Reeves, Brent; Sarsa, Sami; Prather, James; Denny, Paul; Becker, Brett A.; Hellas, Arto; Kimmel, Bailey; Powell, Garrett; Leinonen, Juho

Evaluating the Performance of Code Generation Models for Solving Parsons Problems with Small Prompt Variations

Published in:
ITiCSE 2023 - Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education

DOI:
10.1145/3587102.3588805

Published: 29/06/2023

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY

Please cite the original version:

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.
Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations

Brent Reeves  
Abilene Christian University  
Abilene, Texas, USA  
brent.reeves@acu.edu

Sami Sarsa  
Aalto University  
Espoo, Finland  
sami.sarsa@aalto.fi

James Prather  
Abilene Christian University  
Abilene, Texas, USA  
james.prather@acu.edu

Paul Denny  
University of Auckland  
Auckland, New Zealand  
paul@cs.auckland.ac.nz

Brett A. Becker  
University College Dublin  
Dublin, Ireland  
brett.becker@ucd.ie

Arto Hellas  
Aalto University  
Espoo, Finland  
arto.hellas@aalto.fi

Bailey Kimmel  
Abilene Christian University  
Abilene, Texas, United States  
blk20c@acu.edu

Garrett Powell  
Abilene Christian University  
Abilene, Texas, United States  
gbp18a@acu.edu

Juho Leinonen  
University of Auckland  
Auckland, New Zealand  
juho.leinonen@auckland.ac.nz

ABSTRACT
The recent emergence of code generation tools powered by large language models has attracted wide attention. Models such as OpenAI Codex can take natural language problem descriptions as input and generate highly accurate source code solutions, with potentially significant implications for computing education. Given the many complexities that students face when learning to write code, they may quickly become reliant on such tools without properly understanding the underlying concepts. One popular approach for scaffolding the code writing process is to use Parsons problems, which present solution lines of code in a scrambled order. These remove the complexities of low-level syntax, and allow students to focus on algorithmic and design-level problem solving. It is unclear how well code generation models can be applied to solve Parsons problems, given the mechanics of these models and prior evidence that they underperform when problems include specific restrictions. In this paper, we explore the performance of the Codex model for solving Parsons problems over various prompt variations. Using a corpus of Parsons problems we sourced from the computing education literature, we find that Codex successfully reorders the problem blocks about half of the time, a much lower rate of success when compared to prior work on more free-form programming tasks. Regarding prompts, we find that small variations in prompting have a noticeable effect on model performance, although the effect is not as pronounced as between different problems.

CCS CONCEPTS
• Social and professional topics → Computing education; • Computing methodologies → Artificial intelligence.

KEYWORDS
academic integrity; AI; artificial intelligence; ChatGPT; code generation; code writing; Codex; computer programming; Copilot; CS1; deep learning; generative AI; introductory programming; GitHub; GPT-3; large language models; machine learning; ML; neural networks; natural language processing; novice programming; OpenAI

ACM Reference Format:

1 INTRODUCTION
Code generation tools such as Github Copilot\(^1\) which is powered by OpenAI’s Codex\(^2\) model have gained significant attention in the last year due to their ability to produce highly accurate code for a variety of programming tasks [3, 38]. Their rapid and widespread availability has led to speculation about the future of computing education and programming in general [1, 41]. Recent work has shown that models such as Codex can solve typical introductory programming problems with greater accuracy than most students [14]. This raises concerns around plagiarism and over-reliance, as students may be tempted to use code generation tools to complete programming assignments without understanding the underlying concepts, and without developing critical thinking and problem-solving skills.

Prior research evaluating Codex and Copilot in the context of computing education has focused on programming problems, where the tools generate source code in response to natural language problem descriptions [4, 14]. How well these tools perform on other problem types is currently unexplored. Parsons problems are often used to help students develop skills needed for writing code [9].

\(^1\)https://github.com/features/copilot  
\(^2\)https://openai.com/blog/openai-codex/
A typical Parsons problem consists of a set of randomly ordered blocks representing lines of code that a student must order correctly to form a working solution. These have many benefits, including teaching programming concepts without the complexities of low-level syntax. In addition, it is reasonable to believe that they could be more resistant to solution by code generation tools. For example, models like Codex generate output in a left-to-right fashion by learning the probability distribution of the next token given the current context. Generating solutions that must include specified code fragments, such as the blocks in a Parsons problem could impact performance. Indeed, prior work has shown that Codex performs poorly when problems include restrictions, such as prohibiting the use of specific elements in the solution [14]. Parsons problems may be a promising alternative to traditional programming problems where the use of code generation tools is a concern.

We investigate the performance of Codex in solving Parsons problems. We extract candidate Parsons problems from the literature, present the randomly scrambled blocks as input to Codex, and then execute the rearranged solutions using test suites. We also explore how different prompting variations impact model performance. This contributes to the growing body of knowledge on code generation tools in computing education and may help educators make informed decisions about the use of these tools in the classroom. We are guided by the following research questions:

RQ1 How effectively does Codex generate solutions to Parsons problems?
RQ2 To what extent is the performance of Codex affected by different prompting variations?

2 RELATED WORK

In this section we summarize recent work on Parsons problems and large language models models in introductory programming.

2.1 Parsons Problems

It is well known that writing code from scratch can be overwhelming for novices. Parsons problems [33, 43] avoid some of the factors leading to this. These “mixed-up code problems” introduced by Parsons & Haden in 2006, require users to rearrange randomly ordered lines of a program into the correct order [34]. Since then, Parsons problems have received significant attention as an evidence-based pedagogical learning tool [7, 9]. Parsons problems have been studied in motivating students to practice [34]; as summative assessments [5]; as a bridge between code tracing and code writing activities [12]; to catalog common student errors when problems solving [21]; in helping students learn programming in block-based environments [43]; and even to scaffold the problem-solving process and encourage metacognition [17, 35].

Although many variations of Parsons problems have been explored, a common feature is that the rearrangeable blocks represent syntactically correct source code. As would be expected, variations which increase the possible solution space are more difficult for students to solve. Such variations include problems with distractors (code fragments not required for the solution) [5, 18, 32], problems that consist of a larger number of fragments [11, 21], and problems where students are required to indent code fragments as well as order them correctly (so-called two-dimensional problems) [23]. Ericson et al. used these observations to study adaptive Parsons problems [10] where intra-problem adaptation enabled problems to be simplified on student request by limiting the solution space through fragment combination or distractor removal. Inter-problem adaptation allowed for the automatic selection of subsequent problems based on current problem performance, an idea also studied by Kumar [25]. Cheng & Harrington studied the benefits of using Parsons problems on summative assessments with respect to grading consistency, finding that constraints imposed by the problems yielded faster grading times and higher grader confidence [2]. After observing students using clues in the syntax of the code lines to solve the problem without understanding why it is correct, Weinman et al. introduced faded Parsons problems where some syntax is missing or incomplete [39, 40]. Wu et al. have even applied Parsons for a horizontal problem space with regex [42].

2.2 LLMs and Introductory Programming

The recent emergence of large language models (LLMs) promise both opportunities and challenges for introductory programming [1] resulting in significant research activity in this area. This work has mainly focused on LLM performance in solving code writing exercises [4, 14, 15] and on using LLMs for creating or improving educational resources [6, 27, 29–31, 36].

Early work by Finnie-Ansley et al. [14] explored the performance of the OpenAI Codex model on typical introductory (CS1) programming exercises, including the well-studied Rainfall problem [16, 37]. They found that Codex performed better than the average student – it ultimately scored around 80% across two tests and ranked 17 out of 71 when its performance was compared with real students on the same exams. On the Rainfall problem, Codex was able to generate various correct solutions that differed in both algorithmic approach and source code length. One question that was left unanswered was whether Codex could solve more complex programming problems, e.g., those used in typical data structures and algorithms (CS2) courses. In a continuation study, it was found that Codex does outperform most students on these more complex problems [15]. In a recent study, Denny et al. [4] found that GitHub Copilot was able to solve about half of the introductory programming problems they explored. This increased when the user utilised “prompt engineering” – modifying the prompt given to the model with the aim of achieving better performance [28].

In addition to solving code writing questions, LLMs can be used for creating educational resources. Sarra et al. found that given an example exercise, Codex could create novel exercises and that the contents of the exercises could be influenced by providing keywords related to both programming concepts (e.g., loops, lists, etc.) and thematic concepts (e.g., basketball, cooking, etc.) [36]. In a similar vein, Denny et al. proposed “robosourcing”, where LLMs are used to scaffold learnersourcing [6]. In traditional learnersourcing, students create resources that can be used by other students, while in robosourcing, LLMs are used in the process, e.g., to provide initial artefacts that are then improved by other students.

LLMs have also been found to be capable of explaining source code in natural language. This can be helpful for novices, who might struggle to understand code. Prior work has used both Codex (which is optimized for source code) [36] and GPT-3 (which is optimized
for natural language) [29, 31] for producing code explanations. Similarly, recent work has explored using Codex to enhance and explain programming error messages [27].

As large language models continue to improve in tasks traditionally used for practice in introductory programming courses, a question that arises is whether there are any traditionally used problem types that can not be solved (easily) by these tools. The research gap we explore in the present work is to what extent large language models (specifically, Codex) can solve Parsons problems. As Parsons problems are common in introductory programming courses [9], it is important to understand whether they suffer from the same risk of students potentially over-relying on LLMs for solving them which already is the case for code writing problems.

3 METHODOLOGY

3.1 Data: A Body of Parsons Problems

To construct a dataset of problems for evaluating large language models, we searched the ACM Digital Library for papers on Parsons problems from the past decade. We iterated over the search results chronologically, identifying papers that provided concrete Parsons problem examples. We continued the search until we had a total of 10 papers with concrete examples [8, 10, 12, 13, 20, 22, 24, 26, 40, 43], which we used as a starting set.

We constrain our problem set to those having a unique solution regarding block ordering, i.e., we excluded problems for which there were multiple correct ways to order the blocks. This resulted in a total of 6 distinct problems – not all papers provided a usable problem with this restriction in place. In addition to the problem itself, we extracted the (textual) problem statement, or created a simple problem statement if one was not provided in the source article. For each Parsons problem, we created twenty distinct unindented variants with scrambled ordering. This involved dividing the solution code into lines, stripping any indentation, and then randomly scrambling the line order so that each new shuffled set of lines was distinct (and did not match the original solution). This led to a total of $6 \times 20 = 120$ scrambled (unsolved) Parsons problems.

3.2 Input: Constructing Prompts

Large language models are known to be sensitive to small alterations in prompts, so we constructed seven prompt variants, including also a no-prompt option. These prompts are (1) “Reorder and indent the lines”; (2) “Reorder and indent the lines if needed”; (3) “Reorder the lines”; (4) “Sequence the lines correctly”; (5) “Put the lines in order to solve the problem”; (6) “Produce the right answer”; and (7) no explicit prompt (herein denoted –).

Each prompt variant was evaluated with each scrambled Parsons problem, leading to a total of $7 \times 120 = 840$ different variant inputs. The structure of the inputs followed the structure in Listing 1, where the input contained four parts: (1) the problem statement; (2) the scrambled lines of code without indentation; (3) the prompt; and (4) an indicator highlighting the end of input, which we defined as the following comment: “# Solution with indentation” to indicate to the model that the code blocks should be indented correctly.

<table>
<thead>
<tr>
<th>Listing 1: Input structure</th>
</tr>
</thead>
<tbody>
<tr>
<td># Problem description</td>
</tr>
<tr>
<td>&lt;problem description goes here&gt;</td>
</tr>
<tr>
<td># Scrambled lines</td>
</tr>
<tr>
<td>&lt;scrambled code lines go here&gt;</td>
</tr>
<tr>
<td># &lt;prompt&gt;</td>
</tr>
<tr>
<td># Solution with indentation</td>
</tr>
</tbody>
</table>

Listing 2 illustrates a concrete example of a complete input provided to Codex.

<table>
<thead>
<tr>
<th>Listing 2: Complete example input</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Problem description</td>
</tr>
<tr>
<td>Finish the function below to return 'too low' if the guess is %less than the passed target, 'correct' if they are equal, and 'too high' if the guess is greater than the passed target. For example, check_guess(5, 7) returns 'too low', check_guess(7, 7) returns 'correct', and check_guess(9, 7) returns 'too high'.</td>
</tr>
<tr>
<td>#Scrambled lines</td>
</tr>
<tr>
<td>def check_guess(guess, target):</td>
</tr>
<tr>
<td>return 'correct'</td>
</tr>
<tr>
<td>def check_guess(guess, target):</td>
</tr>
<tr>
<td>return 'too low'</td>
</tr>
<tr>
<td>if guess == target:</td>
</tr>
<tr>
<td>return 'correct'</td>
</tr>
<tr>
<td>return 'too low'</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>return 'too high'</td>
</tr>
<tr>
<td># Reorder and indent the lines if needed</td>
</tr>
<tr>
<td># Solution with indentation</td>
</tr>
</tbody>
</table>

In this case the problem is taken from Figure 2 of the paper “Adaptive Parsons Problems as Active Learning Activities During Lecture” by Ericson & Haynes-Magyar (denoted as ‘Ericson2022figure2’ in our results tables). The prompt in this example is (2): “Reorder and indent the lines if needed”. The problem description was taken verbatim from the source article [8] and is quite detailed, including example inputs and outputs. This is also an example of a problem that includes paired distractor blocks – the original problem included three paired distractors, where the distractor was explicitly associated with a correct line of code. In each case the distractors are syntactically incorrect (i.e. missing brace or matching quote mark) and our scrambling process removed the explicit pairing.

3.3 Evaluation: Static Analysis

The 840 inputs were sent to Codex via the OpenAI API, with a setting for randomness of the outputs via a "temperature" parameter. Setting the temperature to 0 yields maximally deterministic outputs while larger values produce more random outputs. Prior research has found that a temperature of 0 works well when using Codex for non-code writing purposes [27]. Given this, we expected little variance in outputs for a given input, and therefore sent each input to Codex once (only), and used this output for analysis.

A correct solution to a Parsons problem is a valid reordering of the blocks, with each block correctly indented. As we selected Parsons problems from the literature where the correct ordering of the solution is unique, it was not necessary to execute the generated code. To evaluate the output we simply compared it to the original solution to the Parsons problem via standard string matching.

From Codex output for each input sent to the API, we (1) extracted the Codex solution (2) ran the static analysis (string matching) on the solution to assess its correctness, and (3) we evaluated...
whether Codex deviated from the assignment, e.g., by modifying the scrambled line set beyond indentation and reordering.

4 RESULTS

4.1 RQ1: Correctness of Generated Solutions

The results of the static analysis can be found in Tables 1, 2, and 3. Table 1 shows, for each priming and problem pair, the number of cases where the output generated by Codex did not match the correct solution. It can be seen that the performance of Codex in solving Parsons problems is very problem-dependent, and also somewhat dependent on the priming that was used. As an example, Codex was capable of correctly solving the Weinmann2021figure1 problem very often (77% of cases) regardless of the prompt (108 of 140 attempts were correct). On the other hand, Codex had difficulty solving the Ericson2022figure4 problem regardless of the priming, with only 29% success (29 of 108 attempts were correct).

Table 2 shows, again for each priming and problem pair, the number of cases where Codex had either modified or added lines to its solution that were not present in the Parsons problem. These cases cannot be considered valid solutions of a Parsons problem even if they are functionally correct as one requirement of Parsons problems is to reorder the existing code blocks, and typically students are not allowed to modify the blocks. It can be seen that Codex often does not modify the blocks, with two thirds of the problems not having any such cases. Even for the problem where Codex performed the worst in this regard (Ericson2022figure2) Codex only modified or added code in 8% (11 of 140) cases.

Finally, Table 3 shows the performance of Codex when indentation is ignored, i.e., a solution is considered correctly solved even if the indentation is wrong or missing. Unsurprisingly, comparing this to Table 1, it can be seen that the performance of Codex is better when indentation is ignored: over all priming-problem pairs, only 20% (167 of 840) of cases were incorrect when indentation is taken into account versus 49% of cases (415 of 840) when indentation is not taken into account.

Codex had the least difficulty solving Weinmann2021figure1 (Listing 3), with 77% of solutions correct. The prompt for this problem very often (77% of cases) regardless of the prompt (108 of 140 attempts were correct). On the other hand, Codex had difficulty solving the Ericson2022figure4 problem regardless of the priming, with only 29% success (29 of 108 attempts were correct).

5 DISCUSSION

5.1 RQ1: Can Codex Solve Parsons Problems?

Codex was able to correctly solve approximately half (51%) of the Parsons problems. This is lower than the performance of Codex on code writing problems where it can solve up to 80% of problems correctly [4, 14]. However, we did observe greater performance (80%) – similar to the performance on code writing tasks – when problems related to indentation are excluded. However, student performance on Parsons problems is often better than their performance on equivalent code writing problems [12], which should be considered when interpreting these results. Essentially, while Codex is better at solving code writing problems than the average student [14], it might still be worse at solving Parsons problems compared to students even though the performance in both tasks is around 80% when indentation is ignored for Parsons problems.

The performance of Codex was lower for Parsons problems than code writing problems, possibly limiting students’ over-reliance on large language models. Prior research also suggests that Parsons problems might reduce cognitive load compared to code writing
Table 1: Number of incorrect solutions generated by Codex for all Parsons problems and across all primings. A correct solution is one that is an exact string match (including indentation) of the original solution code for the Parsons problem. The number in each cell represents the number of incorrect Codex solutions out of the 20 total generated solution for each priming-problem pair.

<table>
<thead>
<tr>
<th>problem</th>
<th>Reorder and indent the lines</th>
<th>Reorder the lines</th>
<th>Reorder and indent the lines if needed</th>
<th>Sequence the lines correctly</th>
<th>Put the lines in order to solve the problem</th>
<th>Produce the right answer</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weinmann2021figure1</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure2</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>43</td>
</tr>
<tr>
<td>Ericson2022figure2</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>68</td>
</tr>
<tr>
<td>Hou2022figure2</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>15</td>
<td>12</td>
<td>15</td>
<td>80</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure4</td>
<td>12</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>93</td>
</tr>
<tr>
<td>Ericson2022figure4</td>
<td>14</td>
<td>13</td>
<td>16</td>
<td>15</td>
<td>13</td>
<td>14</td>
<td>99</td>
</tr>
<tr>
<td>total</td>
<td>44</td>
<td>45</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>69</td>
<td>415</td>
</tr>
</tbody>
</table>

Table 2: Number of incorrect solutions generated by Codex in which the failure was due to a modified line of code or an additional line of code that was not present in the original solution. The number in each cell represents the number of incorrect Codex solutions out of the 20 total generated solution for each priming-problem pair.

<table>
<thead>
<tr>
<th>problem</th>
<th>Reorder and indent the lines</th>
<th>Reorder the lines</th>
<th>Reorder and indent the lines if needed</th>
<th>Sequence the lines correctly</th>
<th>Put the lines in order to solve the problem</th>
<th>Produce the right answer</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weinmann2021figure1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ericson2022figure2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Hou2022figure2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ericson2022figure4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3: Number of incorrect solutions generated by Codex in which the failure was only due to one or more incorrectly indented lines of code. In other words, the generated solution did produce the correct reordering of the code blocks, but the indentation did not match the expected solution. The number in each cell represents the number of incorrect Codex solutions out of the 20 total generated solution for each priming-problem pair.

<table>
<thead>
<tr>
<th>problem</th>
<th>Reorder and indent the lines</th>
<th>Reorder the lines</th>
<th>Reorder and indent the lines if needed</th>
<th>Sequence the lines correctly</th>
<th>Put the lines in order to solve the problem</th>
<th>Produce the right answer</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weinmann2021figure1</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>31</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ericson2022figure2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Hou2022figure2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Haynes-Magyar2022figure4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>Ericson2022figure4</td>
<td>14</td>
<td>12</td>
<td>16</td>
<td>15</td>
<td>12</td>
<td>14</td>
<td>114</td>
</tr>
<tr>
<td>total</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>167</td>
</tr>
</tbody>
</table>
was found that Codex would very rarely modify the lines of code provided in the input or added new lines. This suggests that it can follow the orders given as input quite well.

There were substantial differences in Codex’s performance depending on the problem it was solving, ranging from 29% correct for the Ericson2022figure4 problem to 77% correct for the Weinmann2021figure1 problem. For the problems Codex struggled with and for those where it performed better, we found that similarly to students [18], Codex struggled more with Parsons problems with distractors (an example of such a problem is presented in Listing 4).

5.2 RQ2: Does Prompting Make a Difference?

Prior work exploring the accuracy of large language models such as GPT-3 and Codex has indicated that model outputs are often very sensitive to their inputs, and this has led to detailed discussions around effective prompt engineering strategies [28]. In our results, we do not observe very large differences in performance between the prompts that were used – in fact, the second most effective strategy was to not include an additional prompt (providing more explicit guidance) at all. A possible explanation for this is that the prompts we investigated formed a relatively small portion of the overall input to the model. The inputs already included the problem description and the blocks of code to be used in the solutions, as well as the end of input prompt “# Solution with indentation” – thus the individual prompts we manipulated represented a relatively small fraction of the input (in terms of character count).

5.3 Future Work

There are multiple interesting questions that arise from these results that can form the basis for future work. We found that Codex performed better when indentation was ignored. It would be interesting to study how Codex performs in solving Parsons problems in a language other than Python, as in many other languages indentation does not affect the functionality of programs.

In this study, we evaluated a set of Parsons problems which had unique solutions in terms of block ordering. Although we believe this to be a useful first step in evaluating the performance of LLMs in solving Parsons problems, such problems tend to be relatively simple to solve for students. For instance, the code shown in Listing 2 provides a number of syntactic clues that a student could use to solve the problem even without a complete understanding of the programming constructs (for example, the output should begin with ‘def’, and the ‘return’ statements should be nested inside conditionals such as ‘if’ and ‘else’). This is a potential weakness of Parsons problems that has been discussed in the literature. In their work exploring Faded Parsons problems, Weinman, Fox & Hearst raise this issue, noting that a sufficiently prepared student could solve such problems purely through syntactic clues [40]. Future work should explore how the performance of Codex changes as the difficulty of the Parsons problems increase, for example where problems contain more blocks and thus allow for a greater number of combinations, and multiple correct solutions.

6 THREATS TO VALIDITY

This study has multiple threats to validity. First, we used a “zero shot” setup where Codex was not given any examples of how to solve Parsons problems. While we could have reasonably achieved better results by doing so, we believe our approach more accurately represents how students would utilize the tool. Second, we used the minimum “temperature” of 0 for all attempts. While a higher temperature setting could have resulted in more creative solutions, we chose the minimum due to the very nature of Parsons problems. A solution to a Parsons problem is ultimately one that is exactly like the prompt, only in the correct order. Third, we obviously biased the system in both problem selection and priming choices. To mitigate problem selection, we chose Parsons problems from the literature. To mitigate priming bias, we attempted to phrase primers in multiple ways as a student might.

7 CONCLUSION

In this work, we presented a study on how well large language models, specifically Codex, can solve Parsons problems using small variations in prompts. We found that Codex was able to solve Parsons problems in about 50% of the cases, and that performance was somewhat dependent on the problem. If problems with indentation in Codex-produced solutions were ignored, performance was better (80% of the solutions were correct). The latter performance is similar to previously reported performance on traditional introductory programming ‘write code’ problems, where Codex has been demonstrated to be capable of solving up to 80% of problems correctly. Regarding prompt variations, we noticed a clear difference in performance for different variations. Notably, having “Reorder the lines” in the prompt produced roughly 1/3 fewer incorrect solutions compared to having “Produce the right answer” and 1/6 fewer incorrect solutions compared to having “Put the lines in order to solve the problem”.

The overarching motivation for our study was to explore the extent to which Codex could be used to support students and also examine how slight modifications in prompts can affect its performance. Indeed, Parsons problems could be seen as an instance of a situation, where a student is stuck and needs help. Further studies could explore Parsons problem-like inputs, but where some of the necessary lines are missing or contain invalid content. In addition to reordering content and potentially avoiding distractors, this would also require fixing bugs, which Codex has been previously found to perform well at for specific types of bugs [27].

Contrasting our results with prior work that has shown that Codex can solve introductory programming code writing problems [14], our results suggest that Parsons problems are not as easy for large language models to solve as code writing problems. Thus, they could be considered to be more reliable for assessing students’ performance, in a setting where students could use Codex (or similar). We acknowledge that our work focused on only Parsons problems with a unique solution, and thus future work is required to examine what aspects of Parsons problems might affect the performance of large language models in solving them.