Code for Thought: Teaching Computer Programming Courses in a Post-COVID World

Wei Wei
Computer Science & Engineering Department
University of Connecticut
Storrs CT, USA
wei.wei@uconn.edu

***Abstract*—Many students chose remote learning during COVID. While most students have returned to campus in person in Fall 2021, dynamic pandemic conditions call for educational methodology that is highly effective in engaging students in learning. In this paper, we report our experience of "Code for Thought", a practice that we developed for programming-heavy courses in Computer Science and Engineering at the University of Connecticut. The main idea behind "Code for Thought" is to break down the learning into small steps, gradually easing students into solving more difficult problems by reinforcing the concepts, and building up their skills as well as confidence. Specifically, it includes small programming assignments given to students immediately after a class (e.g., we gave 31 "Code for Thought" assignments in a Data Structure course in two semesters). Each assignment is tied with the content just covered in class, with only one or two lines of code for students to fill in to complete the task, and is designed to be interesting (e.g., based on intriguing games, puzzles or math problems) to engage students. These small "bite-size" exercises provide students opportunities to immediately practice what they learned in a class and build on coding examples written by the instructors. We use data collected from several offerings enhanced with this practice to demonstrate its effectiveness in stimulating students' learning and helping them to gain confidence. We believe it can be particularly effective in teaching programming courses in post-COVID engineering education.**

I. Introduction

Computer programming is a critical component of the Computer Science discipline. It is considered a problem solving process, including formulating, planning and designing the solution, translating, testing, and delivery [1]. For students to be proficient in computer programming, they need to learn the syntax of a programming language, writing new programs, debugging, understanding, reusing and integrating existing programs, which require intensive cognitive activities and extended practice [1], [2]. As a result, teaching and learning of computer programming are rated as one of the greatest challenges in the area of Computer Science Education [3], [4].

It has been reported that traditional lecturing-based teaching is not sufficiently engaging in teaching computer programming [5]. This is particularly true in remote teaching during COVID, where lack of face-to-face interaction between the instructors and students makes the teaching even less engaging. Even after COVID, innovative teaching methodologies need to be developed and adopted in teaching computer programming [6], [7]. In this paper, we report a novel *"Code for Thought" (CFT)* methodology that we designed for teaching computer programming and our experience of using it in three semesters in a computer programming course at the University of Connecticut (UConn).

The main idea behind CFT is to break down the learning into small steps, gradually easing students into solving more difficult problems by reinforcing the concepts in a timely manner, and building up their skills as well as confidence. CFT includes a sequence of small programming assignments, each given to students immediately after a class and is due before the next class; each assignment is tied with the content just covered in class, with a few lines of code for students to fill in. These small "bite-size" exercises provide students opportunities to practice what they learned soon after a class. In the process, we set up good programming examples for students, illustrate difficult concepts, point out to student common mistakes, and encourage students to practice new skills, find better solutions and learn to debug.

CFT is in contrast to the traditional programming assignments (TPAs) that are spaced out in longer intervals (e.g., at least one or two weeks apart) and require significantly more time to complete. Our design of CFT is motivated by the drawbacks of TPAs: (i) TPA are not due immediately, and hence students are more likely to procrastinate [8], [9]. As an example, if a programming assignment is due in two weeks, many students will not start working on it until one week before the due date. (ii) When completing a TPA, students are using concepts, techniques, and skills that were covered one or two weeks earlier. At the same time, students are learning new materials but are not practicing them since they are busy with finishing the current TPA. (iii) Because students did not practice the new material in time, when they need to finish the next programming assignment, the skills that are needed for the assignment are not ready for the students to use and they need to review the material learned previously while the new material keeps coming. (iv) Because of the nature of a large programming assignment and the tendency of student procrastination, students may see a large number of error messages and bugs right before deadlines. This leads to significant pressure and frustration. Even if students finish the programming assignment in time, they do not have much time or energy to fully digest all the information.

To address the above drawbacks of TPA, CFT has the following features: (i) *Practice immediately after class.* CFT problems are designed to let students practice what they learned in a lecture immediately after the lecture is over. The problem is available right after a lecture and is due before the beginning of the next lecture. (ii) *No burden to students.* CFT problems are very small programming assignments. Usually, students only need to fill in one or two lines of code to complete the assignment, and hence is not burdensome to students. (iii) *Instantaneous feedback.* CFT problems are automatically graded to provide students instantaneous feedback (see Section II). In addition, the automatic grading does not increase grading load for TAs or professors. (iv) *Learning new skills from good examples.* In CFT assignment, we set up the template code to ensure students will solve the problem using the concepts, skills and techniques that they have just learned. Students have a tendency to use knowledge that they are already familiar with to solve problems. By setting up template code, we can enforce the learning and practicing of new skills. In addition, we also pass on good programming practice to the students through the template code.

Our experience of using CFT in three semesters in a data structure course at UConn demonstrates its significant benefits to both students and instructors:

- **Improved understanding.** Our analysis on students' homework grade demonstrates that CFT improves students' understanding of the course materials. A comparison between students who finish at least half of CFT problems score significantly higher in regular homework assignments (i.e., large programming assignments). In addition, the former has significantly lower variance in homework grade than the latter.
- **Timely review of course materials.** Our analysis also demonstrates that CFT helps students to form a habit of review course materials in time. In a semester with 31 CFT assignments, the students who finished the first 10 assignments (i.e., in the first 3-4 weeks), 73% of them finish at least 90% of all the 31 assignments.
- **Benefits to instructors.** CFT problems also provide fast feedback to instructors. Based on how well students did on the previous CFT problem, instructors can explain certain topics/concepts in more detail or in a different way if needed in the next lecture. This is much better than receiving feedback after two weeks when students finish a large programming assignment. By that time, instructors and students may have already moved on to a topic that is not relevant to the one in the programming assignment.

The rest of the paper is organized as follows. In Section II, we describe key factors that are important for successful design and implementation of CFT. In Section III, we describe multiple aspects of CFT design in helping students learn computer programming. In Section IV, we present our experience of using CFT in three semesters at UConn. Lastly, Section V concludes the paper and presents future work.

TABLE I
EXAMPLE PROBLEMS AND THE CONCEPTS COVERED IN CFT.

Example	**Concept**
Powerball lottery	Python expressions, statements
Card game 24	binary expression tree
Stock trading	priority queue
Card shuffling	Algorithm complexity
Fibonacci number	Algorithm complexity
Fibonacci & Geometric sequences	Generator
Pascal Triangle	Recursion and list comprehension
Handshakes and Catalan numbers	Divide and Conquer

II. CODE FOR THOUGHT: KEY FACTORS

When designing and implementing coding for thought problems, we need to consider multiple key factors.

- **Instantaneous feedback.** To serve as a tool to quickly review the materials covered in class, students need to obtain instantaneous feedback on whether they have completed the CFT problem correctly, or in other words, whether they have understood the class materials correctly. To achieve this instantaneous feedback, we use an automatic grading system. Specifically, once a student completes a CFT problem (typically fill in one or two lines of code), he/she can submit the code, and immediately see whether the code works correctly or not; if it is not correct, he/she can think about it, correct the code and resubmit for an unlimited number of times. The automatic grading system that we use achieves this goal. It provides unit tests and I/O tests, which can be used to grade the CFT problems. Specifically, we need to upload the skeleton code and solution code, and provide unit test and/or I/O tests to enable the automatic grading.
- **Interesting and easy-to-understand problems.** To motivate students to work on CFT problems, we strive to make the problems fun to work on and easy to understand. Many of the problems are designed based on intriguing games, puzzles or math problems. Table I lists several interesting problems and the concepts that they intend to cover. For instance, we use multiple games (e.g., Powerball lottery, card games, stock trading, card shuffling) and math problems (Fibonacci number, Fibonacci and Geometric sequences, Pascal Triangle, Catalan numbers). One example CFT problem that was designed based on a math problem will be described in more detail in Section III.
- **Small incentives.** In addition to making the problems engaging, we also provided a small incentive for students to complete CFT problems. Specifically, we offered up to 3 extra credits for completing CFT problems. For example, if there are 31 CFT problems in one semester, a student only get 3 credits after completing all 31 problems successfully; if a student completes 10 problems successfully, then he/she only gets 1 extra credit. We found this small extra credit was important to incentize students in completing more CFT problems (see Section IV-B).

When CFT is being implemented in a different institution, the instructors can consider the institution-specific conditions and include other key factors into CFT.

III. Application of Code for thought problems by examples

CFT assignments are intended for helping students digest course materials covered in a lecture right after that lecture. These problems can be designed and used in many different ways. We list several ways below, including helping students learn to debug and practice new skills, setting up good examples, cautioning students on common mistakes, illustrating difficult concepts, and helping students to find more efficient solutions.

A. Learning to Debug

Debugging is a notoriously hard problem. Many students have frustrating experiences debugging their program. In some CFT assignments, we purposely introduce bugs in the code and provide students the opportunity to debug the code. By going through this process, students can associate error messages with potential problems, and gain more confidence in debugging. Fig. 1 shows an example that includes bugs in the recursion part of a program.

When students run this code, they will see the following error message:

```
RecursionError: maximum recursion depth
exceeded
```

After reading this error message, students will check whether the recursion is defined properly, and they will find out that the base case of the recursion is not defined, and hence leading to infinite depth, causing the maximum recursion depth to be exceeded. Once they realize this error, they can easily correct the code.

B. Practicing New Skills

We can use CFT assignments to show students how to apply the knowledge they have just learned in a lecture. Students tend to use techniques they are familiar with to solve problems. Therefore, they may not consciously try out new techniques, even though they have just learned the new techniques in class. Instead, they tend to use old knowledge that they are comfortable with. To overcome this tendency, we set up CFT problems in ways so that students have to use the new techniques that they have just learned.

In this example (see Fig. 2), we convert a math problem into a coding problem to let students practice using lists, a basic and important data structure in computer programming. Specifically, the math problem is from a past MathCounts competition [10]:

A fair, twenty-faced die has 19 of its faces numbered from 1 through 19 and has one blank face. Another fair, twenty-faced die has 19 of its faces numbered from 1 through 8 and 10 through 20 and has one blank face. When the two dice are rolled, what is the probability that the sum of the two numbers facing up will be 24? Express your answer as a common fraction.

Fig. 2 shows the corresponding CFT program. It covers two important concepts: list comprehension and difference between index and value of a list. It further shows how to use simulation to calculate probability of a certain event. The comments in the program (in red) provide students guidance on how to solve the problem. We first use list comprehension to construct two lists that describe the two dice mentioned in the problem. We then generate two random numbers that reflect the resultant faces after rolling the dice. At the end, we ask students to fill in the code to check whether the sum of numbers on the faces of the two dice equals to 24 or not. The intention here is for student to fill in just one line of code.

In the main part of the Python code, we set up the template code to repeat this experiment $100, 1000$ times, and calculate the empirical probability that the sum of the two numbers facing up equals 24. If we did not set up the template code, students might solve the problem using other ways, and lose the opportunity of practicing lists that they just learned in class. By setting up the code and only asking students to fill in one line, students are forced to use lists, and practice how to use list comprehension, obtain the length of a list, and use index to obtain values from a list. All of the above are critical elements of using lists.

C. Setting up Good Examples

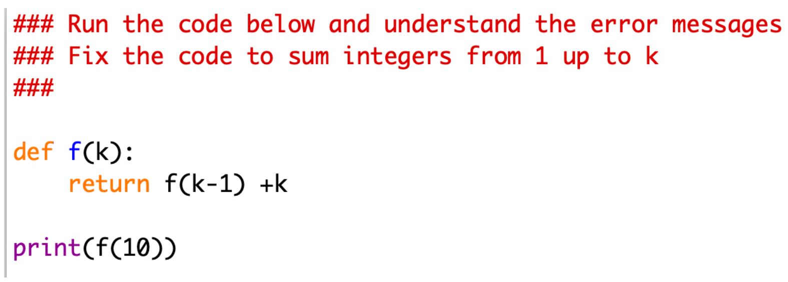
Whenever possible, we show students good programming practice to help them program more professionally and avoid mistakes. Our goal is that by emphasizing such practices through many small sample codes, they can be gradually embedded into a student's mind set and the student will then apply such good practice in his/her own programming. Such good practice will not only benefit students in this particular course, it will also benefit their future career. As an example, in Fig. 2, we point out it is better to use `len(die1)` and `len(die2)`, instead of using a fixed value 20, for better generality and readability of the code. Moreover, this example provides a general template for students to calculate the empirical probability of an event, which is commonly used in simulation and has many applications in practice.

D. Pointing out Common Mistakes

From our past experiences, we identify some common mistakes that students make. To avoid students making such mistakes in their large programming assignments, we show these mistakes in CFT assignments, so that students are aware of such mistakes and do not make them when working on their own computer programs. One example is shown in Fig. 3.

In this example, we are trying to reverse a list that contains integers from 0 to 9, and print out the reversed list. But what is printed out is "None" instead. This is because `L.reverse()` does not return anything. Instead, what happens is that `L` is reversed. In order to achieve what we desired, we need to do

```
L.reverse()
print(L)
```

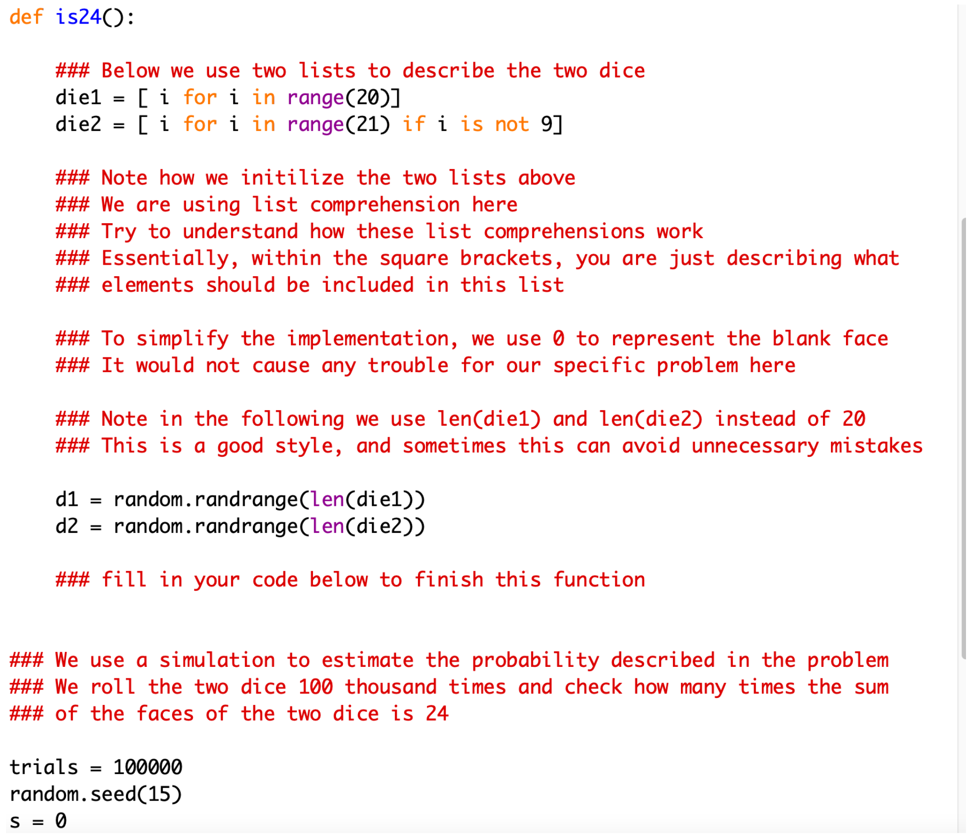


```
### Run the code below and understand the error messages
### Fix the code to sum integers from 1 up to k
###

def f(k):
    return f(k-1) +k

print(f(10))
```

Fig. 1. CFT example #1: learning to debug.



```
def is24():

    ### Below we use two lists to describe the two dice
    die1 = [ i for i in range(20)]
    die2 = [ i for i in range(21) if i is not 9]

    ### Note how we initilize the two lists above
    ### We are using list comprehension here
    ### Try to understand how these list comprehensions work
    ### Essentially, within the square brackets, you are just describing what
    ### elements should be included in this list

    ### To simplify the implementation, we use 0 to represent the blank face
    ### It would not cause any trouble for our specific problem here

    ### Note in the following we use len(die1) and len(die2) instead of 20
    ### This is a good style, and sometimes this can avoid unnecessary mistakes

    d1 = random.randrange(len(die1))
    d2 = random.randrange(len(die2))

    ### fill in your code below to finish this function


### We use a simulation to estimate the probability described in the problem
### We roll the two dice 100 thousand times and check how many times the sum
### of the faces of the two dice is 24

trials = 100000
random.seed(15)
s = 0
```

Fig. 2. CFT example #2: practicing new skills.

E. *Illustrating Difficult Concepts*

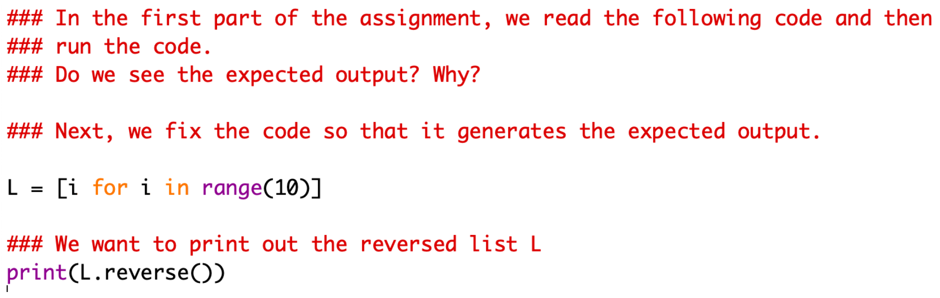
Some concepts in programming are difficult to grasp, and require multiple ways of illustration and practices from the students to achieve a deeper understanding. Generator in Python is such a concept. It is used in the course material, but is not the focus of the course. In order to make students gain familiarity with the concept of generator, we composed a CFT problem (figure omitted), which uses multiple examples to illustrate and demystify this difficult concept.

F. *Finding Better Solutions*

As students gain more and more experience with computer programming, a higher requirement for them is to be able to write more efficient programs (i.e., with lower running time), instead of simply a program that runs. This is a significant leap for many students, and requires careful thinking from the students. We use CFT assignments to provide students with guided exercises to achieve such a leap gradually over time. One example that we used is calculating Fibonacci numbers. We first provide students with a sample code that prints out the number of functions calls in inefficient ways. We then guide the student to fill in only one line of code to implement a more efficient efficient way for the calculation.

IV. Effectiveness of Code for Thought

We applied the CFT approach in a data structure course, one of the most important programming courses for the students in the Computer Science & Engineering Department at the University of Connecticut. In this course, students learn



```
### In the first part of the assignment, we read the following code and then
### run the code.
### Do we see the expected output? Why?

### Next, we fix the code so that it generates the expected output.

L = [i for i in range(10)]

### We want to print out the reversed list L
print(L.reverse())
```

Fig. 3. Code for thought example #3: pointing out common mistakes.

important data structures and apply them in problem solving through extensive programming practices using Python.

We report our experiences of using CFT in three semesters, indexed in chronological order. In Semesters 1 and 3, there were two sections in each semester, and we applied CFT in both sections. In Semester 2, there were two sections: in one section, we provided a total of 3 extra credits for CFT assignments, while in the other section, students had access to CFT assignments, but did not get extra credits for doing them and hence had less incentives in working on CFT assignments.

In the following, we first report the results in Semester 1 and 3. We then report the results in Semester 2, with a focus on comparing the experience of the two sections in that semester. In Semester 1, there were 20 CFT assignments; in Semesters 2 and 3, we significantly extended the number of CFT assignments to 31. In all the three semesters, the major assignments were homework assignments, which were large programming assignments that were due due in one to two weeks after posting the assignments. We examine whether finishing CFT problems help students in achieving better grade in their homework assignments. The number of homework assignments in the three semesters is 7, 11 and 10, respectively. For each semester, the homework grade is a weighted average of all the homework assignments. Similarly, CFT grade is an average of all the CFT assignments.

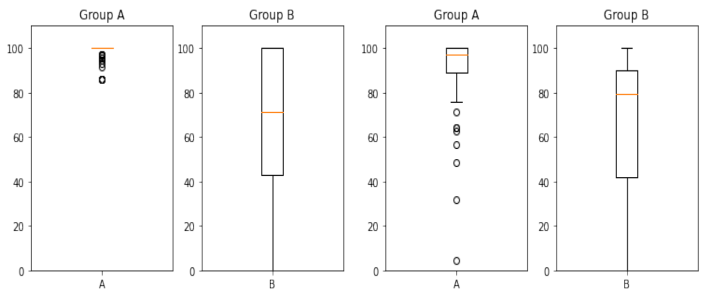
A. *Result for Semesters 1 and 3*

There were 198 and 211 students in Semesters 1 and 3, respectively. For each of these two semesters, we measure the effectiveness of the CFT approach by comparing the homework grades of two groups of students. Specifically, we divide students into two groups based on how many CFT assignments they completed. The students in Group A finished at least half of all the CFT assignments; the rest of the students belong to Group B. We compare the homework grades of the students in these two groups.

From Table II, we can clearly see that for both semesters, the two group of students achieve very different homework grades: the students in Group A obtain much better homework grades than those in Group B, demonstrating the benefits of completing CFT assignments. Specifically, the mean homework grade of the students in Group A is 32.4 and 26.4 points higher than those in Group B in Semesters 1 and 3, respectively. The

TABLE II
COMPARISON OF HOMEWORK GRADES OF TWO GROUPS OF STUDENTS (SEMESTERS 1 AND 3).

	Semester 1		**Semester 3**	
	Group A	**Group B**	**Group A**	**Group B**
count	117	81	95	116
mean	98.8	66.4	90.8	64.4
stdev	3.4	33.8	15.3	34.0
min	85.7	0	4.5	0
25%	100	42.9	88.7	41.9
50%	100	71.4	97.0	79.3
75%	100	100	99.9	89.9
max	100	100	100	100



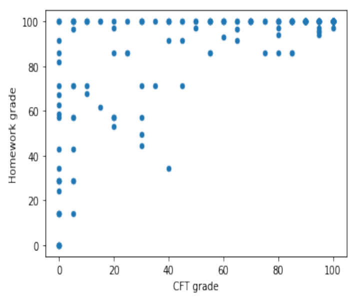
(a) Semester 1 (b) Semester 3

Fig. 4. Boxplots of homework grades for Groups A and B for Semesters 1 and 3.

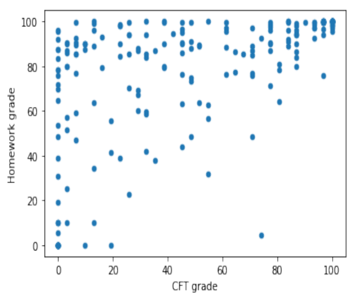
grades of the median and 25th percentile also show significant differences.

Fig. 4(a) plots the homework grades for Groups A and B for Semester 1. The results are consistent with the values in Table II: we observe more uniform higher grades for the students in Group A; for the students in Group B, the grades tend to be lower and the variance is significantly larger. Fig. 4(b) plots the results for Semester 3. It shows similar trends as those observed for Semester 1.

Figures 5(a) and (b) plot the homework grade (scaled to 100) versus CFT grade (scaled to 100) for the students in Semesters 1 and 3, respectively. We see that students with higher CFT grades tend to have higher homework grades. The correlation coefficient between homework and CFT grades is 0.64 for Semester 1 and 0.56 for Semester 3. These significant positive correlations confirm the positive impact of CFT on students' performance on homework assignments.



(a) Semester 1



(b) Semester 3

Fig. 5. Homework grade versus CFT grade.

TABLE III
COMPARISON OF HOMEWORK GRADE BETWEEN TWO SECTIONS IN SEMESTER 2.

	Section A	Section B
count	58	49
mean	76.1	72.9
stdev	25.5	25.9
min	0	0
25%	71.3	60.0
50%	84.8	83.7
75%	92.2	93.5
max	100	100

Last, we report the consistency of students completing CFT assignments throughout a semester. In Semester 1 (with 20 CFT assignments), for the students who finished the first 7 assignments (in the first 2-3 weeks of the semester), 78% of them finish at least 90% of all the 20 CFT assignments in the entire semester. In Semester 3 (with 31 CFT assignments), for the students who finished the first 10 assignments (in the first 3-4 weeks of the semester), 73% of them finish at least 90% of all the 31 assignments in the entire semester. The above results demonstrate that CFT problems help students to form a habit of reviewing course materials in time.

B. *Results for Semester 2*

During this semester, students are in two sections, both have access to CFT assignments. The first section has 58 students and the second section has 49 students. In the first section (that was taught by the author), students were encouraged to work on CFT assignments and can get up to 3 extra credits for completing them, while in the second section (taught by another faculty), no extra extra credit was given. For the 31 CFT problems assignments to all students, i.e., considered as 31×58 and 31×49 assignments for these two sections respectively, 44% were completed correctly in the first section, while only 4% were completed correctly in the second section. The significantly different completion rate in these two sections thus provides us an opportunity to compare the impact of CFT on these two sections. In the following, we refer to the first section as Section A and the second as Section B. We first compare the homework grade of these two sections, and then compare the homework grade of the two groups in Section A.

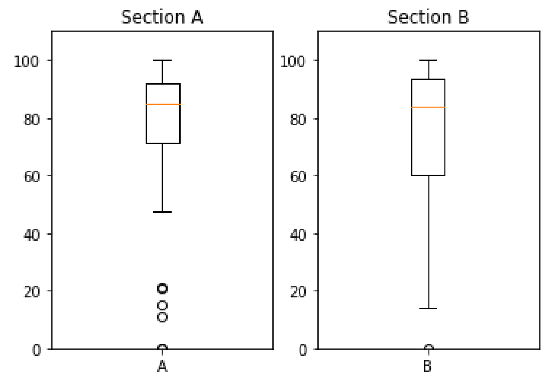


Fig. 6. Homework grade comparison between two sections (Semester 2).

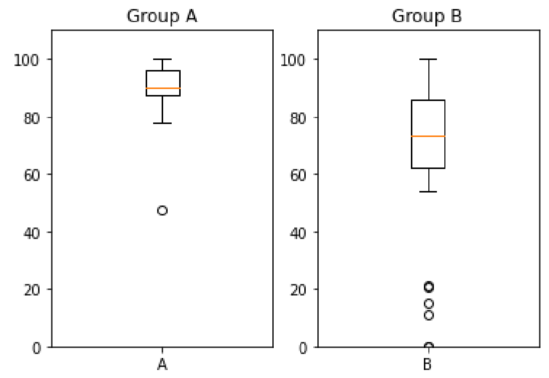


Fig. 7. Homework grade for Group A & B in Section A (Semester 2).

1) Comparison Between Two Sections: Table III lists the statistics of these two sections. The mean and median homework grade are 76.1 and 84.7 respectively for Section A, which are 3.2 and 1.6 points higher than that in Section B. The differences between these two sections are shown more clearly in the boxplots in Fig. 6. We see significantly lower variance in Section A than that in Section B. Although the median homework grades are not significantly different across these two sections, the 25-th percentile (71.3 versus 60.0) is significantly different. Specifically, 16% of homework grade is below 60 for Section A, while 26% of homework grade is below 60 for Section B.

2) Comparison Between Two Group in Section A: Following the methodology in Section IV-A, we divide the students in Section A into two groups, Groups A and B, based on how many CFT assignments they completed. Fig. 7 plots the boxplot of the homework grade of these two groups of students. We can clearly see that students in Group A did much better in homework assignments than their counterparts. For instance, the students in Group A have a mean homework grade of 89.5, while the students in Group B have a mean homework grade of 66.6.

Fig. 8 plots homework grade versus CFT grade for all the students in Section A. This figure again shows that students who have higher CFT grades tend to have higher homework

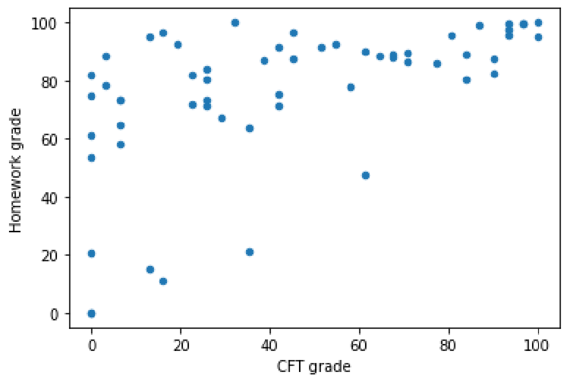


Fig. 8. Homework grade vs CFT grade for students in Section A (Semester 2).

grades. The correlation coefficient between homework grade and CFT grade is 0.55.

C. Feedback from Students

We have received very positive feedback from students on CFT. Below are quotes from students evaluations related to CFT problems. Some students like the timeliness and instantaneous feedback provided by CFT:

- "Code–for–thoughts allowed hands–on experience immediately after a lecture."
- "I personally really like the code for thought problems, and the ability to check each part of my code through mimir."

A lot of students feel that CFT help them practice and gain hands-on understanding:

- "A lot of code for thought assignments were given that helped reinforce understanding of concepts covered in class."
- "Code for thought problems were helpful for practice and exam preparation."
- "The code for thought assignments were helpful and reflected the material which appeared on the exams."
- "Making detailed code for thought problems which allowed an excellent way to both improve your grade and study for upcoming tests in material that the instructor knew was going to be tricky."
- "(What I really like) Offering codes for thoughts and practice exams."

Many students think CFT enhances their conceptual understanding of computer programming:

- "The Code For Thought segments were good ways of keeping us coding for habit."
- "(What I really like) code for thought which was extra credit and helped as conceptualize the specific topic we had in classes."
- "(What I really like) Give out code for thought that enhance student's knowledge on many details."
- "Code examples & code for thought assignments were useful in helping me understand the concepts of the class."
- "Code for thoughts really helped with earlier material."

Many students like the idea of CFT overall:

- "Code for thoughts were a very fair way of giving extra credit."
- "Code in class and Code for thought is awesome!"
- "Your code for thought exercises really helped me learn!"

V. Conclusion and Future Work

In this paper, we developed a novel methodology called "Code for Thought" (CFT) for teaching computer programming effectively in Computer Science and Engineering. CFT uses a sequence of small programming assignment, each after a lecture to help students reinforce their understanding, review course materials, learn how to debug and good programming practices. It provides student instantaneous feedback through an automatic grading system and the problems are designed to be fun and engaging. Our experience of using CFT in three semesters in teaching a data structure course shows that it is well received by the students. Our data analysis demonstrates its significant benefits to students in achieving better understanding of the course materials and forming a good habit of reviewing course materials in time. It is also beneficial for instructors to obtain quick feedback and adjust their teaching accordingly.

As future work, we plan to use the idea of CFT in our computer science courses and explore further improvements to the design and practice of using CFT in classrooms.

References

[1] F.P. Deek and J. McHugh. Problem solving and cognitive foundations for program development: An integrated model. In *Proc. of International Conference on Computer Based Learning in Science (CBLIS)*, pages 266–271, Nicosia, Cyprus, 2003.

[2] Chan E. Y. K. Lee V. C. S. Lam, M. S. W. and Y. T. Yu. Designing an automatic debugging assistant for improving the learning of computer programming. In *Lecture Notes in Computer Science, 5169*, 2008.

[3] T Jenkins. On the difficulty of learning to program. In *Annual Conference of LTSN-ICS*, 2002.

[4] Mendes A. J. Gomes, A. Learning to program - difficulties and solutions. In *International Conference on Engineering Education (ICEE)*, Coimbra, Portugal, 2007.

[5] L. Rolandsson. Changing computer programming education: the dinosaur that survived in school. an explorative study about educational issues based on teachers' beliefs and curriculum development in secondary school. In *Proc. of IEEE Learning and Teaching in Computing and Engineering*, 2013.

[6] J. Biggs. What the student does: teaching for enhanced learning. *Higher Education Research & Development*, 18(1):57–75, 1999.

[7] Bassey Isong. A methodology for teaching computer programming: first year students' perspective. In *I.J. Modern Education and Computer Science*, 2014.

[8] Hill D. A. Chabot A. E. Barrall J. F. Hill, M. B. A survey of college faculty and student procrastination. *College Student Journal*, 12(3):256–262, 1978.

[9] Norzad F. Badri Gargari R, Sabouri H. Academic procrastination: The relationship between causal attribution styles and behavioral postponement. *Iran J Psychiatry Behav Sci*, 5(2):76–82, 2011.

[10] Mathcounts competition series. https://www.mathcounts.org/programs/mathcounts-competition-series.