My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises

James Finnie-Ansley
The University of Auckland
Auckland, New Zealand
james.finnie-ansley@auckland.ac.nz

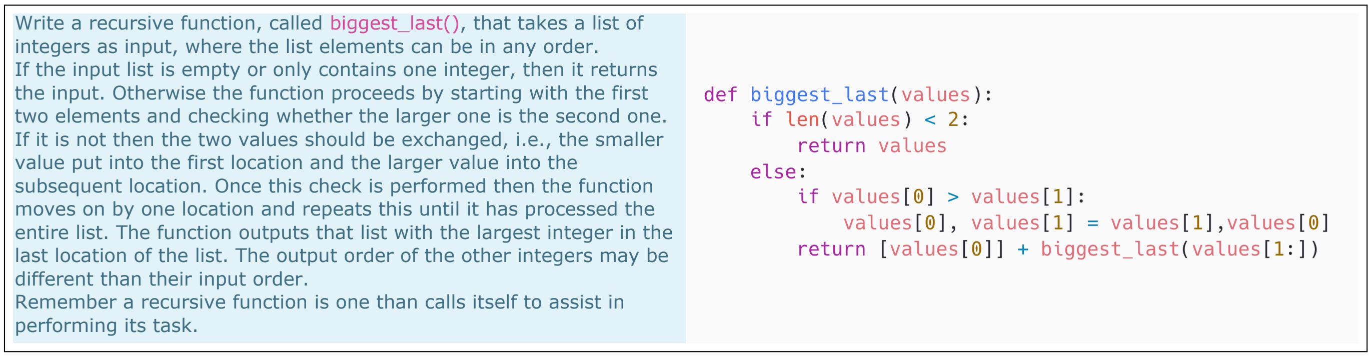
Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Andrew Luxton-Reilly
The University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

Eddie Antonio Santos
University College Dublin
Dublin, Ireland
eddie.santos@ucdconnect.ie

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

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



Write a recursive function, called biggest_last(), that takes a list of integers as input, where the list elements can be in any order.
If the input list is empty or only contains one integer, then it returns the input. Otherwise the function proceeds by starting with the first two elements and checking whether the larger one is the second one.
If it is not then the two values should be exchanged, i.e., the smaller value put into the first location and the larger value into the subsequent location. Once this check is performed then the function moves on by one location and repeats this until it has processed the entire list. The function outputs that list with the largest integer in the last location of the list. The output order of the other integers may be different than their input order.
Remember a recursive function is one than calls itself to assist in performing its task.

```python
def biggest_last(values):
    if len(values) < 2:
        return values
    else:
        if values[0] > values[1]:
            values[0], values[1] = values[1],values[0]
        return [values[0]] + biggest_last(values[1:])
```

Figure 1: A lengthy question given in a CS2 exam (left) and code generated by OpenAI Codex to answer the question (right)

ABSTRACT

The introduction of OpenAI Codex sparked a surge of interest in the impact of generative AI models on computing education practices. Codex is also the underlying model for GitHub Copilot, a plugin which makes AI-generated code accessible to students through auto-completion in popular code editors. Research in this area, particularly on the educational implications, is nascent and has focused almost exclusively on introductory programming (or CS1) questions. Very recent work has shown that Codex performs considerably better on typical CS1 exam questions than most students. It is not clear, however, what Codex's limits are with regard to more complex programming assignments and exams. In this paper, we present results detailing how Codex performs on more advanced CS2 (data structures and algorithms) exam questions taken from past exams. We compare these results to those of students who took the same exams under normal conditions, demonstrating that Codex outscores most students. We consider the implications of such tools for the future of undergraduate computing education.

ACE '23, January 30-February 3, 2023, Melbourne, VIC, Australia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9941-8/23/01.
https://doi.org/10.1145/3576123.3576134

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **Computer science education**; **CS1**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

academic integrity; AI; AlphaCode; algorithms; artificial intelligence; code generation; Codex; copilot; CS1; CS2; data structures; deep learning; DeepMind; introductory programming; GitHub; GPT-3; machine learning; neural networks; novice programming; OpenAI

ACM Reference Format:
James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Australasian Computing Education Conference (ACE '23), January 30-February 3, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3576123.3576134

1 INTRODUCTION

In 2021, OpenAI unveiled Codex [5], an AI model capable of generating source code from plain English prompts. Codex powers GitHub Copilot [12], an extension available within several code editors and IDEs which has resulted in calls for "rethinking computer science education".[1] On June 21, 2021, Copilot was made available

[1]theregister.com/2022/10/20/ai_programming_tools_mean_rethinking

free to any student with a verified GitHub Education account.[2] It was then made availabe for free to teachers on September 8, as it became evident that Codex and Copilot will impact curriculum, as well as present emerging educational opportunities—as early research demonstrated that Codex is effective at solving many introductory programming assignments [10] and that it can generate explanations of code examples [23].[3]

Codex's ability to understand plain English and produce code is impressive [20]. It is based on the third generation of OpenAI's Generative Pre-trained Transformer (GPT-3), one of the most advanced natural language models that is widely-available as of this writing, and is trained on "tens of millions of public repositories" [5, 25].

Codex has a number of features that allow users to interact with and generate source code. With plain English prompts as input, it will generate code in several programming languages including Python, C, JavaScript, Go, Perl, PHP, Ruby, Swift and TypeScript, and Shell.[4] It will also translate code from one programming language to another and provide English-language descriptions of the functionality of code provided as input including its computational complexity. It can also generate code that calls APIs allowing it to access information in publicly available datasets—for instance getting the air quality index for cities based on ZIP codes. Codex is available via the OpenAI API[5] and via GitHub Copilot[6].

1.1 Motivating example & research questions

Learning to program has a long history of known difficulties [4, 19] and the challenges students face have been discussed from many angles [3, 18]. Just a few of the known barriers are dealing with programming tools and environments [15], programming error messages [7], program comprehension [24], and how students approach problem solving [8, 9]. In fact, much of the research on teaching programming focuses on challenges that students face [4].

Figure 1 shows an example of a CS2 problem description written in plain English (left). Despite the description being lengthy and containing many specific details, Codex is capable of generating a correct solution (right).

Although the details and emphasis of CS2 courses may vary [13], the main topics of programming methodologies, recursion, searching, sorting, and data structures have remained a staple of CS2 courses for more than 30 years [16]. CS2 courses typically build on the basic programming constructs taught in CS1 by covering common structures used to store data (e.g., linear structures such as lists, stacks, and queues; and non-linear structures such as sets, dictionaries, and trees) and the implementation of those data structures. It also typically covers the algorithms that use the data structures, including searching and sorting algorithms. Although CS2 may introduce more elements of theory (e.g., analysis of running time and efficiency trade-offs), it typically continues to develop programming skills with problems that involve implementation of data structures [26].

Codex has significant implications for educational practices involving programming. Prior work demonstrated that Codex was able to outperform the majority of CS1 students when given the same English language problem descriptions [10] used in tests and exams. However, the questions used in CS2 courses are typically more advanced and may not be so easily solved by models like Codex. In this paper, we aim to quantify how well Codex performs on more complex programming problems used in CS2 exams. We are motivated by the following research questions:

RQ1: How does Codex perform on CS2 assessments compared with students?
RQ2: How does Codex perform on CS2 assessments compared with CS1 assessments?
RQ3: What question characteristics appear to influence the performance of Codex?

Codex and other tools like it (based on large language models) are still nascent. It is therefore unclear how this emerging technology will impact the computing education classroom. Answering our research questions is important because computing educators need to know what tools like Codex are capable of doing, in what ways they excel, and in what ways they fail. Exposing this will help shape the ongoing conversation of how to adapt to the new capabilities at the fingertips of our students (and their educators).

This paper is organized as follows: Section 2 provides the background of Codex and related AI-powered code generation tools. We then evaluate the performance of Codex on a suite of programming problems taken from CS1 and CS2 tests with Section 3 describing our method and Section 4 reporting the performance of the CS2 problems in relation to student performance and assessing the overall performance of Codex on different questions. We then discuss the implications of our findings in Sections 5 onward.

2 RELATED WORK

One hope of modern artificial intelligence is that it could provide dynamic educational support at massive scale never before thought possible [2]. Modern systems like Codex seem poised to move significantly towards this goal. For instance, Finnie-Ansley et al. demonstrated that Codex can solve typical CS1 problems and perform amongst the top quartile of (real) students on (real) CS1 exams [10]. However, it is currently unclear how such technologies will be adopted by students and by teachers in computing classrooms, and whether the opportunities presented by code generation models will outweigh the challenges relating to their inappropriate use.

2.1 AI-powered code generation

Early investigations of GPT-3 revealed that it could generate simple programs from Python docstrings, despite GPT-3 being a natural language model not trained for code generation [5]. This observation led to the creation of Codex, discussed in Section 1.

Other AI code generation tools exist and have gained rapid ground in very recent years. Tabnine is an "AI Assistant for Software Developers"[7] that auto-completes lines of code in several languages and can be integrated into popular IDEs. It can be trained on in-house repositories to learn the code patterns of specific teams. However, Tabnine can only generate fully correct code for less than 8% of unseen programming tasks, given 100 samples [5]. DeepMind

[2] github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers
[3] github.blog/2022-09-08-github-copilot-now-available-for-teachers
[4] openai.com/blog/openai-codex
[5] beta.openai.com
[6] copilot.github.com

[7] www.tabnine.com

AlphaCode is a transformer-based model that was trained on code from competitive programming competitions [17]. As a result, AlphaCode can generate code to solve complex problems entirely from a plain English description, ranking in the top 54% when entered into actual competitive programming competitions. Amazon CodeWhisperer[8] is yet another "machine learning-powered" IDE plugin that is intended to automatically generate code, including code that calls APIs.

2.2 Effectiveness of AI-generated code

There have been many recent attempts to assess the effectiveness of AI-powered code generation. Most studies focus on professional software developers [5, 6, 27], while few study its implication in the classroom [10].

Dakhel et al. explored Copilot's ability to generate code for various introductory algorithms, as well as five introductory Python programming problems [6]. They found that Copilot was performant in generating correct and often quite optimal code for common algorithms such as insertion sort, breadth-first search, and depth-first search. In cases where Copilot generated incorrect code, they found that these were easier to fix than typical code submitted by novices.

Vaithilingam et al. compared the experience of programmers (predominantly students from undergraduate through PhD) using Copilot and typical IntelliSense support in standard IDEs [27]. They found that most users preferred Copilot, but that it did not necessarily reduce task completion time or increase success rate. Users tended to over-rely on the code produced by Copilot and some users struggled to understand and debug the auto-generated code.

Finnie-Ansley et al. evaluated the performance of OpenAI's Codex on a private repository of CS1 test questions and found that it solved roughly half of the questions on the very first attempt [10]. The model ultimately scored around 80% across two tests when multiple attempts (with resubmission penalties consistent with the course grading scheme) were taken into account, and ranked 17 out of 71 when its performance was compared with students who were enrolled in the course.

However, AI code generation is far from perfect. Nguyen and Nadi evaluated the performance of Copilot on 33 LeetCode questions, with varying difficulty, observing correctness rates of between 27% and 57% across four languages [21].

Recently, code generation models have been applied to the task of generating learning resources for students. Sarsa et al. explored the potential for Codex to generate novel programming exercises and explanations of code [23]. Using appropriate priming examples as input, they found that exercises could be created that targeted specific programming concepts and the contextual themes. In most cases exercises were sensible, novel, and included matching sample solutions.

3 METHOD

To answer RQ1 and RQ2 we assess the correctness of solutions generated by Codex in response to problems from CS1 and CS2 courses. To answer RQ3 we compare characteristics of the questions in which Codex performed very well with characteristics of the questions in which Codex performed poorly.

3.1 Comparing Codex with students in CS2

RQ1 seeks to compare the performance of Codex on CS2 questions to the performance of students. We utilize 26 programming questions that were used as summative assessments on two invigilated (proctored) lab-based tests conducted in a CS2 programming course at the University of Auckland in 2019 (14 questions from the first test and 12 from the second test). In total, there were 264 students enrolled in the course.

The CS2 course provides a review of the Python programming language, and covers efficient ways to organize and manipulate data, including sorting and searching algorithms, as well as writing software that uses and implements common abstract data types such as lists, stacks, queues, dictionaries, and binary trees.

The lab-based tests were designed to be two hours in duration. Each test question consisted of a problem statement and at least one example test case illustrating successful program execution. Students completed the test questions on machines with standard software images, and used the IDLE environment to develop their programs. Figure 2 shows and example of how test questions appeared to students.

The completed programs were submitted to an automated testing system (CodeRunner) which provided immediate feedback to students who were shown a subset of the test cases used to determine answer correctness.

Students were permitted to resubmit their code multiple times, but repeated incorrect submissions accumulated a penalty up to a maximum penalty of 50%, as shown in Figure 2. Marks for each question were assigned on an all-or-nothing basis: marks were only awarded for a question if the submitted code successfully passed all tests. Code that did not successfully pass all tests received 0 marks for that question.

As input to Codex, the problem prompt and example test cases (as doctests) were given as the `instruction`, and any function headers or pre-loaded code included in the answer window was given as the `input` to the edit model.[9] The edit model was chosen as many problems provide a function header as a starting point or ask students to modify existing code provided in the answer window. Specifically, we used OpenAI's `code-davinci-edit-001` model which takes a prompt "instruction" and starter code "input" for each question. Aside from using the edit model, Codex was configured the same as it was for a similar recent study (see [10]).

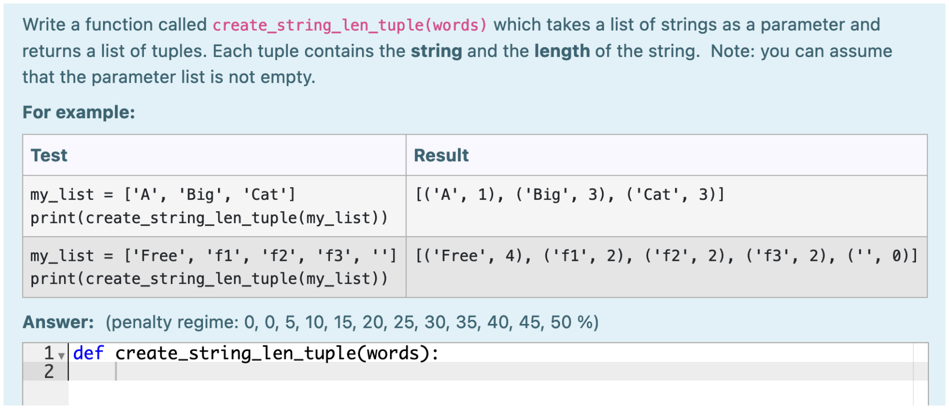
In the evaluations reported in this paper, we provide as input to Codex the problem statements exactly as they were presented to students. We did not perform any prompt engineering—instead, we are simulating students copy-pasting the problem description verbatim into a downstream application of Codex, such as GitHub Copilot.

3.2 Codex performance in CS1 vs. CS2

RQ2 seeks to compare the performance of Codex on CS1 versus CS2 style questions. We compare the performance of Codex on the CS2 problems discussed above with performance on CS1 problems.

[8]https://aws.amazon.com/codewhisperer/

[9]https://beta.openai.com/docs/api-reference/edits/create


Write a function called `create_string_len_tuple(words)` which takes a list of strings as a parameter and returns a list of tuples. Each tuple contains the **string** and the **length** of the string. Note: you can assume that the parameter list is not empty.

For example:

Test	Result
`my_list = ['A', 'Big', 'Cat']` `print(create_string_len_tuple(my_list))`	`[('A', 1), ('Big', 3), ('Cat', 3)]`
`my_list = ['Free', 'f1', 'f2', 'f3', '']` `print(create_string_len_tuple(my_list))`	`[('Free', 4), ('f1', 2), ('f2', 2), ('f3', 2), ('', 0)]`

Answer: (penalty regime: 0, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 %)

```
def create_string_len_tuple(words):
    
```

Figure 2: Presentation of Question 1 from CS2 Test 1 within the web-based examination tool as seen by students

We collated 28 CS1-type problems that were used as summative assessments on two un-invigilated lab-based tests from a CS1 course at our institution in 2020 (14 questions each).

The CS1 course covers the basics of programming such as variables, arithmetic, conditional branches, loops, reading and writing to files, and functions. Students submitted the lab-based tests in CS1 using the same assessment system for CS2 discussed above. The tests were designed to be 2 hours in duration and were time-limited, but were not invigilated. In this study we do not consider student performance in the CS1 tests due to difficulties interpreting student performance in an un-invigilated test environment. However, the test questions are representative of typical CS1 problems used in this course, and provide an appropriate problem set to compare with CS2 when considering the performance of Codex.

3.3 Scoring Codex

In our previous study [10], we manually generated one run of responses to question prompts that were copied into CodeRunner until we got a correct response. While this gave results that were scored in the same way as students, the single run was susceptible to the random hit-or-miss nature of Codex responses. As the Codex model is non-deterministic, there is variability in the output it generates for a given input. Scoring the performance of Codex using a single set of outputs is therefore unlikely to lead to results that are replicable. To ameliorate this issue in this study, for each question we generate 50 Codex responses as a representative sample to assess the performance of Codex for that question (Chen et al. [5] used 100). In total we generated 2700 code responses.

We used two general metrics to assess the performance of Codex:

Chance-x This is the probability of generating a fully-correct solution within x attempts (inclusive). For example, chance-1 is the probability of getting a fully correct solution in 1 attempt, and chance-5 is the probability of generating a fully correct solution within 5 attempts.

Simulated This is the expected score (out of 1) that Codex would attain on a question if it were being marked in the same way as students (with the penalty scheme described in Section 3) capped at a maximum of 5 attempts per question.

The *simulated* metric allows us to compare Codex to students as it is effectively grading Codex with the same penalties as students. In addition, it is capped at a small number of attempts, consistent with how a student might use Codex in a time-limited test. The *chance-x* metrics are intended to give an overview of the general "power" of Codex if it were being used by a student. It is unlikely that a student using a tool such as Codex to generate answers on tests or assignments would only generate one solution if they were able to test the code it generates; the chance-x metrics, while simple, are intended to show how likely a student is to generate a fully-correct solution within a finite number of repeated attempts.

4 RESULTS

4.1 RQ1: Comparing Codex with students in CS2

Our first research question asks how Codex performs on CS2 assessments compared with students; specifically, we compared the performance of Codex on CS2 questions with students answering the same questions under invigilated test conditions. Prior work that explored the same question at the CS1 level revealed that Codex performed well, scoring approximately 80% on two tests, and placing 17 out of 71 when ranked alongside students [10].

Table 1 contains the per-question scores of Codex and the mean score of students as well as the mean scores for each test respectively. It also lists the overall topic/subject of each question. We compare the "Simulated" score of Codex, which uses the same penalty scheme as students were graded with, with the "Student" scores. We use an asterisk to denote which of the "Simulated" and "Student" scores is greater for each question.

For both tests, the mean simulated score of Codex is higher than the mean student score, and on individual questions Codex exhibits equal or better performance compared to the mean student score for 77% of questions (20 of 26).

Figure 3 plots the Test 1 and Test 2 scores (scaled to a maximum of 100) of 264 students enrolled in the CS2 course in 2019 who completed both tests. For comparison, the performance of the responses generated by Codex is marked with an orange '×'. Averaging both Test 1 and Test 2 performance, Codex's score is in position 66 when ranked alongside the 264 students' scores, placing it just within the top quartile of class performance.

Test-Question	Question Tags	Chance-1	Chance-3	Chance-5	Simulated	Student
1-1	Algebra	0.56	0.91	0.98	*0.98	0.80
1-2	IO, Data Sanitisation	0.24	0.56	0.75	*0.87	0.84
1-3	Algebra, Finite Series	0.68	0.97	1.00	*0.99	0.82
1-4	File IO, String Manipulation	0.70	0.97	1.00	*0.99	0.62
1-5	String Manipulation	0.30	0.66	0.83	*0.91	0.76
1-6	Mapping/Aggregation	0.26	0.59	0.78	*0.89	0.60
1-7	Algebra, Finite Series	0.36	0.74	0.89	*0.94	0.37
1-8	String Manipulation	0.34	0.71	0.87	*0.93	0.59
1-9	Classes/OOP	0.76	0.99	1.00	*1.00	0.61
1-10	String Manipulation	0.34	0.71	0.87	*0.93	0.42
1-11	Tree Search	0.04	0.12	0.18	*0.31	0.20
1-12	IO, Rainfall	0.14	0.36	0.53	*0.71	0.32
1-13	Insertion Sort, String Formatting	0.00	0.00	0.00	0.00	*0.08
1-14	Classes/OOP	0.02	0.06	0.10	*0.17	0.10
Avg.	–	**0.34**	**0.60**	**0.70**	***0.76**	**0.51**
2-1	Discrete Mathematics	0.66	0.96	1.00	*0.99	0.98
2-2	String Manipulation	0.54	0.90	0.98	=0.96	=0.96
2-3	Filtering	0.48	0.86	0.96	*0.94	0.91
2-4	List Manipulation	0.06	0.17	0.27	0.25	*0.31
2-5	Rainfall Variant	0.08	0.22	0.34	0.32	*0.54
2-6	Classes/OOP	0.08	0.22	0.34	0.32	*0.47
2-7	Classes/OOP	0.12	0.32	0.47	*0.45	0.18
2-8	Stacks/Queues	0.26	0.59	0.78	*0.75	0.38
2-9	Classes/OOP	0.20	0.49	0.67	*0.64	0.13
2-10	Binary Heaps	0.40	0.78	0.92	*0.90	0.23
2-11	Hashing	0.00	0.00	0.00	0.00	*0.15
2-12	Binary Search Trees	0.02	0.06	0.10	0.09	*0.11
Avg.	–	**0.24**	**0.46**	**0.57**	***0.55**	**0.45**

Table 1: CS2 question scores of grading methods rounded to two decimal places; the higher score for each question appears with an asterisk (*)

4.2 RQ2: Codex performance in CS1 vs. CS2

Our second research question asks how Codex performs on CS2 assessments compared with CS1 assessments. To assess this, we applied the simulated score calculation described in section 3.3 to the CS1 questions. As established previously, Codex has been shown to perform well on CS1 questions [10]. When comparing CS1 and CS2 performance, it would be reasonable to expect Codex to perform better on CS1 questions than on CS2 questions, given their lower overall complexity.

Figure 4 plots a kernel density estimation of the simulated per-question scores (the "Simulated" column in Table 1) for each of the two CS1 tests and the two CS2 tests. The x-axis indicates the simulated per-question test score, and the y-axis indicates the "density" of questions with that score. A higher peak means that more questions achieved a simulated score in that relative region.

The distributions for both CS1 tests, as well as well as the first CS2 test are heavily bimodal: for these tests, Codex usually achieves a high average score (accounting for the larger, higher mode), or it fails to consistently answer a question correctly (accounting for the smaller, lower mode). In other words, Codex's performance on these tests is hit-or-miss; Codex either produces a good solution (more likely), or will produce very low scoring solutions (less likely). However, the per-question score distribution that Codex achieved on CS2, test 2—while still bimodal—is far flatter than all other distributions, meaning that, for CS2, Codex produces responses with a wider variety of correctness in CS2.

4.3 RQ3: Differences in questions by scores

RQ3 seeks to explore some of the characteristics that tend to be common to questions where Codex performs well and to those where it does not perform well. We examined the CS1 and CS2 questions where the simulated score was above and below the average simulated score of 72% and report in this section some observations of elements common to these questions. In total there were 35 questions with a simulated score that was above average and 19 questions that got a simulated score that was below average.

Between these sets of questions we make some general observations. There is a striking difference between the number of characters included in each prompt given to Codex with the best performing questions having a mean of 742 characters per question and

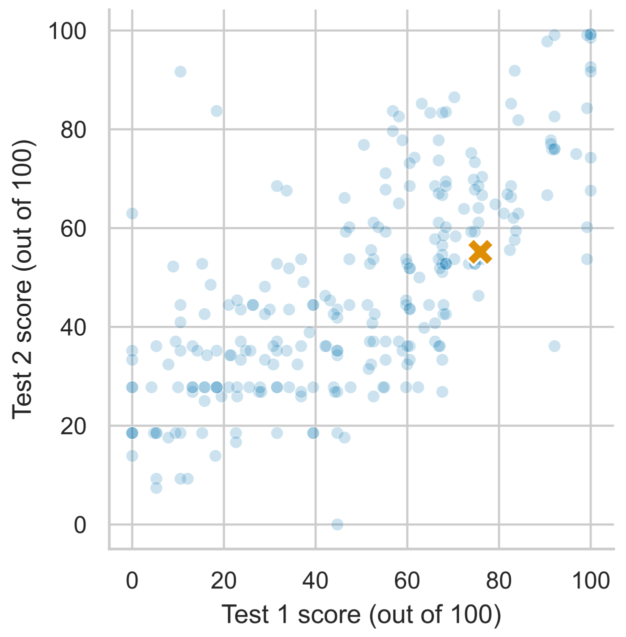


Figure 3: Student scores on two invigilated tests for CS2 with the performance of Codex plotted as an orange '×'

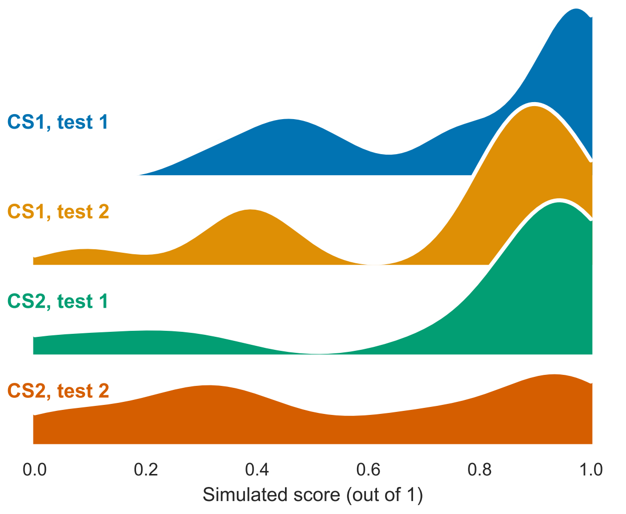


Figure 4: Kernel density estimation of the distribution of Codex's per-question scores, grouped by test

the worst performing questions having a mean of 1443 characters per question—nearly double that of the best performing questions. Although some of this may be explained by the fact that more complex questions may require longer prompts, it also aligns with observations of Chen et al. that performance of the Codex model tends to decrease exponentially as the number of basic building blocks in a question increases [5]. Figure 5 shows Codex's question

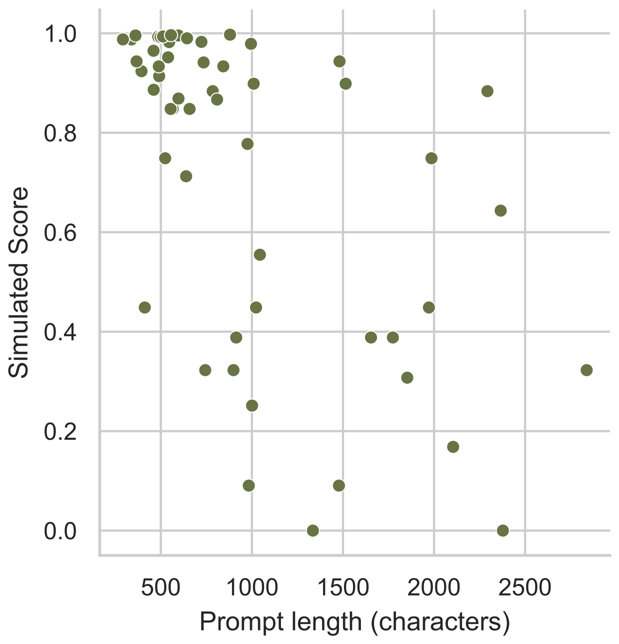


Figure 5: Simulated score by prompt character length

scores by prompt length. Simulated score has a moderate negative correlation with prompt length ($r = -0.57$).

A greater proportion of the worst performing questions also contained additional code that needed to be edited or used (e.g., helper functions) with 5 of the 19 worst performing questions including such code as input (26%) versus 2 of the 35 best performing questions (6%). We also note some general differences in the nature of the problems between the two sets of questions.

The best performing questions tend to be posing simple problems that require the application of standard algorithmic patterns [11] (e.g., filtering, mapping, etc.) or computing common mathematical operations such as multiplying numbers, computing prime factorizations, or the magnitude of a line in 2D space. Below is one of the best performing questions taken from one of the CS1 tests:

> *Complete the*
>
> `get_list_of_four_letter_words(words_list)`
>
> *function which is passed a list of strings as a parameter. The function returns a new list of all the words from the parameter list which are four characters in length. All the words in the returned list should be in lowercase characters.*
>
> *Notes:*
>
> - *You MUST use the* `append()` *method to add elements to the end of the list.*
> - *If the parameter list is empty or contains no words which are four characters in length, the function returns an empty list.*
>
> *For example:*

```
>>> words = get_list_of_four_letter_words(
    ['into', 'elephant', 'room', 'the'] )
>>> print(words)
['into', 'room']
```

The worst performing questions tend to be those with implicit edge cases (e.g., not explicitly stating words might contain uppercase letters), questions that operate on complex data (nested structures, 2D lists, etc), questions that need specific output formatting, or questions that give a non-itemized specification of a class API with several methods that need to be implemented. Below is one of the lowest scoring questions taken from one of the CS2 tests:

> *The following code implements the Insertion Sort algorithm and maintains the sorted part of the list at the start.*
> *[... Insertion Sort code provided here ...]*
> *Modify the code so that the sorted part of the list is maintained at the end of the list.*
>
> *Additionally, modify the code so that it prints out the list before each element is added to the sorted sublist. The output should contain the number of elements in the sorted sublist (size), and should print the sublist that is still unsorted, and the sublist that is sorted.*
>
> *For example:*

```
>>> insertion_sort([4, 1, 3, 5, 2])
1 : unsorted - [4, 1, 3, 5] sorted - [2]
2 : unsorted - [4, 1, 3] sorted - [2, 5]
3 : unsorted - [4, 1] sorted - [2, 3, 5]
4 : unsorted - [4] sorted - [1, 2, 3, 5]
5 : unsorted - [] sorted - [1, 2, 3, 4, 5]
```

In this question students are asked to edit code that is provided to them. The goal is to modify a sorting algorithm by changing its intermediate behavior (which portion of the list is sorted as the algorithm progresses) and to print intermediate states. The additional input code was provided as the "input" field of the edit API with the question prompt given as the "instruction"[10]. This question was the lowest scoring by both students and Codex. Codex was unable to produce a single fully-correct response from the 50 generated and students attaining a mean score of 0.08 out of 1.

5 DISCUSSION

The results of this study and our previous study [10] demonstrate that Codex can perform better than most students on code writing questions from both CS1 and CS2 tests, scoring in the top quartile of test marks when compared with students in both cases.

Overall, our results show that Codex performs better than most students on code writing questions at the CS2 level, mirroring the findings of Finnie-Ansley et al. [10] that focused on CS1 questions. In both cases, the performance of Codex falls within the top quartile of test marks when compared to students. Although Codex does appear to perform better in CS1 compared with CS2, it is still capable of completing most CS2 tasks assessed in these exams. Since Copilot is available as a free extension to commonly used IDEs, we can assume that most students will have access to an AI-generated solution that is likely better than their own (unassisted) solution.

[10]https://beta.openai.com/docs/api-reference/edits/create

Our analysis of question characteristics suggests that Codex performs well on tasks that start with a 'blank slate' and have explicit, well-defined requirements. Those with much longer descriptions are less able to be completed by Codex. Tasks that require modification of existing code, or in which there are implicit requirements to handle edge cases typically have poorer performance. The characteristics that make tasks more challenging for Codex may also make tasks more challenging for students—ambiguous or implicit requirements, difficult edge cases, wordy questions, and the need to understand existing code and make modifications.

Our results indicate that Codex is very good at solving semi-complex code reading and writing problems. It falters when there are odd edge cases or the requirements are somewhat esoteric. This is probably because these data were not in its training set and it struggles to extrapolate a solution. Since Codex was trained on code from GitHub, it is also important to recognize what else was possibly not in (or less well-represented in) the training set, including multiple choice questions, card sorting exercises, Parsons problems, programming error message diagnosis, reverse code-tracing, and more. These may be more out of Codex's immediate reach, and we propose that these be explored further by the research community. If Codex can handle current assignments and exams as suggested by our results (including [10]) educators should reach beyond typical questions that can still adequately measure student learning. It appears that students are likely to be over-reliant on AI-generated code and often struggle to properly utilize it [27]. Acting on this could yield approaches that teach students how to effectively use tools like Codex - while still enhancing learning.

5.1 Speculation

We have no clear signals to indicate how AI code generation technology will develop in the future, or how widely it will be adopted by industry professionals. However, it seems fairly safe (likely obvious) to predict that this technology will become faster, more accessible, and more accurate [22]. Exploratory work by Vaithilingam et al. [27] reports that users of Copilot prefer it to the Intellisense support provided in Visual Studio Code. This user preference suggests that students will likely adopt the technology, particularly if it helps them to complete assessed work.

The existence of AI code generation tools complicates the delivery of computing education as students will have ready access to uniquely generated solutions that are frequently correct, but not curated (i.e., the solution may be flawed, or use programming constructs or idioms that are inconsistent with the course instruction). The temptation to use such a technology for graded assignments will be high, and may impact student behavior in negative ways.

Given this, our educational effort is perhaps best directed at better supporting students to understand the code to which they are exposed. This may emphasize reading over writing, which is consistent with the approach advocated by Xie et al. [28].

We have limited understanding of how these technologies will impact student behavior or how they might impact computing education practices. Regardless, we believe there *will* be an impact, and our community urgently needs to understand how to best mitigate the drawbacks and to leverage the potential benefits.

6 LIMITATIONS

Since our students have continued access to their test questions after a test is completed, it is possible that some students may have uploaded the test problems to GitHub or other online repositories. In this case there may be exemplars present in the data that Codex was trained on. The questions used in this study are drawn from a single institution, and both CS1 and CS2 courses that contribute the data use a web-based automated assessment tool that includes test cases. The nature of the course and the tool used, may mean that these assessments are not representative of questions used in other institutions. However, we note that both courses use the online Runestone textbooks [1] and cover standard CS1 and CS2 content aligned with the ACM Curriculum [14].

It should be noted that our analysis focuses on the artifacts produced—essentially, this is aligned with the behaviorist paradigm and should not be treated as a measure of the "intelligence" of the AI-generative models under cognitive perspectives.

7 CONCLUSIONS

We find that Codex is able to solve most CS2 questions, performing similarly to students in the top quartile of the class that answered the same questions we provided to Codex. We find evidence that Codex may perform better on questions that are more precisely defined, succinctly written, have fewer edge cases, and do not require adapting existing code. This work confirms that Codex is well capable beyond the complexity of CS1 problems. It is unknown at what point the complexity of questions will markedly impact the performance of Codex (in the programming education domain). Further, how educators should adapt to this new technology that is freely available to our students, remains an open question. More work is needed in this rapidly emerging so that educators can to best adapt their classroom practices in ways that continue to benefit student learning.

REFERENCES

[1] Runestone Academy. 2022. Foundations of Python Programming. https://runestone.academy/ns/books/published/fopp/index.html

[2] Brett Becker. 2017. Artificial Intelligence in Education: What Is It, Where Is It Now, Where Is It Going. *Ireland's Yearbook of Education* 2018 (2017), 42–46.

[3] Brett A. Becker. 2021. What Does Saying That 'Programming is Hard' Really Say, and About Whom? *Commun. ACM* 64, 8 (jul 2021), 27–29. https://doi.org/10.1145/3469115

[4] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, NY, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374. (2021).

[6] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, et al. 2022. GitHub Copilot AI Pair Programmer: Asset or Liability? https://doi.org/10.48550/arXiv.2206.15331

[7] Paul Denny, James Prather, and Brett A. Becker. 2020. Error Message Readability and Novice Debugging Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. ACM, NY, NY, USA, 480–486. https://doi.org/10.1145/3341525.3387384

[8] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, et al. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling '19)*. ACM, NY, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3364510.3366170

[9] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices *(ITiCSE '20)*. ACM, NY, NY, USA, 159–165. https://doi.org/10.1145/3341525.3387404

[10] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference (ACE '22)*. ACM, Online, 10–19. https://doi.org/10.1145/3511861.3511863

[11] James Finnie-Ansley, Paul Denny, and Andrew Luxton-Reilly. 2021. A Semblance of Similarity: Student Categorisation of Simple Algorithmic Problem Statements. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*. ACM, NY, NY, USA, 198–212. https://doi.org/10.1145/3446871.3469745

[12] GitHub. 2021. GitHub Copilot - Your AI Pair Programmer. https://github.com/features/copilot/. (Accessed July 21, 2022).

[13] Matthew Hertz. 2010. What Do "CS1" and "CS2" Mean? Investigating Differences in the Early Courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, NY, NY, USA, 199–203. https://doi.org/10.1145/1734263.1734335

[14] ACM Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, NY, NY, USA.

[15] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM, NY, NY, USA, 759–765. https://doi.org/10.1145/3328778.3366882

[16] Elliot B. Koffman, David Stemple, and Caroline E. Wardle. 1985. Recommended Curriculum for CS2, 1984: A Report of the ACM Curriculum Task Force for CS2. *Commun. ACM* 28, 8 (Aug 1985), 815–818. https://doi.org/10.1145/4021.214936

[17] Yujia Li, David Choi, Junyoung Chung, Julian Schrittwieser, et al. 2022. Competition-level Code Generation With AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[18] Andrew Luxton-Reilly. 2016. Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, NY NY, USA, 284–289. https://doi.org/10.1145/2899415.2899432

[19] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, et al. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. ACM, NY, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[20] Cade Metz. 2021. A.I. Can Now Write Its Own Computer Code. Thats Good News for Humans. *The New York Times* (Sep 2021). https://www.nytimes.com/2021/09/09/technology/codex-artificial-intelligence-coding.html

[21] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, Pittsburgh, US, 1–5. https://doi.org/10.1145/3524842.3528470

[22] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, et al. 2022. Synchromesh: Reliable Code Generation from Pre-Trained Language Models. (January 2022). http://arxiv.org/abs/2201.11227

[23] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*. ACM, NY, NY, USA, 27–43.

[24] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 308–311. https://doi.org/10.1109/ICPC.2017.9

[25] Maddie Simens. 2022. Understanding Codex Training Data and Outputs | OpenAI Help Center. https://help.openai.com/en/articles/5480054-understanding-codex-training-data-and-outputs. (Accessed May 03, 2022).

[26] Beth Simon, Mike Clancy, Robert McCartney, Briana Morrison, Brad Richards, et al. 2010. Making Sense of Data Structures Exams. In *Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10)*. ACM, NY, NY, USA, 97–106. https://doi.org/10.1145/1839594.1839612

[27] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, NY, NY, USA, 1–7.

[28] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, et al. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235