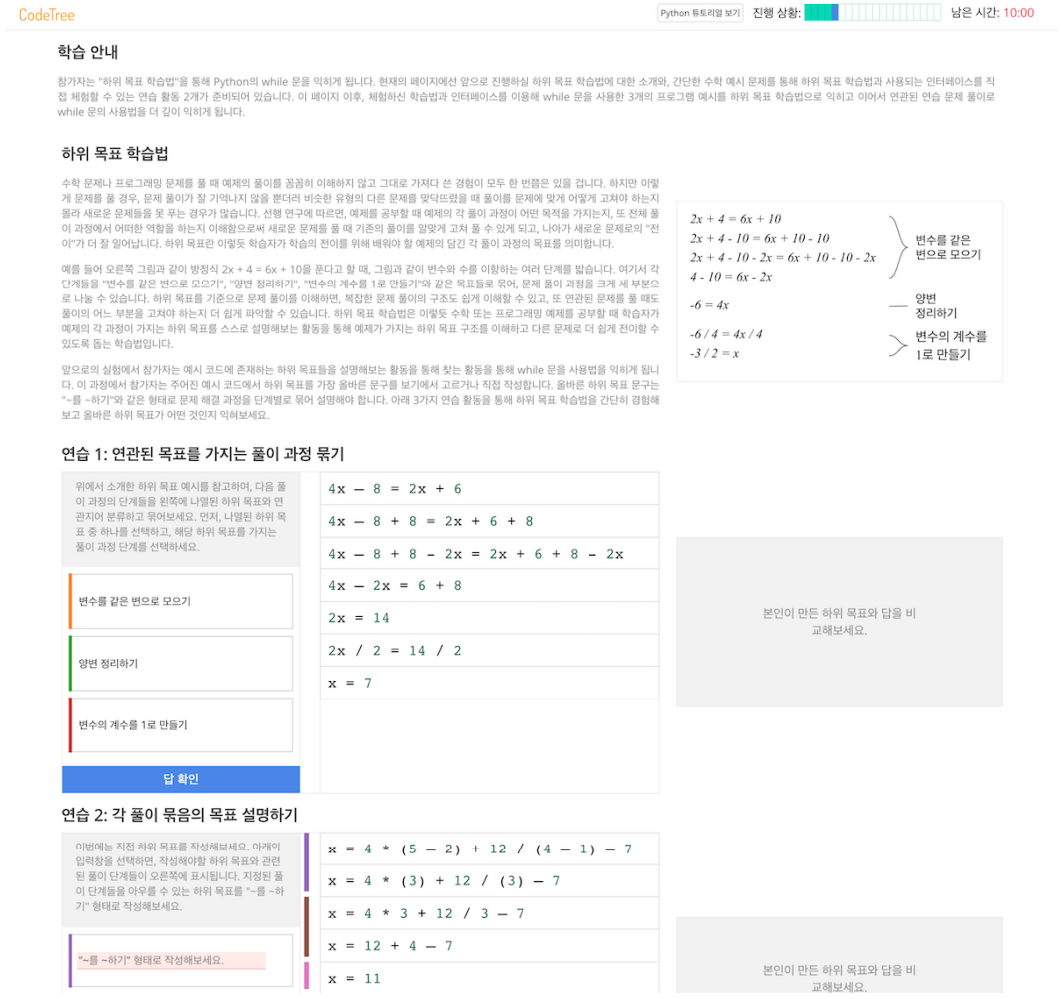
Answer: I, II, III

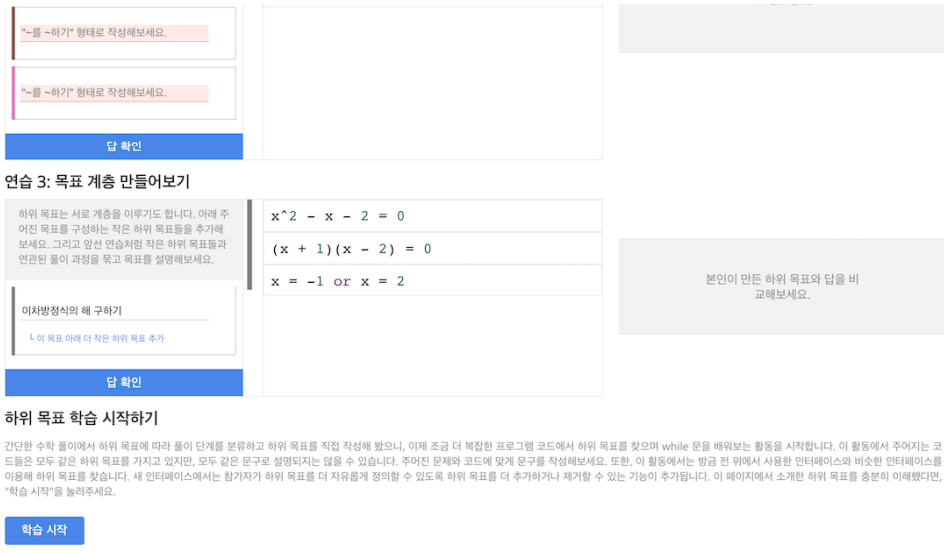
A.2 Materials for Subgoal Training

We show the full-page screenshots of our interface and instructions for the subgoal training steps. We provide both localized and English versions for precision and legibility.

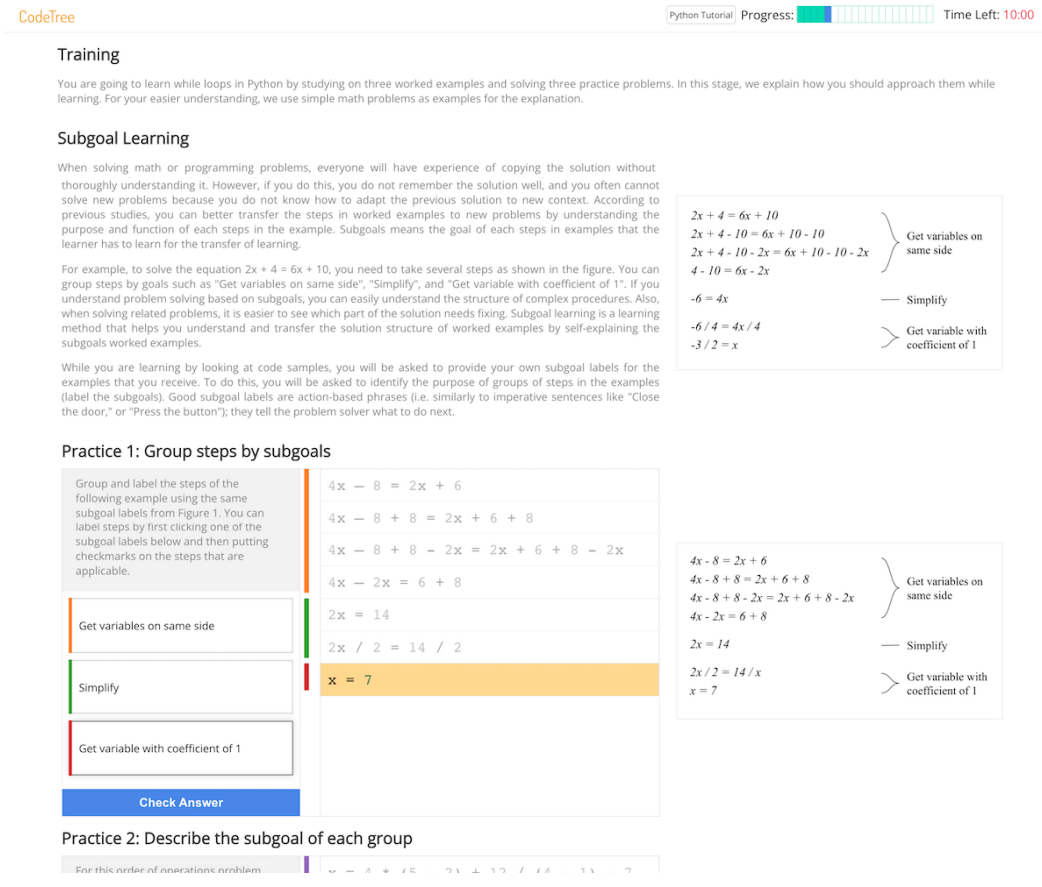
A.2.1 Subgoal Training Interface for Generate Condition.

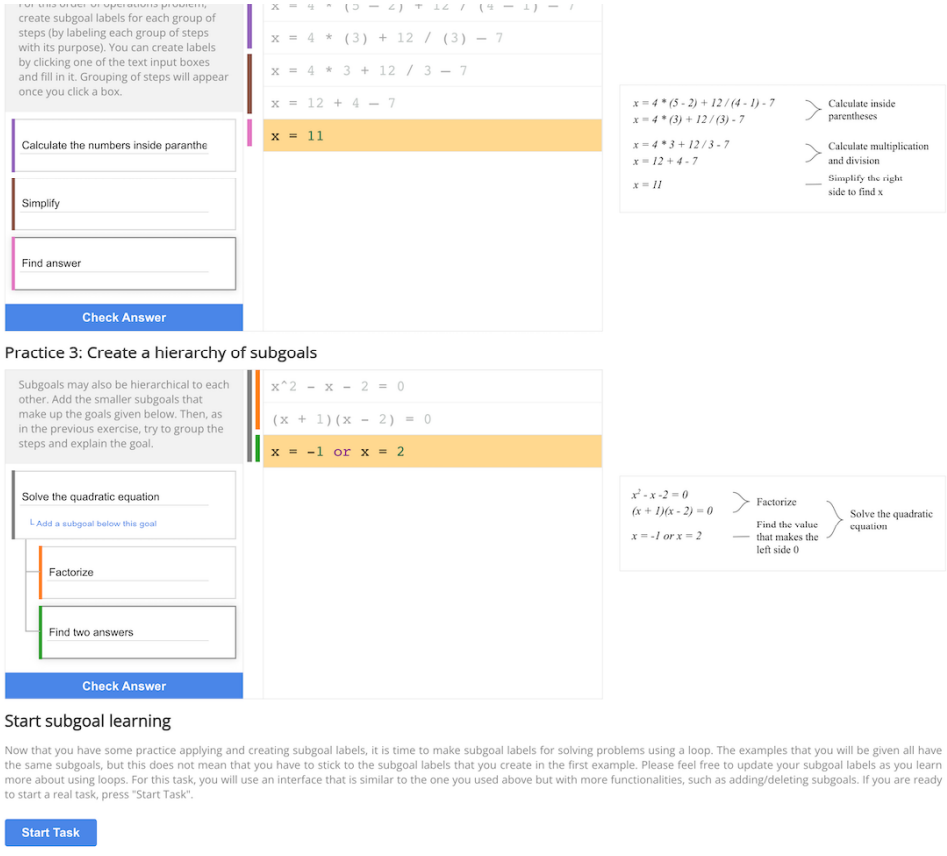
The original (i.e., localized) version.





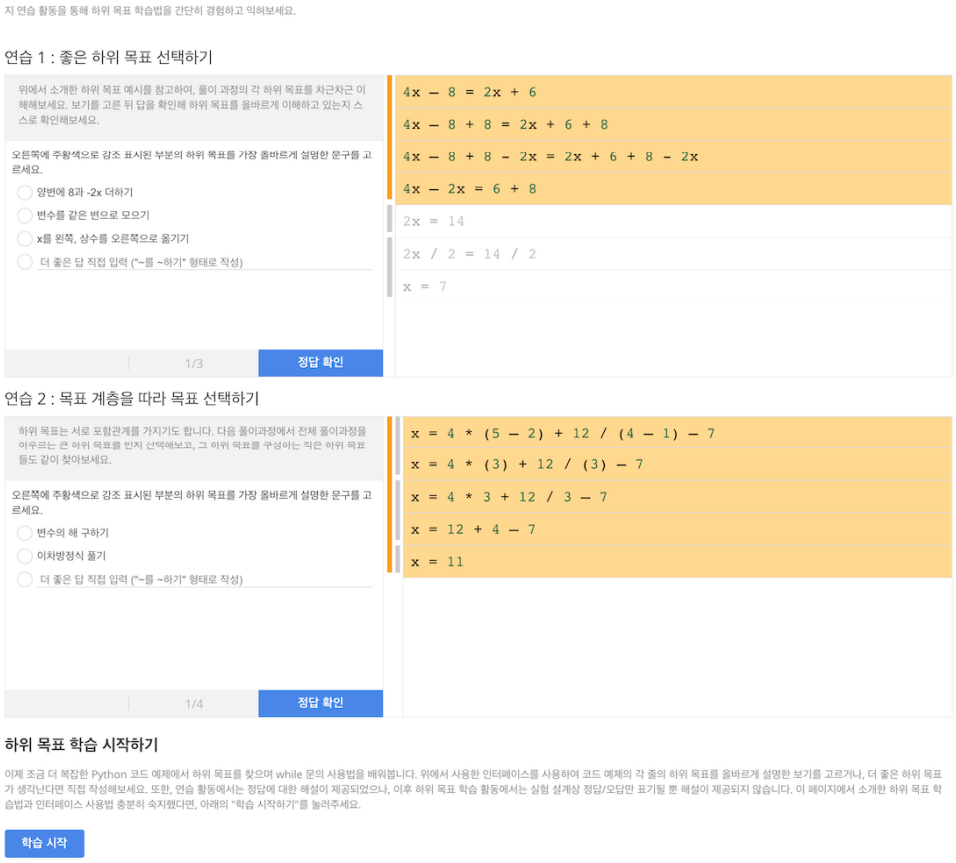
The English version.



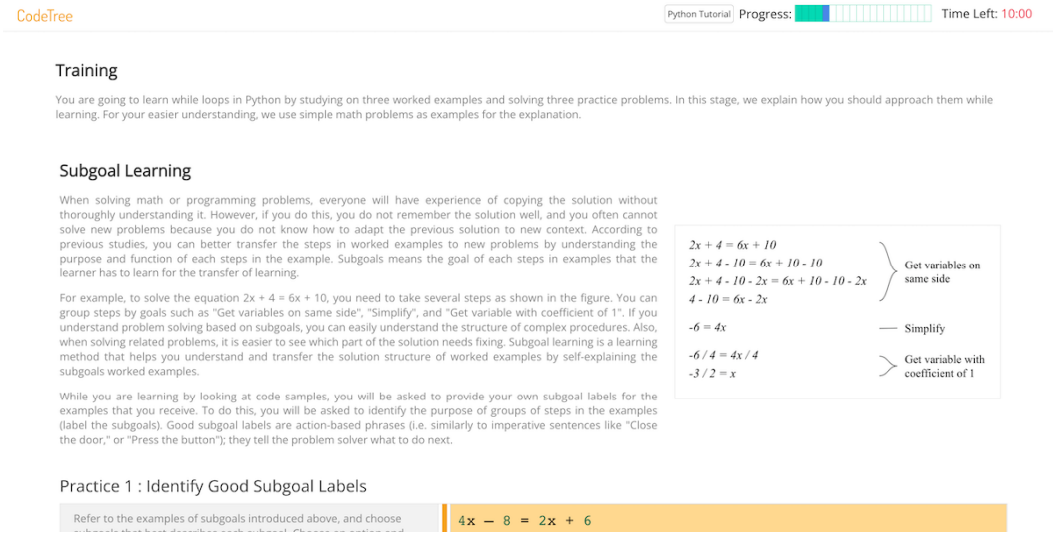


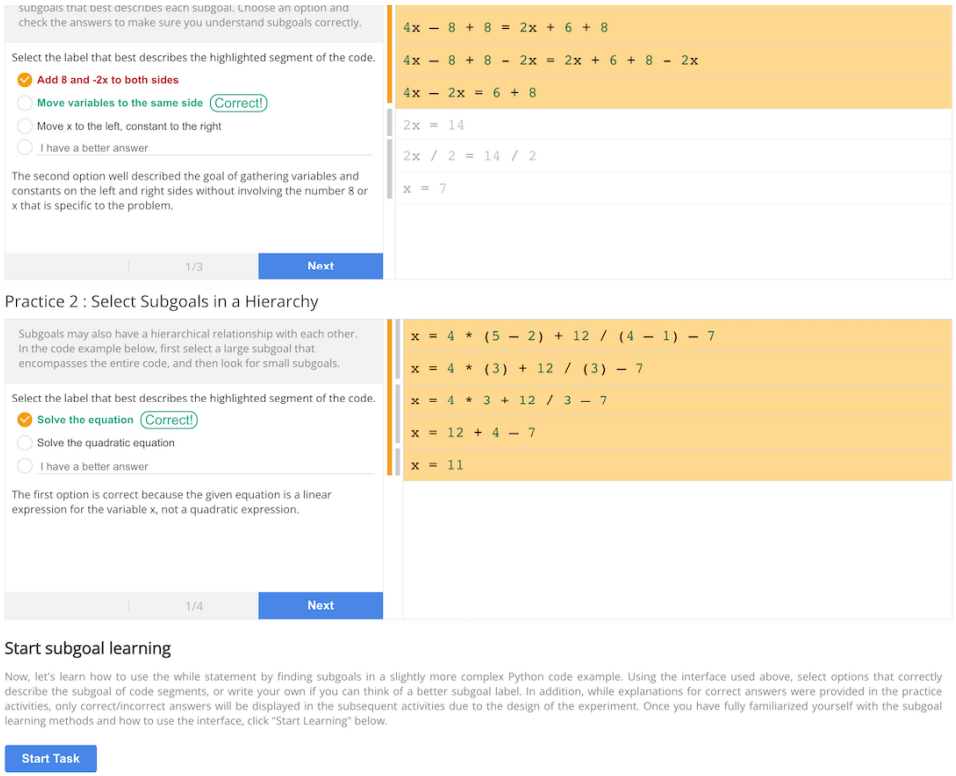
A.2.2 Subgoal Training Interface for Select Condition.

The original (i.e., localized) version.



The English version.





A.3 The Questionnaire for Measuring Cognitive Load

All the questions were answered with a linear scale of 10 (1: Not at all the case (전혀 그렇지 않다), 10: completely the case (완전히 그렇다)).

(1) The topics (usage of while loop) covered in the activity were very complex. (original: 활동에서 다룬 주제(while 문 사용법)는 매우 어려웠다.)
(2) The activity covered solution (code example) that I perceived as very complex. (original: 활동에서 다뤄진 예시 문제 풀이(예제 코드)가 내게는 매우 어려웠다.)
(3) The activity covered the concepts and definitions of while loop that I perceived as very complex. (original: 활동에서 다룬 while loop의 개념과 정의는 내게는 매우 어려웠다.)
(4) The instructions and/or explanations during the activity were very unclear. (original: 활동의 지시 사항과 설명이 명확하지 않았다.)
(5) The instructions and/or explanations were, in terms of learning, very ineffective. (original: 활동의 지시 사항과 설명은 학습하는데에 효과적이지 못 했다.)
(6) The instructions and/or explanations were full of unclear language. (original: 활동의 지시 사항과 설명은 불분명한 말로 설명되었다.)
(7) The activity really enhanced my understanding of the usage of while loop. (original: 활동을 통해 while loop 사용법에 대한 이해도가 높아졌다.)
(8) The activity really enhanced my knowledge and understanding of computing / programming. (original: 활동을 통해 프로그래밍에 대한 이해도가 높아졌다.)

(9) The activity really enhanced my understanding of the program code covered. (original: 활동을 통해 활동에서 다뤄진 코드에 대한 이해도가 높아졌다.)
(10) The activity really enhanced my understanding of the concepts and definitions. (original: 활동을 통해 다뤄진 개념과 정의에 대한 이해도가 높아졌다.)

A.4 Code Examples

The three code examples that participants studied. We also provide the line-level explanations generated by the Codex AI model in the form of comments. The explanations were not localized in the study.

Code Example 1

```
# Create a list of tips
tips = [15, 5.50, 6.75, 9]
# Create a variable to hold the sum
sum = 0
# Create a variable to hold the loop control value
lcv = 0
# Loop through the list
while (lcv < len(tips)):
    # Add the current tip to the sum
    sum = sum + tips[lcv]
    # Increment the loop control value
    lcv = lcv + 1
# Divide the sum by the number of tips to get the average
average = sum / len(tips)
# Print the average
print(average)
```

Code Example 2

```
# Create a list of 20 rolls
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
# Create a counter for the number of 7s rolled
count = 0
# Create a counter for the index of the rolls list
lcv = 0
# While the index is less than the length of the rolls list
while (lcv < len(rolls)):
    # If the roll at the index is a 7
    if (rolls[lcv] == 7):
        # Add 1 to the count
        count = count + 1
    # Add 1 to the index
    lcv = lcv + 1
# Print the count
print (count)
```

Code Example 3

```
# Initialize the number of primes to three (since 1, 2, and 3 are already primes)
count = 3
# Initialize the number to check if it is prime to four
num = 4
# Check if the number is less than 100
while (num < 100):
    # Initialize the counter variable to one less than the number
    lcv = num - 1
    # Initialize the variable that indicates whether the number is prime to true
    isPrime = True
    # Check if the counter variable is greater than one
    while (lcv > 1):
        # Check if the number is divisible by the counter variable
        if (num % lcv == 0):
            # Set the variable that indicates whether the number is prime to false
            isPrime = False
        # Decrement the counter variable
        lcv = lcv - 1
    # Check if the number is prime
    if (isPrime == True):
        # Increment the number of primes
        count = count + 1
    # Increment the number
    num = num + 1
# Print the number of primes
print (count)
```

A.5 Parsons Problem

The problem

Let's imagine that you have a list that contains amounts of rainfall for each day, collected by a meteorologist. Her rain gathering equipment occasionally makes a mistake and reports a negative amount for that day. We have to ignore those. We need to write a program to:

(a) calculate the total rainfall by adding up all the positive integers (and only the positive integers),
(b) count the number of positive integers, and
(c) print out the average rainfall at the end. Only print the average if there was some rainfall, otherwise print "No rain".

매일 내린 강수량을 측정한 데이터가 있습니다.하지만, 강수량 측정 기기에 오류가 있어 가끔 강수량이 음수로 기록되는데, 이런 측정값은 제외해야 합니다. 아래 기능을 가진 프로그램을 작성해보세요.

(a) 양수 값으로 기록된 강수량만을 더해 전체 강수량을 계산하기
(b) 양수 값으로 강수량이 기록된 날을 세기
(c) 양수로 기록된 강수량이 있을 땐 평균 강수량을 출력하고, 없을 땐 "No rain" 출력하기

The code segments to rearrange. Each segment is separated by "###".

```
lcv = lcv + 1
###
if (count > 0):
###
while (lcv < len(rain)):
###
else:
###
sumRain = sumRain + rain[lcv]
count = count + 1
###
lcv = 0
###
ave = sumRain / count
print(ave)
###
rain = [0,5,1,0,-1,6,7,-2,0]
sumRain = 0
count = 0
###
print("No rain")
###
if (rain[lcv] >= 0):
```