Universiteit Utrecht

Enhancing natural-language prompts for code completion tools using sub goals

L.C. Wannee

First Supervisor: J.T. Jeuring

Second Supervisor: H.W. Keuning

September 2023

Contents

1 Introduction 3

2 Introducing GPT-3 4

2.1 GitHub Copilot . . . 5

2.2 ChatGPT . . . 5

3 Task location and prompting the model 7

3.1 N-shot prompting . . . 7

3.2 Prompt engineering . . . 8

4 Subgoals 9

5 Literature gap 10

6 Experiment design 12

6.1 Research question . . . 12

6.2 Experiment approach . . . 13

6.3 Setup . . . 15

6.4 A sub-goal based prompt engineering approach . . . 17

7 Dataset 17

8 Results 19

8.1 Using zero-shot prompts for solving exercises . . . 19

8.2 Subgoal details . . . 21

9 Discussion 23

10 Threats to validity 25

10.1 Representativeness of dataset . . . 25

10.2 Model training . . . 26

10.3 Non-reproducible code 27
10.4 Remembering input 27

11 Conclusions **28**
11.1 Limitations and future work 28

12 References **29**

13 Appendix **36**

A CodeCheck example **36**

1 Introduction

Since the release of GitHub Copilot in 2021 and of ChatGPT in 2022, the use of large-language models (LLMs) for developing programs has exploded[1]. Besides other tasks, these LLM-based tools can help programmers by providing feedback and suggesting code completions for incomplete programs, and often even developing textual descriptions for the program itself. They also show the ability to work as artificial teaching assistants due to their ability to provide specific feedback to students [13].

To request code generation, Copilot and ChatGPT are prompted via snippets of natural language. The quality of the generated code is affected significantly by the structure and specific wording of these prompts [7, 38], making the ability to write the prompts a crucial skill indeed. Recent research has uncovered that LLMs can be "taught" to approach a problem step-by-step, which has shown to be beneficial to the success rate of the generated code [35, 36].

Tools like ChatGPT and Copilot create both opportunities and challenges for computer science education [7]. On the one hand, these tools can help students by fixing problems in their code or by explaining difficult steps [14]. On the other hand, it may result in an increased dependency on the tool, with possibilities for academic misconduct [3]. It has also been shown that generated code often contains imperfections. Many researches claim that computer science programs will need to pay more attention to evaluating generated code, code quality, and higher-level program design, given both the strengths and weaknesses of using code generated by LLMs, and still expect human elements to play a significant role in programming at this stage [27, 32, 2, 1, 22].

Due to the potential that LLMs have for assisting students while being prone to faulty generations or misuse, we need to develop methods for writing prompts

[1]SimilarWeb reported 100 million users two months since the launch of ChatGPT, see `https://www.thehindubusinessline.com/info-tech/social-media/chatgpts-popularity-tops-globally-100mn-users-in-2-months/article66470565.ece`, accessed: 04-02-2023

for these tools that lead to good quality results with regards to code generation. This is especially important for students, but also for all other user groups.

There is evidence that sub-goals are an effective method for teaching coputer programming [16]. Dividing problems into sub-goals is a well-known approach to problem-solving [16, 5] and has similarities to a widely researched prompting method known as Chain-of-thought (CoT) prompting, which aims to break down the problem-solving process into steps for the model to take one by one [35]. We argue that using sub-goals in the prompts will be an effective way to generate high-quality answers from LLM-based tools. In this paper, we want to verify that the competency to formulate sub-goals is indeed important when using LLM-based tools to develop programs. The research question we address in this paper is: ***"Does specifying sub-goals in the prompts for GitHub Copilot and ChatGPT improve the quality of generated code for Python problems that should be solvable by students that have completed CS1?"***

We answer our research question by investigating how Copilot and ChatGPT deal with sets of programming problems. We submit the corresponding prompts to Copilot and ChatGPT and analyze the results. We then attempt to engineer the prompts of those problems for which the respective LLMs are unsuccessful using sub-goals in an effort to generate a correct result. Finally, we will discuss the significance and meaning of the results, threats to validity, and limitations of the study.

2 Introducing GPT-3

GitHub Copilot and ChatGPT are built upon a large language model called Generative Pre-trained Transformer 3 (GPT-3). GPT-3 is an auto-regressive transformer language model that was released by OpenAI shortly before the summer of 2022 [21]. At its core, an LLM predicts the most likely next token

from a preceding textual input. Their auto-regressive properties enable them to extrapolate from their own output - thus predicting tokens that follow upon each other [26, 24].The GPT-3 model is large: according to Brown et al. [4] GPT-3 uses 175 billion parameters, at least ten times as many as previously developed LLMs. In a collaboration between OpenAI, Microsoft and GitHub, Chen et al. [6] trained GPT-3 on a large dataset of programming code. The resulting model, Codex, is used in GitHub Copilot. ChatGPT was developed and released one year after as a closely related model, being based on a similar - though improved - version of the same LLM, and with a different aim [20].

We chose Github Copilot (more specifically, its Codex model) and ChatGPT for this study for their ease of access and the model's aims. Codex was developed specifically as an "AI-pair programmer", in particular for the Python programming language, among others.

2.1 GitHub Copilot

GitHub Copilot utilizes a version of the GPT-3 model specialized for code generation [34], the code-davinci-002 model. It is accessible as a plugin from a number of programming IDEs [2]. GitHub Copilot's functionality can be divided into two parts [2]: One, suggesting code snippets while a user develops code. Often these code snippets complete the line of code the user is writing. Two, Copilot can interpret a natural language problem prompt in comments, and suggest code solving the problem described in the prompt. Figure 1 shows an example for both functionalities.

2.2 ChatGPT

ChatGPT is another AI-driven tool, which is not solely trained on program code, unlike Copilot. Using OpenAI's text-davinci-003, this AI is capable of conversing

[2] 4 different IDEs have been listed on the corresponding GitHub Docs page, see `https://docs.GitHub.com/en/copilot`, accessed: 03-03-2023

```
1   a = 16
2   b = 11
3   c = 7
4
5   print("{} {} {}".format(a, b, c))
```

```
arr = [5,2,6,6,1,5,0,12,1,10]

# write a method that takes an array and returns each position where a number is repeated

def findRepeats(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            if i == j:
                continue
            if arr[i] == arr[j]:
                print("The number %d is repeated in position %d and %d" % (arr[i], i, j))
```

Figure 1: Examples of GitHub Copilot functionality. The top image shows an example of the first functionality, and the bottom image an example of the second.

in natural language with the user and is deployed in a chatbot format. This enables it to use information that was provided earlier in the same session by the user. A web demo of ChatGPT was launched in November 2022, shortly after the launch of Copilot. The major difference between ChatGPT and Copilot lies in how the underlying model is trained.

Like Copilot's Codex, it's based on the GPT-3 architecture and its 500 billion token training set. Unlike Codex, however, ChatGPT is additionally trained on supervised learning via simulated natural language conversations [20, 4], whereas the former is trained on repositories for programming language only. However, ChatGPT has demonstrated good abilities to solve programming problems as well. It is therefore interesting to see how its qualities compare to those of Copilot. Compared to other available LLMs and Masked Language Models (MLMs), ChatGPT is considered the game changer by Wang et al. [34] for its ability to perform well in in-context learning and prompting.

3 Task location and prompting the model

Copilot and ChatGPT can be prompted to generate code by the user with a natural language description detailing what the code should do. The underlying LLM then attempts to compare this prompt to the natural language snippets it was trained on, and creates a response that is formed by predicting the likeliest response per snippet. The model's ability to determine what part of the model's training set is useful to generate an answer to the prompt is called "task location".

3.1 N-shot prompting

Prompting an LLM using a single description without any further examples to aid the model's task location is known as "Zero-shot prompting". Several researchers [6, 7] attempt to improve GPT-3's task location by first presenting the model with examples of similar problems to solve [24]. This is known as "n-shot prompting", where n is the amount of given examples. Prompting with $n > 1$ examples shown to the model is called "Few-shot prompting". A currently popular research topic is to use few-shot prompting to show examples to the model on how to divide a problem into intermediate steps to improve the model's reasoning capabilities. This is known as CoT-prompting [9, 35, 11]. This approach was inspired by Nye et al. [19]'s approach known as "Scratchpad", which had the model show its work and how it achieved its answer, encouraging the model to break the answer into steps. Current research investigates the effectiveness of chain-of-thought for smaller language models, which gives additional indication that the chain-of-thought approach is effective for LLMs too. For example, Ho et al. [10] use a fine-tuning approach to generate reasoning capabilities in smaller language models, and Shridhar et al. [29] propose an approach using semantic decomposition: breaking up the problem into smaller subproblems to solve, and use that to solve complex reasoning problems.

Instead of priming the model via few-shot prompting, we have chosen to not prime the model with examples and instead use zero-shot prompting. This is because we believe this to be the closest to the normal setting found in most educational environments. The zero-shot prompting method also works well given the chosen dataset for the experiment, which we will elaborate on later. Reynolds & McDonnell [24] have also shown that GPT-3, in some cases, performs identically - if not better - when using zero-shot prompting compared to one-shot or few-shot prompting.

3.2 Prompt engineering

Denny et al. [7] and Zhou et al. [38] demonstrate that the quality of the code generated by LLM-based tools is heavily affected by the information provided in the prompt. For this reason, Denny et al. [7] investigate how to engineer prompts such that the resulting generated code is of better quality. Zhang et al. [36] propose a method to auto-generate CoT examples to aid in the few-shot prompting of LLMs. Shin et al. [28] propose an automated approach to prompting as a solution to its time-consuming nature for Medium Language Models, before the emergence of Copilot and ChatGPT.

The activity of engineering prompts is dubbed "prompt engineering". Prompt engineering is challenging, and careful prompting is needed to generate code [38, 24]. The knowledge pertaining to prompt engineering appears to be hard to acquire. Sarkar et al. [27], conclude from the interviews they performed that the act itself is of considerable difficulty and the art behind it seen as uncharted territory, which can only be explored through trial and error.

LLMs do not possess the same capabilities as humans to correctly answer a question, making them prone to misinterpreting prompts. Manshadi et al. [15] and Rahmani et al. [23] argue that ambiguity in the natural language description can cause an LLM-based tool to generate different output than what is expected.

However, there appears to be no consensus on what kind of prompts cause the most errors. Gao et al. [9] show that the main mistakes made by LLMs in reading prompts fall under the categories 'incorrect reasoning' or 'incorrect calculation'. Zhou et al. [37] specifically emphasize the difficulties that LLMs have with complex reasoning tasks when they differ significantly from what they were trained on. Veres [33] finds that semantics is the main error category in interpreting prompts.

From all these findings, we conclude that prompt engineering is important for ensuring that LLM code completion tools generate code that matches a user's expectations. Correct prompt engineering is not easy, however, and especially for students, guidance for prompt engineering will be required.

4 Subgoals

According to Catrambone [5], Margulieux et al. [16] and Lee et al. [13], splitting the goal of a problem to be solved into sub-goals has been a proven effective method in general problem-solving for a long time. The main benefit of introducing sub-goals is that it allows students to better adapt to novel problems when an established procedure to solve the problem does not fully work. Catrambone et al. [5] define a sub-goal as "a meaningful conceptual piece of the overall solution" . This means that when taken together, sub-goals should describe the conceptual steps to be taken in a procedure from start to finish. In our case, this procedure will take the form of "the solution to a programming problem". Morrison et al. characterize sub-goals as "function-based instructional explanations" [18], and thus each sub-goal is constrained to performing one single function. We develop our subgoals according to these two definitions.

The effectiveness of sub-goals has been proven across a number of STEM fields according to Morrison et al. [18], especially with respect to the cognitive load theory [16, 31]. Margulieux et al. [17] have specifically shown the usefulness

of sub-goals in a programming context. In their study, they implement sub-goals which follow a strict structure. The structure is explained in Figure 2 using a subgoal example. The first element is an imperative verb, an action that the sub-goal sets out to complete. Second is the affected object, most of the time a particular variable. Finally, all the way at the end, a condition may be introduced by preceding it with the word "if". A sub-goal with a condition need only be addressed if that condition is met, otherwise it will be skipped.

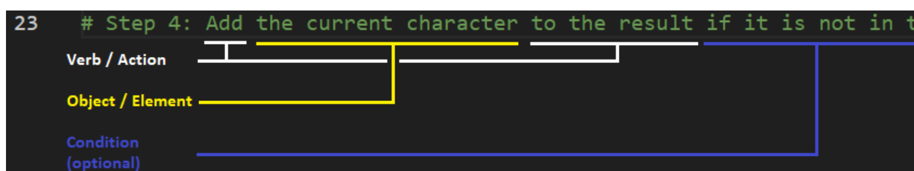

Figure 2: Example of subgoal with condition. The colored labels each indicate different parts of the subgoal structure.

The examined related works show that GPT-3 performs better when presented with a prompt that is structured using sub-goals compared to prompting using only a problem description.

5 Literature gap

Little literature exists on how to correctly engineer prompts to improve LLMs performance in code generation. Several researches do employ prompt engineering: Denny et al. [7] show that Copilot is pretty good at generating code for minor programming problems using prompt engineering, Wei et al.[35] show it can be used to overcome reasoning problems common to the underlying GPT-3 model, and Siddiq et al. [30] use it in their proposed framework as a way to tackle code of insufficient quality, but they do not elaborate on how prompt engineering is utilized with respect to the natural language involved or how it is practiced. Some researches do show it in different settings, however, which we will talk about here and how exactly those researches differ from ours.

A very close match with respect to the method in prompting is Kojima et al. [11]. Their research indicates that simply prompting LLMs to "think step-by-step" vastly improves performance, which they achieve by using two prompting instances. The first distills the reasoning steps from the model, and the second uses those steps to generate the answer. Our work differs from theirs on a few points. First, even though they also use GPT-3, they do not use Codex and instead focus on its performance relative to Instruct GPT-3 and PaLM. Second, Kojima et al. focus on researching reasoning capabilities of LLMs in a diverse set of reasoning task categories while our focus is specifically on programming code generation. Finally, while their method is a zero-shot technique, it does utilize two different prompts per problem, while we limit ourselves to a single prompt.

Zhou et al. [37] argue that while CoT has marked improvements in multi-step reasoning for LLMs, reasoning tasks that differ from those used to train the model have remained difficult for LLMs to solve. In their work, they leverage the CoT's principle of step-by-step computations as well as Scratchpad's approach of showing the steps themselves in order to develop "algorithmic prompting": they increase the amount of detail shown in the rationales and specify the steps taken in the prompt. The findings of Kojima et al. [11] and Zhou et al. [37] support our hypothesis that LLMs like those used in Copilot and ChatGPT for code generation benefit from approaching problems step-by-step.

Another closely related work that was recently published, Sakib et al. [25] delved into ChatGPT's ability to solve programming problems on a custom Leetcode dataset of 128 problems. They concluded that ChatGPT's ability to change answers via feedback after generation was very limited. They used a follow-up prompt via ChatGPT's conversational interface to attempt and guide ChatGPT into correcting itself, however, this was only successful in no more than 30% of the cases, highlighting the need for research on how to correctly (re-)prompt the model after an unsatisfactory result. While utilizing a research

design very similar to ours, they made use of a different dataset, did not evaluate the OpenAI Codex model, and used a different approach for failed exercises by issuing a follow-up prompt rather than editing the starting prompt and using that to allow new code to be generated from scratch.

Finally and perhaps the most closely related in goal, Denny et al. [8] have developed an approach to teach students to effectively prompt for LLMs via a new concept: a "Prompt Problem". They are given a visual representation of a problem they are required to solve by prompting the LLM correctly using natural language. This very closely matches our research goal and methods, the latter which we will elaborate on in the next section. The main difference is that when the solution is not satisfactory yet, the approach is to write a follow-up prompt asking for a revision of particular parts of the solution generated by the model, whereas we instead focus on the use of sub-goals in an entirely independent prompt.

6 Experiment design

Denny et al. [7] show that prompt engineering affects Copilot's ability to generate correct code. In an example they provided, by re-writing the prompts to elaborate on the computational steps to take, they were able to eliminate bugs in the code. We want to expand on this by taking this concept further, introducing sub-goals into the prompts and identifying factors and features of the exercises that Copilot would fail initially. Additionally, we introduce a different dataset to examine the effect of having different prompt structures and have ChatGPT attempt to solve the same problems.

6.1 Research question

We aim to answer the following main research question: ***"Does specifying sub-goals in the prompts for GitHub Copilot and ChatGPT improve***

the quality of generated code for Python problems that should be solvable by students that have completed CS1?"

We will answer this with an experiment that compares both models against a series of public programming exercises. We hope to be able to indicate a significant difference between subgoal and non-subgoal prompts. After that, we attempt to extract important features of subgoals, identifying characteristics of successful prompts and the pain points of prompts that failed. We hope this will help prompting of similar tools in similar settings in the future.

6.2 Experiment approach

We use two sets of publicly available coding problems and ascertain Copilot and ChatGPT's ability to solve them. One by one, exercises and their description are presented to the tool as prompts. We record the output the LLM in question generates in response to the prompt. When we observe that the tools fail to generate correct code for an exercise, we start the process of incorporating subgoals into the prompt.

Before creating a sub-goal-centered approach, we first define what "using sub-goals in prompt engineering" means. Margulieux et al. [16, 17], building upon Catrambone's work from 1998 [5], label groups of steps that contribute to a particular goal. We introduce sub-goals into the prompts with a structure identical to the ones introduced during their experiments: An imperative verb followed by an object. A conditional statement is added at the end whenever necessary. When taken together the sub-goal labels should form a complete list of actions describing the problem in steps from start to finish. After deliberation and testing using this structure, we decided to adapt it for this research to create the necessary sub-goals for the prompts.

With this structure in mind, the process of adding sub-goals to the exercise prompt goes as follows. First, we attempt to identify the sub-goals of the

exercise. This is done according to the following principles:

1. Start with the given variables. The first sub-goal always indicates the variables and information on them, like the data type or what the variable represents, that is provided by the exercise description itself.

2. Creating new variables should be explicitly indicated in their corresponding sub-goals. If one or more new variables should be used in the program, a sub-goal should exist indicating their creation.

3. Each sub-goal with the exception of the first described an operation to be executed on a particular object. Multiple complicated computations may happen under a single sub-goal, but they should all contribute to the described operation.

4. Loops, breaks and return statements should be explicitly indicated in their corresponding sub-goals. These are steps that change the order of execution in a code file and thus are important to indicate. "If" blocks are excluded here because they are determined by the conditional part of a sub-goal already.

The sub-goals per exercise should provide a complete list of programming steps to take in order to produce the desired output, making use only of the variables and elements defined in the problem description.

Once that is done and all sub-goals are identified, we engineer the prompt by adding a section at the end of the prompt describing each sub-goal. Then we check whether the sub-goals have led to a correct answer. If the resulting code still does not produce satisfactory output, we apply changes to the prompt iteratively, one by one and re-evaluate the exercise until the output is satisfactory. Changes are processed in the following order:

- Re-order subgoals that are not addressed/handled incorrectly by the generated code
- Identify specific elements in the prompt that the generated code failed to

address and reword those to ensure the model understands the element correctly

- Remove the standard description of the exercise, leaving only the subgoals in place, to avoid confusing the model and to make it focus explicitly on the steps

Finally, we analyze the initial and the engineered prompt to establish the missing factors that initially caused the incorrect code to be generated. In doing this, we hope to gain insight into what is considered important for Copilot and ChatGPT in prompts to generate code correctly.

An example is provided in Figure 3. Without using sub-goals like the top example, the model mistakenly assumes that it should only update the variable when the next evaluated value is larger than the stored value. This creates a bug where an equal value does not update the variable *position*. The result is that only the first occurrence of the largest value is stored. By adding sub-goals like in the bottom example, we can clarify that an equal value should also update *position*, eliminating the bug and making the code function correctly. It now indicates the last occurrence of the value instead of the first.

6.3 Setup

We use Copilot with all default parameter settings as plugin for Microsoft Visual Studio Code (VSC), a free coding environment that is often used by students. We choose to use the programming language Python for this experiment. There are two reasons to choose for Python. First, the model itself is trained on Python and is the most proficient when generating in this language [21].[3] Chen et al. [6] also use Python to evaluate Codex's code-writing abilities. Second, the dataset used in this research is also directed at Python programming. This

[3] GitHub reports the tool works "especially well in Python" and a few other languages. see `https://GitHub.blog/2021-06-29-introducing-GitHub-copilot-ai-pair-programmer/`, accessed: 16-2-2023

```python
# Given a list of integers, return the position
# of the last occurrence of the largest element.
# You may assume the given list has at least one element.
# Do not use the index, find, or max methods.

def lastOccur(arr):
    largest = arr[0]
    pos = 0

    for i in range(len(arr)):
        if arr[i] > largest:
            largest = arr[i]
            pos = i

    return pos
```

```python
# Given a list of integers, return the position
# of the last occurrence of the largest element.
# You may assume the given list has at least one element.
# Do not use the index, find, or max methods.
# Step 1: Given list of integers arr
# Step 2: Create variables to store the largest element and its index in
# Step 3: Loop through arr
# Step 4: Store the current element and its index if it is equal to or larger than the current largest element
# Step 5: Return the index of the largest element once the loop is done

def lastOccur(arr):
    largest = arr[0]
    index = 0
    for i in range(len(arr)):
        if arr[i] >= largest:
            largest = arr[i]
            index = i
    return index
```

Figure 3: Examples of prompting GitHub Copilot. In both images, code below the “def” was generated by Copilot from the comments above it. The top image uses the prompt directly. The bottom image uses the prompt and five determined sub-goals - the fourth of the five sub-goals is conditional.

is especially important given that other researchers have also used one of these dataset to evaluate Codex [7]. We will further elaborate on this in Section 7.

ChatGPT is prompted via its dedicated web application from OpenAI itself.[4] The prompts are fed into the request bar, in the same form as for Copilot. However, Copilot - being encased in a dedicated IDE - is aware from the start that the file is to be written in the Python language and that a code snippet is requested in the first place. We therefore precede the exercise text with the following statement: “Generate a Python code snippet according to the description below”.

Each exercise from the dataset is submitted to Copilot/ChatGPT directly in the form of a natural language prompt using the descriptions published with the exercises. Copilot/ChatGPT then generates code based on the prompt. That generated code is then submitted for evaluation via a number of test cases. Each test case contains a value for every variable and the expected result.

[4]The application is publicly accessible to users registered on their webpage. see `https://chat.openai.com/auth/login`, accessed: 27-9-2023

For each exercise, once the code suggestion is generated and tested, we record whether the code was successful at meeting the requirements of that exercise using the available test cases. We also identify and record a number of key features of the exercise, such as (but not limited to): the type of input and output, whether the exercise clarifies any edge cases, whether it gives an example, and whether it limits itself to particular input values. "Edge cases" are combinations of values for the variables in the exercise that warrant a different treatment than usual, like when particular values are 0 or a negative value (which causes some computations to return exceptions - for example, if said value is used as a position argument for an array).

6.4 A sub-goal based prompt engineering approach

In this final step of the research, we analyze the changes made during the engineering of prompts. Together with the features of the prompts and the specifications of the subgoals, we aim to draw conclusions about how and in which situations sub-goals may be useful.

7 Dataset

We use the 166 public exercises of CodeCheck, a list of educational programming problems also used by [7]. We intended to compare our own work with theirs and build upon it using the sub-goal method. The dataset is divided into categories of subjects, and is then further categorized by a particular kind of operation which the exercise intends to teach the user.

The exercises in the dataset are categorized into 4 supercategories by the authors, which revolve around a particular type of input on which certain operations are asked to be performed:

- Branches: 22 exercises. Varied input types.

- Strings: 29 exercises. 1 or more strings as input.
- Lists: 65 exercises. 1 or more lists of integers as input.
- 2D arrays: 50 exercises. 1 or more 2-dimensional arrays as input.

The questions from the dataset are aimed at returning single objects or an array of one object type. The code format is mostly limited to one code block with a single dependency. No external libraries are used. For each exercise, a link is supplied by CodeCheck which directs to an online editor for that exercise. The editor comes with:

- Comments explaining the expectations of the exercise and the desired result structure of the code
- A prewritten function name with arguments in line with the description in the comments (or a simple variable), which is where the result is to be stored/returned
- A check button, which causes the editor to check the result, alongside a reset button

If the check button is pressed, the editor will run a number of tests using different values for the arguments to check that the submitted code works as intended. It will then return the number of tests it ran as well as the number of tests in which the code returned the expected results ("successful" tests).

Additionally, we experimented with expanding the dataset using other sources. This however brought a number of concerns. To start, the dataset had to be compatible with the input method for Copilot and ChatGPT. For example, if the necessary description was embedded into an image or the description was not long enough, there would be little use for a natural-language driven tool like these. The dataset would also need to be reputable and thorough, representing the CS1 level of programming.

8 Results

We analyzed the 166 different exercises and recorded the results using a pass/fail notation, with a "pass" signifying that the code fulfilled the requirements of the matching description. For the description, we counted the number of lines and words per exercise, both for the textual description as well as the generated code. This was done for both Copilot and ChatGPT.

Due to the use of a uniform structure for writing the sub-goals, the position of particular words is consistent among sub-goals. This allowed analysis of particular parts of sentences. One such analysis is the analysis of the verbs used in the sub-goal, which indicate what that particular sub-goal aims to achieve or what element within the code it will add or change with regard to the eventual code generation.

8.1 Using zero-shot prompts for solving exercises

As the first step, we check if Copilot and ChatGPT are able to solve the exercises of Codecheck using zero-shot prompts without prompt engineering. The results of these attempts at the 166 exercises can be seen in Figure 4 for Copilot and Figure 5 for ChatGPT. We distinguish between four possible results:

1. Syntax fail: The generated code showed unhandled exceptions when run using the test cases. The defining characteristic is a failure to compile said code when entered and executed via a console.

2. Rule fail: The code used methods forbidden by the test cases. These are sometimes explicitly mentioned in the exercise description - usage in such a case triggers a special exception.

3. (Normal) fail: The code compiled and was executed to completion, but at least one test case failed to generate a correct result.

4. Pass: The results are identical to all the test cases' expected results.

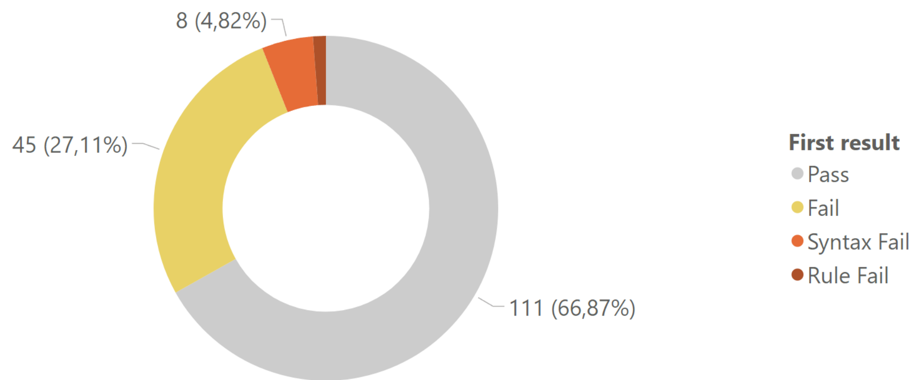

Figure 4: Chart of first-try code generation results for Copilot.

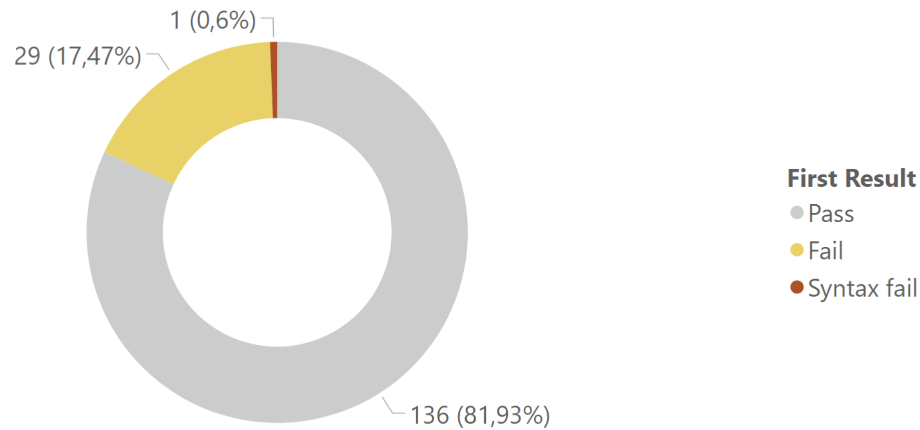

Figure 5: Chart of first-try code generation results for ChatGPT.

For Copilot, of the 166 exercises that it attempted to solve, 110 exercises were solved completely on the first attempt (66,27%) when evaluated using the test cases from CodeCheck. The remaining 56 failed exercises (33,73%) 9 (5,42%) contained Syntax failures and 2 (1,2%) Rule failures. We formulated subgoals for the failed exercises.

ChatGPT could solve 135 of 166 (81,3%) exercises in the first attempt. Of the 31 other failures (18,67%), only 1 (0,6%) contained a Syntax failure. Notable here is that ChatGPT also has the habit of generating additional elements during its attempt at generation. Namely a post-code explanation of the variables and

Number of generated lines of code

non-sub-goal files			Sub-goal files		
Application	Average	Std. Dev.	Application	Average	Std. Dev.
Copilot	12,81	6,49	Copilot	13,16	5,70
ChatGPT	13,34	7,32	ChatGPT	18,00	6,08

Figure 6: Averages and standard deviations for lines of code generated from prompts for non-sub-goal and sub-goal exercises, respectively.

Number of sub-goal lines

Application	Average	Std. Dev.
Copilot	5,80	1,75
ChatGPT	5,81	1,39

Figure 7: Average and standard deviations for lines of sub-goals used in the exercise prompts.

operations used, as well as test cases to show the code's output most of the time. It was also able to identify and correct spelling errors in the method names provided via the exercise. Like Copilot, some of the generated solutions came with additional comments explaining the steps taken in the code. ChatGPT's explanations, though part of its output, are not included in the assessment of whether the output is correct or not.

Figure 6 contains tables that denote the average amount of code lines for both sub-goal and non-sub-goal files, along with the standard deviation. Figure 7 contains a similar table for the number of sub-goal lines.

8.2 Subgoal details

We then analyzed the results of the previous stage via NLP to show the prevalence of particular (combinations of) words in the exercises with sub-goals implemented. The following characteristics have been extracted:

- The total length in characters of:
  - The exercise description
  - The established subgoals

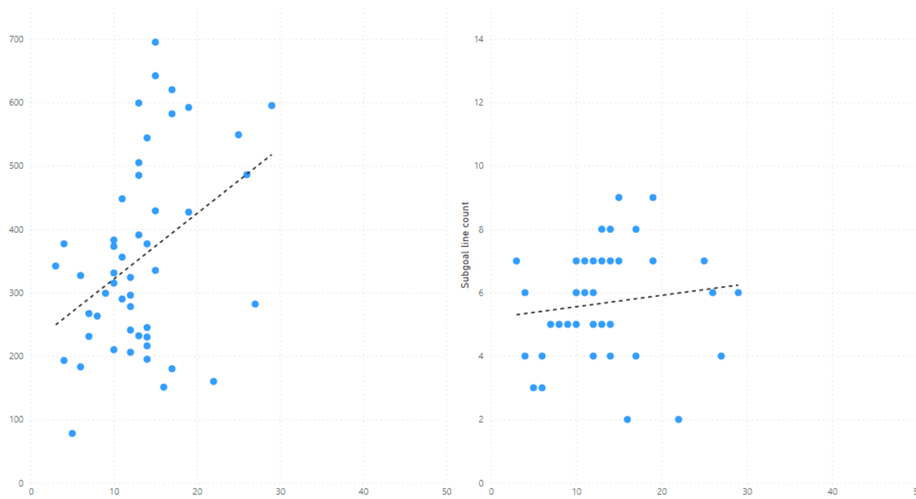

Figure 8: Plot of subgoal length/line count versus code line count for Copilot subgoal items.

  - The generated code solution

- The amount of times particular words are present in the exercise description

We also aimed to establish a relationship between the amount of code lines generated and the amount of lines used/total length of the subgoals. Figure 8 and 9 show the amount of generated code lines plotted against the subgoal length, including a trend line calculated using least-squares.

Comparing results from both LLMs, we find that 19 exercises were failed by both Copilot and ChatGPT in the first try. Of those, 6 belonged to the "2D Arrays" category. 8 belonged to the "Lists" category. We analyzed the faults made by both models in further detail for these 19 exercises. The reasons for failing the exercise for both models are displayed in Figure 10. Of the 19 exercises, both models exhibited the same output errors in 7 of them.

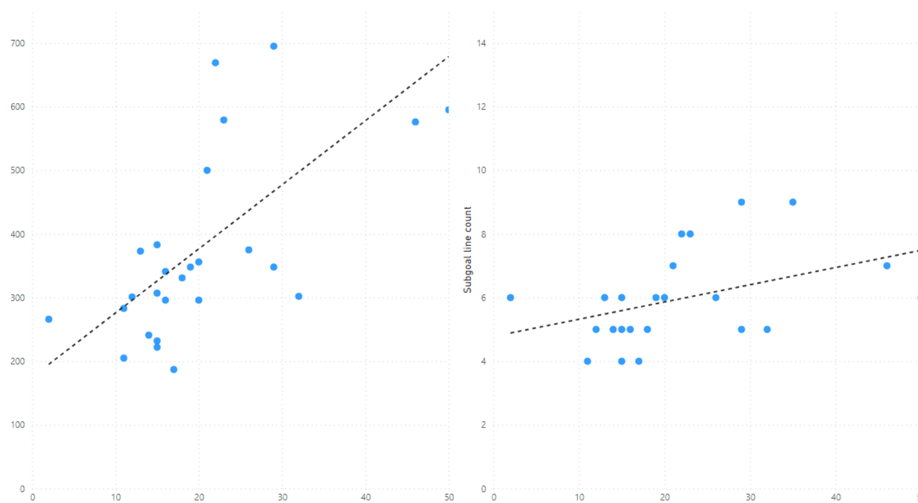

Figure 9: Plot of subgoal length/line count versus code line count for ChatGPT subgoal items.

9 Discussion

Notable is that ChatGPT has scored a much higher first-try solving percentage than Copilot. We would have expected Copilot to perform better since the experiments are coding-oriented and in the Python language on top of that. This would be close to the optimal use case for Copilot.

Codex and ChatGPT both draw from the same base model, however, they are fine-tuned differently. This might be giving ChatGPT an edge with respect to understanding the natural language in the prompts.

Using a two-sample t-test, we conclude that at a confidence interval of 95% there is not enough evidence to support that both LLMs create results of significantly different size or that they require more or less sub-goal lines to prompt properly. However, in sub-goal files, the difference between the amount of generated lines of code is statistically significant, suggesting that the LLMs do not generate code snippets of equal length on average when prompted using sub-goals.

Exercise nr.	Problem description	Copilot's error	ChatGPT's error
9	Given three input strings, print all but the shortest one	Only prints the longest string	Only prints the longest string
19	Given four numbers a, b, c, d, return true if exactly two of them are the same	The output is reversed	Fails at a specific case: [1,2,1,2] returns False, expected True
35	Given a string s, find the longest prefix that also occurs at the end (so that s = p + t + p), and return t, the string without the beginning and the end	Finds the shortest instead of the longest prefix	Result not properly returned
50	Given a string s and a character c (as string of length 1), return a string with the characters surrounding the first occurrence of c reversed. For example, if s is 'Hello' and c is 'l', return 'Hlleo'	Code ineffective	Swaps the wrong indices; offset by 1
51	Given a string s and a character c (as string of length 1), return a string with the characters surrounding any occurrence of c reversed. For example, if s is "Hello beautiful world" and c is a space, return "Hellb oeautifuw lorld". Skip any positions where c occurs twice in a row.	Code ineffective	Swapped characters are written twice
55	Given a list of integers, swap the first two and the last two elements	First two elements swap with each other instead, as do the last two	Correct elements are swapped but in the wrong order
71	Given a list of integers and a value, return the position of the element that is closest to the value. If there is more than one, return the position of the first one.	Values identical to the given value are also counted	Values identical to the given value are also counted
90	Given a list of integers, give the largest n so that the sum of the first n elements equals the sum of the remaining elements	Faulty method of calculation: does not account for the count starting at 0, returns erroneously due to this	Fails at a specific case: [1,2,3,4,10] returns 1, expected 4
91	Given a list of integers, give the largest n ≤ length / 2 so that the stum of the first n elements equals the sum of the last n elements	Assumes that the two halves have the same sum	Returns nothing but 1 or -1
97	Given a list of integers, replace each element with the average of its neighbours	Skips indices with single neighbours	At indices with single neighbours, uses the value at the current index instead of the missing neighbour
98	Given a list of integers and an integer n, remove all elements > n or < -n. Shift all remaining elements to the front, and return the number of removed elements	Fails to fill the empty spots with 0's which was indicated in the prompt	Fails to fill the empty spots with 0's which was indicated in the prompt
103	Given a list of integers, remove all adjacent duplicates	Triple pairs aren't properly removed	Code ineffective
113	Given a list of integers, return the largest sequence whose reverse also occurs somewhere in the list	Attempts to return the length of the sequence instead. Calculations for this are also incorrect	Ignores values inbetween and returns the sequence in the wrong order
124	On a chessboard, positions are marked with a letter between a and h for the column and a number between 1 and 8 for the row. Given an 8x8 array of integers and a string with such a position, return the element at that position	Row and column call swapped	Row and column call swapped
131	Given a two-dimensional array of integers, return the sum of all elements along all borders but not the corners.	Did not acknowledge "but not the corners"	Computes incorrectly if the array only has one row
151	Given a two-dimensional array of integers and row/column positions r, c, return the compass direction of the largest neighbor as a string N E S W NE NW SE SW	Returns "N" if the selected position itself holds the largest value	Returns "" if the selected position itself holds the largest value
160	Given a two-dimensional array a, return a list whose [i][j] element is the average of the neighbors of a[i][j] in the N, E, S, W direction	Returns only the elements that needs to be changed, not an array. Value is calculated correctly	Returns only the elements that needs to be changed, not an array. Value is calculated correctly
162	Given a two-dimensional array of integers, remove any adjacent duplicate rows by filling the duplicates with zeroes.	Uneven amounts of duplicate rows are not filled with zeroes	Uneven amounts of duplicate rows are not filled with zeroes
166	Repeat the previous exercise with subarrays that are not necessarily square. (Given a two-dimensional array of integers, return the position of the largest square subarray filled with zeroes, as a list of length 2. Return [-1, -1] if there is no such subarray)	Returns the largest horizontal length of the subarray, not its area	Often returns the incorrect row from which the subarray starts

Figure 10: Chart of first-try code generation results for ChatGPT. "Code ineffective" is used when the generated code returns the variable to be changed without modifications, or when it returns a default result (like "None" or "").

10 Threats to validity

Four main threats to the validity of this research have been identified. The first two arise from the nature of the exercises used in the research. One, the dataset should be representative of real-world scenarios where users attempt to utilize the software to resolve similar programming exercises. Two, it must be ensured that Copilot does not recognize the dataset from training and tailor its responses as a result. The third identified threat stems from the non-deterministic and black-box nature of the LLMs powering Copilot and ChatGPT. The fourth and final threat concerns the method by which information is fed to the programs themselves. Here we also elaborate on what kinds of validity are at risk for each risk: Internal, external, construct, and conclusion validity.

10.1 Representativeness of dataset

The dataset used is publicly accessible as a web resource. As this is a single source written in a particular style and language, it does not fully represent other programming problems available as those may be written by different authors and in different languages, which threatens the external validity. This The dataset was used because of its connection to the research of Denny et al. [7] and its low pre-processing needs, as the text could be directly used to feed the prompts for the LLMs without needing to be altered or converted.

For our prompts, we convert the problem description of each exercise directly into the initial prompt. While we did add a single line for ChatGPT to clarify the goal of the prompt and bring it on equal footing with Copilot, the exercise descriptions themselves are completely identical as what is available via the web resource.

10.2 Model training

According to Microsoft and OpenAI [20], GitHub Copilot was trained on public repositories. The vagueness of this statement is understandable, as the nature of the training dataset has a huge impact on the model, and rival parties will recreate their own versions should they find out. The problem here is that there is a chance that the dataset used in this research has been picked up by Copilot. According to an article on Software Conservancy's [12] website and Chen et al. [6]'s evaluation of GitHub Copilot with respect to its code-writing abilities, the model is trained on public repositories from GitHub. No information is available on which specific kinds of repositories were used. Whether GitHub is the only public repository it is trained on is unknown. The currently used dataset has not been found to be directly replicated in any publicly accessible GitHub repositories. ChatGPT uses a fine-tuned model of the same GPT-3 series, supposedly trained on simulated conversations amongst other uniquely generated training data [20], but is most likely also trained on the same initial set.

Should the exercises be present in the training set directly, it may have affected the results which challenges the construct and conclusion validity. To further verify the threat, we searched the web for indications of the dataset within Github. The main indicators to look for were repositories that contained the name, description or "math problem" in the title. For exercises originating from CodeCheck, no such repositories were found - other earlier investigated datasets were, which have not been included in this research. There is no telling however what share of the training data comes from Github, and what other sources or methods were employed to train the models.

10.3 Non-reproducible code

With GitHub Copilot and ChatGPT as closed-source and the output of the models being non-deterministic, they are essentially black boxes. As such, there is no direct reproduction possible of the generated answers [22]. Time or other factors may change the outcome of identical prompts.

The possibility of the AI models changing their answers challenges the internal validity of this research: it is possible that, in future experiments with the same setup, different results will be found. By keeping all of the accessible parameters for the models at their default values and documenting the nature of their prompts as well as some of their important features, we aim to minimize the impact of this threat. There is however no way to completely remove it. Future research may attempt to regenerate the answer to the same prompt multiple times in order to increase the sample size of solutions and decrease the amount of randomness involved, but it is not possible to completely control the output.

10.4 Remembering input

Over the course of reviewing available literature, we grew concerned that Github Copilot was being asked similar problems repeatedly, and thus was starting to recognize the question being asked, resulting in different output as a result due to retaining information between prompts and challenging the conclusion validity as a result. In preliminary experiments, there were instances where Copilot suggested snippets from previously written text or code directly, which raised further suspicions. For ChatGPT, there is a clear border between sessions, and we expect that information between sessions is not retained as each exercise was confined to its own session. For Copilot, while Github has a clause clarifying that Copilot for Business does not retain information between prompts [5], this is

[5] See `https://github.com/features/copilot/#faq`. Accessed: 23-3-2023

not the case for the freely accessible version of Copilot that is being used in this research. There is thus a possibility that information from prompts may have carried over to other prompts, which might have affected the model output.

11 Conclusions

This research was aimed to find out whether LLMs could achieve improved performance in attempting to solve Python programming problems using subgoal-based prompt engineering compared to using none. By expanding the prompts using our sub-goal method, we have demonstrated that Copilot and ChatGPT can solve exercises to which they gave incorrect solutions before.

The natural language structure of the sub-goals is derived from the previous work and is summarized in this work in a set of rules, giving prospective users a way to reproduce the method. While the nature of LLMs as a concept and the lack of information on Copilot and ChatGPT's training means that proving identical results with different datasets is difficult, it is clear that LLMs benefit from prompt engineering using this sub-goal-centered approach.

In a zero-shot setting, ChatGPT was shown to generate a higher percentage of correct solutions to Python programming problems than Copilot, while also generating additional textual output that may be of essence to the user. When generating code according to subgoals, it was shown that there was a significant difference in the length of the code generated between Copilot and ChatGPT.

11.1 Limitations and future work

Even during the short time in which this research was carried out, new LLMs have been developed, and current versions have been updated. With the amount of different LLMs accessible today, comparison between them can be very valuable on datasets such as this one from CodeCheck in determining their problem-solving skills in a programming context, which can serve to both give insight

into the overall capability of such models on a wider scale as well as advising educational institutions into considering what models to focus on investigating themselves.

We have limited ourselves to zero-shot prompting with the use of the problem descriptions provided directly by the dataset. For answer generation we maintained the approach of focusing on the first displayed answer and forgoing the chance to check any multiple solutions of the models. Evidence from other works indicates that regenerating responses with the same prompts have a chance of providing different and perhaps better results. We encourage further research to consider using this ability .

Finally, during the analysis of the gained data, we made attempts to find similarities between the code generated by the two models. However, this was not able to be completed due to time constraints. Particularly, measuring the similarity would require a template not only for the words used, but also the role of each code line: lines in generated code for the same exercise may both serve the same purpose, but not be seen as "equal" in language because the names used for declared variables may be different, or there might be differences in the style of writing. An opportunity for future work is to analyze generated code using characteristic symbols or writing used in particular lines of code, and reading the similarity between the two models based on that.

12 References

References

[1] Al Madi, N. How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*

(New York, NY, USA, 2023), ASE '22, Association for Computing Machinery.

[2] Barke, S., James, M. B., and Polikarpova, N. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang. 7*, OOPSLA1 (Apr 2023).

[3] Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., and Santos, E. A. Programming is hard – or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education* (Mar 2023), Association for Computing Machinery.

[4] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2020), NIPS'20, Curran Associates Inc.

[5] Catrambone, R. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General 127* (1998), 355–376.

[6] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power,

A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

[7] Denny, P., Kumar, V., and Giacaman, N. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education* (Mar 2023), Association for Computing Machinery.

[8] Denny, P., Leinonen, J., Prather, J., Luxton-Reilly, A., Amarouche, T., Becker, B. A., and Reeves, B. N. Promptly: Using prompt problems to teach learners how to effectively utilize ai code generators, 2023.

[9] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models, 2022.

[10] Ho, N., Schmid, L., and Yun, S.-Y. Large language models are reasoning teachers. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Jul 2023), Association for Computational Linguistics.

[11] Kojima, T., Shane Gu, S., Reid, M., Matsuo, Y., and Iwasawa, Y. *Large Language Models are Zero-Shot Reasoners.* Goodreads Inc., May 2022.

[12] Kuhn, B. M. If software is my copilot, who programmed my software? https://sfconservancy.org/blog/2022/feb/03/github-copilot-copyleft-gpl/. Accessed: 24-1-2023.

[13] Lee, C., Myung, J., Han, J., Jin, J., and Oh, A. Learning from teaching assistants to program with subgoals: Exploring the potential for ai teaching assistants, 2023.

[14] MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., Bernstein, S., and Leinonen, J. Experiences from using code explanations generated by large language models in a web software development e-book. SIGCSE 2023, Association for Computing Machinery, p. 931–937.

[15] Manshadi, M., Gildea, D., and Allen, J. F. Integrating programming by example and natural language programming. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence* (San Fransisco Airport, CA, USA, Jul 2013), Assosciation for the Advancement of Artifical Intelligence.

[16] Margulieux, L. E., Catrambone, R., and Guzdial, M. Employing subgoals in computer programming education. *Computer Science Education 26*, 1 (2016), 44–67.

[17] Margulieux, L. E., Morrison, B. B., Franke, B., and Ramilison, H. Effect of implementing subgoals in code.org's intro to programming unit in computer science principles. *ACM Transactions on Computing Education 20*, 4 (Oct 2020).

[18] Morrison, B. B., Margulieux, L. E., and Guzdial, M. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA, 2015), ICER '15, Association for Computing Machinery, p. 21–29.

[19] Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models, 2021.

[20] OpenAI. Chatgpt: Optimizing language models for dialogue. `https://openai.com/blog/chatgpt/`. Accessed: 24-1-2023.

[21] OpenAI. Openai codex. `https://openai.com/blog/openai-codex/`. Accessed: 26-1-2023.

[22] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), pp. 754–768.

[23] Rahmani, K., Raza, M., Gulwani, S., Le, V., Morris, D., Radhakrishna, A., Soares, G., and Tiwari, A. Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang. 5*, OOPSLA (Oct 2021).

[24] Reynolds, L., and McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI EA '21: Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), Association for Computing Machinery.

[25] Sakib, F. A., Khan, S. H., and Karim, A. H. M. R. Extending the frontier of chatgpt: Code generation and debugging, 2023.

[26] Sandoval, G., Pearce, H., Nys, T., Karri, R., Dolan-Gavitt, B., and Garg, S. Security implications of large language model code assistants: A user study, 2022.

[27] Sarkar, A., Gordon, A., Negreanu, C., Poelitz, C., Ragavan, S. S., and Zorn, B. What is it like to program with artificial intelligence? In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)* (Sept 2022).

[28] Shin, T., Razeghi, Y., Logan IV, R. L., Wallace, E., and Singh, S. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. In *2020 Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2020), EMNLP 2020, Association for Computational Linguistics.

[29] Shridhar, K., Stolfo, A., and Sachan, M. Distilling reasoning capabilities into smaller language models. In *Findings of the Association for Computational Linguistics: ACL 2023* (Toronto, Canada, July 2023), Association for Computational Linguistics, pp. 7059–7073.

[30] Siddiq, M. L., Casey, B., and Santos, J. C. S. A lightweight framework for high-quality code generation, 2023.

[31] Sweller, J. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review 22* (Apr 2010), 123–138.

[32] Vaithilingam, P., Zhang, T., and Glassman, E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2022), CHI EA '22, Association for Computing Machinery.

[33] Veres, C. Large language models are not models of natural language: They are corpus models. *IEEE Access 10* (2022), 61970–61979.

[34] Wang, Y., Pan, Y., Yan, M., Su, Z., and Luan, T. H. A survey on chatgpt: Ai-generated contents, challenges, and solutions. *IEEE Open Journal of the Computer Society* (2023), 1–20.

[35] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Proccedings of the Thirty-sixth Conference on Neural Information Processing Systems* (Nov 2022), NeurIPS 2022.

[36] Zhang, Z., Zhang, A., Li, M., and Smola, A. Automatic chain of thought prompting in large language models, 2022.

[37] Zhou, H., Nova, A., Larochelle, H., Courville, A., Neyshabur, B., and Sedghi, H. Teaching algorithmic reasoning via in-context learning, 2022.

[38] Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers. In *Proceedings of the 11th International Conference on Learning Representations* (May 2023).

13 Appendix

A CodeCheck example

prog.py

```
1
2  # Given a string in which words are separated by spaces,
3  # return the longest word. You may assume there is at least
4  # one word in the given list of words. If there are two words
5  # with equal length, return the first longest one.
6
7  def longestWord(s):
8      # Your code here...
```

CodeCheck Reset