

MEASURING CODING CHALLENGE COMPETENCE WITH APPS

Dan Hendrycks* UC Berkeley
 Steven Basart* UChicago
 Saurav Kadavath UC Berkeley
 Mantas Mazeika UIUC
 Akul Arora UC Berkeley
Ethan Guo UC Berkeley
 Collin Burns UC Berkeley
 Samir Puranik UC Berkeley
 Horace He Cornell
Dawn Song UC Berkeley
Jacob Steinhardt UC Berkeley

ABSTRACT

Programming is one of the most broadly applicable skills in modern society. Due to recent advances in language modeling, training machine learning systems to generate code has become increasingly appealing. However, it can be difficult to accurately assess code generation performance, and there has been surprisingly little work testing and assessing code generation capabilities in a way that is both flexible and rigorous. To fill this gap, we introduce APPS, a benchmark for code generation. Unlike prior work in more restricted settings, our benchmark assesses the ability of language models to take an arbitrary natural language specification and generate Python code fulfilling this specification. Similar to how companies assess candidate software developers, we then evaluate models by checking their generated code on a battery of test cases. Our benchmark includes 10,000 problems, which range from having simple one-line solutions to being algorithmic challenges that are difficult even for computer scientists. We fine-tune GPT-2 on both GitHub and our training set, and we find that while computational accuracy remains low, current models often generate reasonable guesses and frequently solve simple problems. As automatic code generation becomes an increasingly possible and critical application of machine learning over the coming years, our benchmark can provide an informative measure for tracking progress. Code and the dataset is available at <https://github.com/hendrycks/apps>.

“Everybody should learn to program a computer, because it teaches you how to think” – *Steve Jobs*

1 INTRODUCTION

Computer programming is an extraordinarily general tool supporting much of modern society. Spanning finance, scientific discovery, entertainment, and more, the applications supported by coding are vast in scope and highly valuable. The Turing-completeness of programming languages ensures that almost any cognitive task can, in principle, be automated with a computer. Accordingly, aspiring programmers study for years to become proficient coders and eventually are able to convert abstract specifications of diverse cognitive tasks into concrete programs.

In the past few years, large-scale language models have shown promise in generalizing to various cognitive tasks, including linguistic inference (Wang et al., 2019a), commonsense reasoning (Zellers et al., 2019; Huang et al., 2019; Bisk et al., 2019), logical deduction (Liu et al., 2020), and general understanding of many domains of human knowledge (Hendrycks et al., 2021b). However, whether large-scale language models can reliably write code remains an open question. In qualitative demonstrations, state-of-the-art systems (Brown et al., 2020) have exhibited impressive coding ability, completing Python functions into correct programs given only a docstring, but there has been surprisingly little work on rigorously testing the coding ability of large-scale language models.

*Equal Contribution.

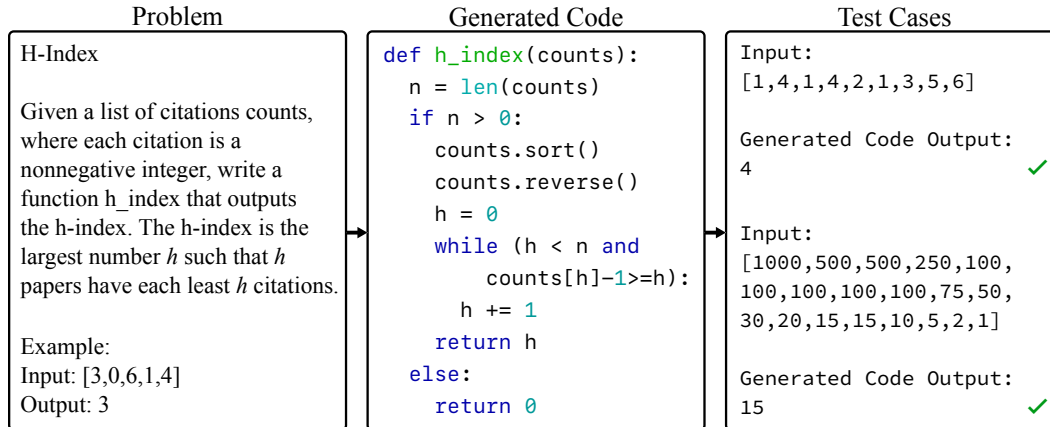


Figure 1: An example problem from APPS (left) along with possible generated code (middle) and two example test cases we use to evaluate the generated code (right). Our evaluation framework is flexible and rigorous, and our dataset contains a diverse set of code generation problems.

Motivated by the need for rigorous evaluation and the potential demonstrated by these models, we introduce APPS, a benchmark for code generation from natural language specifications. Unlike prior work on code generation with language models that focuses on pseudocode and translating between programming languages, we evaluate models on their ability to take specifications given in natural language and write code that meets these specifications. Thus, we are evaluating models on their ability to perform language-based program synthesis. This setting mirrors how human coders are evaluated and provides a highly realistic and economically valuable setting in which to benchmark models.

APPS enables robust evaluation of models along several dimensions, providing a comprehensive and clear view of code generation ability. It contains 10,000 programming problems at various levels of difficulty, ranging from simple introductory problems to complex coding challenges. This highlights a substantial difference between APPS and prior work on evaluating language model code generation: APPS evaluates models not only on their ability to understand coding syntax, but also on their ability to understand task descriptions and devise algorithms to accomplish these tasks. If a model were to perform well on APPS, this would imply a deep understanding of data structures and programming techniques, as well as an ability to correctly interpret diverse task specifications.

In most natural language generation problems, high-quality evaluation requires human feedback, which is expensive. As a result, automatic metrics such as BLEU (Papineni et al., 2002) are often used to compare methods. For code generation, the objective is to produce correct functionality, enabling gold-standard evaluations through the use of test cases and error catching. Evaluating code generation on APPS is facilitated by a large bank of over 100,000 test cases and over 260,000 human-written solutions. The test cases are specifically chosen to probe correct functionality across the input space, including edge cases. Combined with the human-written solutions, this gives a complete picture of model performance in terms of code functionality and similarity to ground-truth solutions.

In experiments with large-scale language models, we evaluate language-based program synthesis using the APPS benchmark. We find that while current models are often able to write functioning code with proper syntax, they are unable to reliably generate correct solutions. Performance improves with scale, and we find that pre-training on an additional 30 GB of Python code from GitHub provides benefits. However, overall accuracy on test cases remains low. The largest model we evaluate, with 1.5 billion parameters, only achieves 2.08% accuracy on the test cases. These results position language-based program synthesis as a challenging testbed for large-scale language models. While our results show that performance can be reliably improved, new methods may be required to reach high accuracies at a reasonable budget.

Writing code to meet specifications in natural language is an economically valuable task with widespread implications should it be solved. With large-scale language models now able to solve tasks requiring reasoning and critical thinking, it is important to measure how well they perform at

Problem

You are given a string $s = s_1s_2 \dots s_n$ of length n , which only contains digits 1, 2, ..., 9. A substring $s[l \dots r]$ of s is a string $s_l s_{l+1} s_{l+2} \dots s_r$. A substring $s[l \dots r]$ of s is called even if the number represented by it is even. Find the number of even substrings of s . Note, that even if some substrings are equal as strings, but have different l and r , they are counted as different substrings. The first line contains an integer n ($1 \leq n \leq 65000$) — the length of the string s . The second line contains a string s of length n . The string s consists only of digits 1, 2, ..., 9. Print the number of even substrings of s .

Model Output

```
n = int(input())
s = input()
ans = 0
for i in range(n):
    for j in range(i, n):
        if int(s[i:j+1]) % 2 == 0:
            ans += 1
print(ans)
```

Problem

You are given a prime number p , n integers a_1, a_2, \dots, a_n , and an integer k . Find the number of pairs of indexes (i, j) ($1 \leq i < j \leq n$) for which $(a_i + a_j)(a_i^2 + a_j^2) \equiv k \pmod{p}$. The first line contains integers n, p, k ($2 \leq n \leq 3 \cdot 10^5$, $2 \leq p \leq 10^9$, $0 \leq k \leq p - 1$). p is guaranteed to be prime. The second line contains n integers a_1, a_2, \dots, a_n ($0 \leq a_i \leq p - 1$). It is guaranteed that all elements are different. Output a single integer answer to the problem.

Model Output

```
n, p, k = map(int, input().split())
a = list(map(int, input().split()))
a.sort()
ans = 0
for i in range(n - 1):
    for j in range(i + 1, n):
        ans += (a[i] + a[j]) % p == k
print(ans)
```

Figure 2: Model-generated code from GPT-2 1.5B. 18/18 test cases pass for the problem on the left and 6/9 test cases pass on the right. Note that models were given in problems exactly as they are formatted here, which even meant parsing \LaTeX .

code generation. Our new benchmark facilitates measuring performance in a comprehensive and rigorous manner. Using APPS, we find that programming is very difficult for modern language models, but that progress is possible. We hope that our benchmark can provide a clearer picture into the performance of future large-scale language models at language-based program synthesis.

2 THE APPS DATASET

The APPS dataset consists of data collected from different open-access domains such as Codewars and Codeforces. The APPS benchmark uses this dataset to mirror the evaluation of human programmers as they progress from beginner to expert level by posing coding exercises in unrestricted natural language and evaluating the correctness of solutions. The exercises range in difficulty from introductory to collegiate competition level and measure coding ability as well as problem-solving.

The Automated Programming Progress Standard, abbreviated APPS, consists of 10,000 programming exercises in total, with over 100,000 test cases for checking solutions, and over 250,000 ground-truth solutions written by humans. The exercises are split evenly into training and test sets, with 5,000 exercises each. Each exercise has at least one ground-truth solution and test case, but many exercises have hundreds of test cases, enabling comprehensive checking of program functionality.

To create the APPS dataset, we manually curated 10,000 programming exercises from open-access sites where programmers share problems with each other, including Codewars and Codeforces. Problems are posed as unconstrained natural language specifications of what should be coded. The APPS dataset has a total of Problems have a mean of 21.6 ground-truth solutions, for a total of 274,045 Python solutions in the APPS dataset.

To measure the correctness of generated solutions, we also collected 102,843 test cases, with a mean of 20.6 test cases per problem for the test set. Across the training and test sets, average question length is 279.5 words. Problems were standardized by difficulty into three different levels: introductory (e.g., get the n th element in a linked list), interview (e.g., Google interviews), and competition (e.g., USACO and IOI problems).

Model	Accuracy				Partial Credit Accuracy			
	Introductory	Interview	Competitive	Average	Introductory	Interview	Competitive	Average
GPT-2 0.1B	2.40	1.27	0.00	1.24	6.28	7.57	3.74	6.55
GPT-2 1.5B	4.80	1.73	0.40	2.08	9.71	9.60	6.52	9.00

Table 1: Accuracy with and without partial credit for the GPT-2 0.1B and 1.5B models. All values are percentages. Accuracy drops off as the difficulty level increases, and increasing model size by an order of magnitude yields a an approximate $2\times$ increase in accuracy.

3 EXPERIMENTS

In this section, we perform experiments to investigate performance on the APPS benchmark using state-of-the-art Transformer (Vaswani et al., 2017) language models.

3.1 ANALYZING MODEL PERFORMANCE.

Models and Hyperparameters. We assess various sizes of GPT-2 (Radford et al., 2019). The GPT architecture is especially appropriate for text generation because it is autoregressive, unlike bidirectional other language models such as BERT (Devlin et al., 2019). GPT-2 also uses a Byte-level Byte-Pair-Encoding tokenizer, capable of representing arbitrary character sequences. This makes it straightforward for the model to read and produce code, which can contain important characters not usually seen in standard text, such as colons and brackets.

Metrics. We show Accuracy and Accuracy with Partial Credit for all models in Table 1. Accuracy indicates the percentage of generated solutions that pass all test cases. This mirrors how humans are evaluated in coding interviews. Meanwhile the Accuracy with partial credit is the fraction of all the test cases passed per problem, averaged across all problems. This is a less stringent metric than standard accuracy.

Quantitative Results. We find that larger models consistently perform better on APPS. The 0.1B parameter model achieves 1.24% accuracy, and the 1.5B parameter model reaches 2.08%. For some problems, models also generate code that passes all test cases (Figure 2). However, absolute accuracies are low across all models, suggesting that solving the APPS benchmark without unreasonable amounts of computational resources may require algorithmic and architectural improvements.

Qualitative Output Analysis. We find that models often output solutions that are not unreasonable even for questions on which they pass zero test cases. For example, in Figure 3, we see that the 1.5B parameter model generates code that is related to the problem statement and makes a plausible attempt to solve it. This further reinforces the idea that even though this task is challenging, progress is now starting to be possible.

4 CONCLUSION

As programming is such a broadly useful and valuable skill, it will likely be one of the main applications of language models over the coming years. However, so far there has been surprisingly limited work on developing and assessing code generation capabilities.

To address this problem, we introduced a new dataset, APPS, for measuring how well machine learning models can generate code. Unlike prior work that focused on pseudocode to code generation or translation between programming languages, our benchmark evaluates how well language models can generate python code given natural language specifications, making it especially useful and natural. We can assess models with hundreds of thousands of test cases and example solutions.

We assessed state-of-the-art generative models on our benchmark and found that while overall performance remains low, models are starting to be able to generate valid code and solve some very basic programming questions. Moreover, even when they make mistakes they can often output reasonable-looking code. This suggests that our benchmark is very challenging, but that progress is starting to be possible on this important problem. As models become more competent at code generation, APPS can help us track this progress in a detailed and rigorous way.

REFERENCES

- Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 207–216. IEEE, 2013.
- Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. Sygus-comp 2018: Results and analysis. *SYNT*, 2018.
- Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language, 2019.
- T. Brown, B. Mann, Nick Ryder, Melanie Subbiah, J. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, G. Krüger, T. Henighan, R. Child, Aditya Ramesh, D. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, E. Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, J. Clark, Christopher Berner, Sam McCandlish, A. Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.
- Jonathon Cai, Richard Shin, and D. Song. Making neural programming architectures generalize via recursion. 2017.
- Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajt Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.
- J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, 2020a. Association for Computational Linguistics.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong (YIMING), Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Findings of EMNLP 2020*, September 2020b.
- Sumit Gulwani, Oleksandr Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4:1–119, 2017.
- Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv 1606.08415*, 2016.
- Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. Aligning ai with shared human values. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021a.
- Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021b.
- Lifu Huang, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Cosmos qa: Machine reading comprehension with contextual commonsense reasoning, 2019.
- Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, October–November 2018. Association for Computational Linguistics.
- Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint*, 2020.

- Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*, volume 32, 2019.
- Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.
- W. Ling, P. Blunsom, Edward Grefenstette, K. Hermann, Tomas Kocisky, Fumin Wang, and A. Senior. Latent predictor networks for code generation. *ArXiv*, abs/1603.06744, 2016.
- J. Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. LogiQA: A challenge dataset for machine reading comprehension with logical reasoning. In *IJCAI*, 2020.
- I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.
- Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, A. Blanco, C. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, L. Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, N. Duan, N. Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. 2021.
- Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *International Conference on Automated Software Engineering (ASE)*, 2015.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.
- Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2020.
- Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic model for code with decision trees. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.
- Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, 2015.
- Shuo Ren, Daya Guo, Shuai Lu, L. Zhou, Shujie Liu, Duyu Tang, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297, 2020.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. Attention is all you need. *ArXiv*, abs/1706.03762, 2017.
- Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *NeurIPS*, 2019a.
- Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *NeurIPS*, 2019b.
- Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*, 2019c.

- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.
- Maksym Zavershynskyi, A. Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. *2nd Workshop on Neural Abstract Machines and Program Induction*, 2018.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.

	Hearthstone	Django	SPoC	NAPS	APPS
Programming Language	Python	Python	C++	UAST	Python
Test Cases	✗	✗	✓	✓	✓
Number of Programs	665	18,805	18,356	17,477	260,447
Lines per Program (Avg.)	7.7	1	14.7	21.7	21.6
Number of Exercises	665	18,805	677	2,231	10,000
Text Input	Card Text	Python	Pseudocode	Pseudocode	Natural Language Problems

Table 2: A comparison of the APPS dataset to existing datasets for converting between text and code. APPS has over an order of magnitude more ground-truth solutions than previous datasets, test cases for every problem, and a language-based program synthesis task where models convert unconstrained natural language specifications into code.

5 APPENDIX

5.1 RELATED WORK

Program Synthesis. Program synthesis is the task of generating a computer program that satisfies given specifications. Often considered a holy grail of computer science if it were to be solved, strong program synthesis capabilities would enable a more intuitive, objective-oriented approach to programming that would save significant amounts of labor. Deductive program synthesis uses specifications in formal logic to define a search problem. Complex optimization techniques are used to generate programs satisfying these specifications Alur et al. (2018). Because specifications must be converted into a formal language, these approaches can be rigid. Inductive synthesis from example input-output behavior can provide an alternative to formal specification Gulwani et al. (2017), but it is often hard to fully specify behavior with examples, as any machine learning practitioner is well-aware.

An alternative to formal or inductive specification is to specify program behavior in natural language. This is arguably how human programmers receive specifications, but few works have investigated whether machine learning methods can do the same. Raza et al. (2015) and Desai et al. (2016) develop approaches to programming by natural language and demonstrate the generation of short programs in domain-specific languages. In contrast, we consider open-ended natural language specifications and general-purpose programming languages. Enabled by recent progress in NLP, we benchmark large-scale Transformers pretrained on tens of gigabytes on language-based program synthesis using our new APPS benchmark.

Neural Representations for Code. Deep learning has provided a new avenue into processing and understanding code and algorithmic structure. Cai et al. (2017) find that incorporating recursion as an inductive bias in neural networks improves generalization when learning sorting algorithms. Feng et al. (2020a) gather a dataset of natural language documentation and accompanying code across 6 programming languages. Using this dataset, they demonstrate that a bidirectional Transformer can perform automatic documentation, even for a programming language not in the training set. Jain et al. (2020) create a source-to-source compiler that can generate diverse, functionally equivalent variants of a program. They then use this with a contrastive learning scheme to train models for various downstream coding tasks, including type inference and code summarization. However, they do not consider code generation.

Code Generation with Language Models. Language modeling is a compelling tool to use for code generation, and several works have achieved success generating code with language models in limited settings. Lachaux et al. (2020) use unsupervised machine translation techniques to translate functions across programming languages, attaining identical behavior after translation in many cases. Zavershynskiy et al. (2018) introduce the NAPS dataset for converting pseudocode to code, obtained by crowdsourcing low-level descriptions of programming exercises, and apply machine translation techniques to the problem. Kulal et al. (2019) introduce SPoC, a method for converting pseudocode to code utilizing seq2seq machine translation with an additional search step. To train SPoC, they collect line-by-line descriptions of C++ programs using Amazon Mechanical Turk. Yu et al. (2018) introduce the Spider dataset for converting natural language queries into SQL. Wang et al. (2019c) develop a framework attaining 65.6% exact match accuracy on this dataset. Lu et al. (2021) introduce the CodeXGLUE benchmark, which unifies a number of published datasets and evaluates performance with the CodeBLEU metric Ren et al. (2020). Iyer et al. (2018) investigate

Model	Maximum Pairwise BLEU				Accuracy			
	Introductory	Interview	Competitive	Average	Introductory	Interview	Competitive	Average
CodeBERT	15.58	8.34	4.39	10.99	0.10	1.01	0.75	0.74
GPT-2 0.1B	20.84	16.00	11.35	17.65	0.41	1.02	0.71	0.77
GPT-2 0.3B	22.15	16.39	12.42	18.47	0.41	1.09	0.49	0.88
GPT-2 1.5B	24.22	17.74	14.95	20.07	0.11	1.53	0.89	1.12

Table 3: Maximum Pairwise BLEU and test case accuracy for each model and difficulty level.

generating Java code from docstrings and evaluate performance with BLEU. The docstrings are often incomplete specifications of what should be coded and only 14.7 words long on average, e.g. “Convert mixed case to underscores”. By comparison, problem specifications in our new APPS benchmark are self-contained and have an average length of 279.5 words. Additionally, APPS contains test cases for every exercise, enabling a high-quality evaluation of code correctness.

Evaluating Large-Scale Language Models. Modern large-scale language models have demonstrated impressive capabilities across a variety of text-based tasks. On the SuperGLUE benchmark Wang et al. (2019b), two models now exceed human accuracy. On many commonsense reasoning benchmarks, performance is rising quickly (Zellers et al., 2019; Huang et al., 2019; Bisk et al., 2019). Even in a multitask evaluation of language models across a broad range of human knowledge, performance is surprisingly high and poised to improve as models are scaled up further Hendrycks et al. (2021b;a).

With the rapid improvements across numerous benchmarks, finding resilient benchmarks on which models significantly underperform humans is challenging. The question of whether more data and more compute is sufficient appears to be answered in the affirmative for most benchmarks that have been proposed so far. APPS differs from existing benchmarks in that, while performance can be improved upon, the baseline performance of large-scale models is very low. Thus, APPS allows us to cleanly separate model performance from that of expert humans.

5.2 FURTHER DATASET DETAILS

5.2.1 PROBLEM FORMATS

Problems in APPS come in two formats, one call-based format and one standard input format:

- **Call-Based Format** problems generally provide initial starter code, usually in the form of a function header, and ask for the solution to be provided as the function’s return value.
- **Standard Input Format** problems generally lack starter code. Instead, the model is only provided with the problem and must output its answers to the STDOUT stream, such as by using print statements.

For the call-based format problems, we prompt models using the following inputs:

```
"\nQUESTION:\n" + q_str + "\n" + starter_code_str + "\n" + "\nUse
Call-Based Format\n\nANSWER:\n"
```

For both of the above prompts, the variable `q_str` represents the raw text of the problem statement. The variable `starter_code_str` represents the starter code given in the problem definition, or the empty string if not starter code was provided. For the standard input format problems, we prompt the model with the input string as before, but we replace “Call-Based Format” with “Standard Input Format.”

Note that if starter code is given, it is only part of the input. This means that to use the starter code, a model must learn to copy the starter code at the beginning of its outputted answer in order to get the question correct.

6 ADDITIONAL DATASET INFORMATION

Dataset Comparison. In the main paper, we compare our dataset to several code translation and pseudocode-to-code datasets Ling et al. (2016); Oda et al. (2015); Kulal et al. (2019); Zavershynskiy

Problem

You are given two integers n and m . Calculate the number of pairs of arrays (a, b) such that: the length of both arrays is equal to m ; each element of each array is an integer between 1 and n (inclusive); $a_i \leq b_i$ for any index i from 1 to m ; array a is sorted in non-descending order; array b is sorted in non-ascending order. As the result can be very large, you should print it modulo $10^9 + 7$. Input: The only line contains two integers n and m ($1 \leq n \leq 1000$, $1 \leq m \leq 10$). Output: Print one integer – the number of arrays a and b satisfying the conditions described above modulo $10^9 + 7$.

Model Output

```
n, m = map(int, input().split())
mod = 10**9+7

fact = [1, 1]
factinv = [1, 1]
inv = [0, 1]

for i in range(2, n+1):
    fact.append((fact[-1] * i) % mod)
    inv.append((-inv[mod % i] * (mod//i)) % mod)
    factinv.append((factinv[-1] * inv[-1]) % mod)

def cmb(n, r, mod):
    if (r < 0) or (n < r):
        return 0
    r = min(r, n - r)
    return fact[n] * factinv[r] * factinv[n-r] % mod

print((cmb(n, m, mod) * cmb(m-1, n-1, mod)) % mod)
```

Figure 3: An example from GPT-2 1.5B. Although the code generated passes 0 test cases, it looks plausible at first glance.

	CodeTrans	CONCODE	GitHub	Java Corpus	PY150	APPS
Programming Language	Java/C#	Java	Java	Java	Python	Python
Test Cases	✗	✗	✗	✗	✗	✓
Number of Programs	11,800	104,000	N/A	N/A	N/A	260,447
Lines per Program (Avg.)	5.4/6.0	26.3	1	1	1	21.6
Number of Exercises	11,800	104,000	10,000	3,000	3,000	10,000
Text Input	Java/C#	Docstrings	Java	Java	Python	Natural Language Problems

Table 4: A comparison of the APPS dataset to additional datasets for converting between text and code. APPS has far more ground-truth solutions than all previous datasets, test cases for every problem, and a language-based program synthesis task where models convert unconstrained natural language specifications into code.

et al. (2018). In Table 4, we compare to additional datasets on code generation from docstrings, code translation, and code completion Lu et al. (2021); Iyer et al. (2018); Raychev et al. (2016); Allamanis & Sutton (2013).

‘Number of Programs’ refers to the number of human-written programs or functions in the dataset, and ‘Number of Exercises’ refers to the number of tasks that the network must solve. These numbers can differ in datasets such as APPS with multiple human-written solutions per exercise.

Excluded Keywords. In creating the GitHub pretraining dataset, we exclude the following keywords to prevent overlap with coding challenge questions similar to those in APPS: ‘atcoder’, ‘coderbyte’, ‘leetcode’, ‘codeforces’, ‘codewars’, ‘hackerrank’, ‘topcoder’, ‘codechef’, ‘checkio’, ‘HackerEarth’, ‘Programmrr’, ‘Exercism’, ‘Codier’, ‘PyBites’, ‘Tynker’, ‘CodinGame’, ‘CodeCombat’, ‘usaco’, ‘IOI’, ‘UVA’, ‘ICFP’, ‘EPIJudge’, ‘SPOJ’, ‘UVaOJ’, ‘judge’, ‘interview’, ‘solution’, ‘coding’, ‘code’, ‘problem’, ‘exercise’, ‘challenge’, ‘algo’, ‘practice’, ‘competitive’, ‘program’.

6.0.1 PRETRAINING DATASET

In addition to the APPS evaluation dataset, we also collect a large dataset of Python code on GitHub from libraries.io/data. We used this dataset for pretraining models before fine-tuning, then testing, using the APPS dataset.

GitHub repositories that were less than one star were filtered out, as were all repositories matching certain keywords that would indicate overlap with common programming exercises. The list of keywords is in the Supplementary Materials. We also discard any GitHub code that contains functions with the same signatures as functions in the starter code in many of our APPS problems. This leaves us with 30 GB of Python code. To improve the efficiency of pretraining, we process all Python code in the pretraining dataset by converting from spaces to tabs, which saves the character conversion when running model tokenizers.

Pretraining Stage. We start with standard English-text pretrained GPT-2 models from the Huggingface Transformers library (Wolf et al., 2020). We then apply two stages of training. In the first stage (“Pretraining”), we use the standard autoregressive LM objective on Python code data that we scraped from GitHub and described in Section 6.0.1.

In order to make sure that pretraining on code for many steps does not cause our models do not forget how to model standard English text, approximately 10% of data is from Wikipedia, and 5% of our examples are programming questions and answers from StackOverflow.

Fine-tuning Stage. In the fine-tuning stage, we train on APPS. The task is to predict the entire coding challenge solution (i.e. Python code), given the problem statement (English text) and the type of problem (call-based format or standard input format). As before, we train at this stage using the standard autoregressive language modeling objective on just the solutions. For problems with starter code, we exclude the starter code from the training loss.

Across both stages, we use the GELU activation function (Hendrycks & Gimpel, 2016), the AdamW optimizer (Loshchilov & Hutter, 2019), a batch size of 256, and a weight decay of 0.05. We fine-tune for 10 epochs. We use DeepSpeed and its implementation of the ZeRO optimizer to reduce memory consumption while training large models (Rasley et al., 2020; Rajbhandari et al., 2020). Unless otherwise specified, we use the default HuggingFace generation parameters, except that we use beam search with a beam size of 10.

6.1 ADDITIONAL RESULTS

CodeBERT Results. We assess CodeBERT’s performance on our task (Feng et al., 2020b). While CodeBERT is bidirectional, it can be fine-tuned for generation, as done in the original paper to generate code documentation. In Table 3, we fine-tune CodeBERT on APPS and observe that it performs worse than our GPT-2 models.

Generated Code. Additional examples of generated code can be found in Figure 4, Figure 5, and Figure 6.

Problem

Given is a directed graph G with N vertices and M edges. The vertices are numbered 1 to N , and the i -th edge is directed from Vertex A_i to Vertex B_i . It is guaranteed that the graph contains no self-loops or multiple edges. Determine whether there exists an induced subgraph (see Notes) of G such that the in-degree and out-degree of every vertex are both 1. If the answer is yes, show one such subgraph. Here the null graph is not considered as a subgraph.

Model Output

```
import sys

sys.setrecursionlimit(10 ** 7)
input = sys.stdin.readline
f_inf = float('inf')
mod = 10 ** 9 + 7

def resolve():
    n, m = map(int, input().split())
    edge = [[] for _ in range(n)]
    for _ in range(m):
        a, b = map(int, input().split())
        edge[a - 1].append(b - 1)
        edge[b - 1].append(a - 1)

    def dfs(u):
        stack = [u]
        seen = [False] * n
        while stack:
            u = stack.pop()
            if seen[u]:
                continue
            seen[u] = True
            for v in edge[u]:
                if seen[v]:
                    continue
                seen[v] = True
                stack.append(v)

    res = None
    for i in range(n):
        if not seen[i]:
            seen[i] = True
            res = dfs(i)
    if not res:
        print(-1)
    else:
        print(res)

if __name__ == '__main__':
    resolve()
```

Figure 4: An example generated by GPT-2 1.5B. The model can sometimes compose function calls.

Problem

Given is a tree G with N vertices. The vertices are numbered 1 through N , and the i -th edge connects Vertex a_i and Vertex b_i . Consider painting the edges in G with some number of colors. We want to paint them so that, for each vertex, the colors of the edges incident to that vertex are all different. Among the colorings satisfying the condition above, construct one that uses the minimum number of colors.

Model Output

```
import sys
from collections import deque, defaultdict, Counter
from itertools import accumulate, product, permutations, combinations
from operator import itemgetter
from bisect import bisect_left, bisect_right
from heapq import heappop, heappush
from math import ceil, floor, sqrt, gcd, inf
from copy import deepcopy
import numpy as np
import scipy as sp

INF = inf
MOD = 1000000007

n = int(input())
ab = [list(map(int, input().split())) for i in range(n - 1)]

graph = [[] for i in range(n + 1)]
for a, b in ab:
    graph[a].append(b)
    graph[b].append(a)

visited = [False] * (n + 1)
color = [0] * (n + 1)
color[0] = 1
stack = [1]
while stack:
    x = stack.pop()
    for y in graph[x]:
        if visited[y]:
            continue
        visited[y] = True
        color[y] = color[x] + 1
        stack.append(y)

ans = min(color)
print(*ans, sep="\n")
```

Figure 5: An additional generated example. Here, the model output has many import statements.

Problem

There is a special square room with mirrors on each of the four walls. Except for the southwest corner, there are receptors on each of the remaining corners, numbered 0, 1, and 2. The square room has walls of length p , and a laser ray from the southwest corner first meets the east wall at a distance q from the 0th receptor. Return the number of the receptor that the ray meets first. (It is guaranteed that the ray will meet a receptor eventually.)

Example 1:

Input: $p = 2, q = 1$

Output: 2

Explanation: The ray meets receptor 2 the first time it gets reflected back to the left wall.

Starter Code

```
class Solution:
    def mirrorReflection(self, p: int, q: int) -> int:
```

Model Output

```
class Solution:
    def mirrorReflection(self, p: int, q: int) -> int:
        self.p = p
        self.q = q
        self.mirrors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        return self.mirrors[self.p - 1][self.q - 1]
```

Figure 6: An example using the call-based format with some starter code.