GENERATING PROGRAMMING PUZZLES TO TRAIN LANGUAGE MODELS

Patrick Haluptzok
Microsoft Research
haluptzok@live.com

Matthew Bowers*
M.I.T.
mlbowers@mit.edu

Adam Tauman Kalai
Microsoft Research
adam@kal.ai

ABSTRACT

This work shows how one can use large-scale language models (LMs) to automatically generate programming problems with verified solutions, in the form of "programming puzzles," which can then in turn be used to fine-tune other LMs to solve more difficult programming puzzles. This work builds on two recent developments. First, LMs have achieved breakthroughs in non-trivial reasoning and algorithm implementation, generating code that can solve some intermediate-level competitive programming problems [e.g., Chen et al., 2021; Li et al., 2022]. However, training code LMs involves curated sets of natural-language problem descriptions and source-code tests and solutions, which are limited in size. Second, a new format of programming challenge called a programming puzzle was introduced, which does not require a natural-language description and is directly specified by a source-code test [Schuster et al., 2021]. In this work we show how generating synthetic programming puzzles and solutions, verified for correctness by a Python interpreter, can be used to improve performance in solving test puzzles from P3, a public benchmark set of Python Programming Puzzles. In particular, we find that knowledge is distilled both from the interpreter and the language model that generates and solves the synthetic puzzles. It also opens the door to iterative self-improvement for LMs in future work.

1 INTRODUCTION

Recent advances [Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021; Li et al., 2022] show that transformer-based language models (LMs), pre-trained on massive corpora of text and code, can complete programming tasks from English instructions. At a high level they share a common structure: they all start with multi-billion-parameter LMs pre-trained on multi-gigabyte corpora including source code (e.g., GitHub), they all curate smaller "problem sets" of standalone programming problems, and they all find that pre-trained LMs can generate code to solve some fraction of held-out test problems. Each problem is defined by a precise English description, as well as test cases to further reduce ambiguity and facilitate correctness evaluation. The problem sets draw from programming competitions and other sources of standalone tests. Each problem can be solved by a self-contained function $g(x)$. Fine-tuning the pre-trained models on training problems from these problem sets further improves performance on test problems.

A key challenge to improving these data hungry LMs is procuring large quantities of high quality training programming problems and matching solutions. Programming problem datasets are expensive to author and curate, and thus limited in size. Manually generating problem descriptions in a natural language such as English, with correct code solutions and comprehensive test cases, is time consuming. While significant efforts have been made to harvest the existing competitive programming websites and GitHub repositories for standalone programming problems, the number of human authored problem descriptions with matching code solutions is limited. Second, one may consider augmenting these datasets with synthetically generated problems but evaluating the quality and correctness of the English language problem descriptions would be difficult. Many problem descriptions wouldn't make sense, and it is difficult with standard metrics like perplexity to verify which English problem descriptions correspond to a meaningful and precise problem. On top of that, a corresponding code

*Work done while at Microsoft Research

```
def f(c: int):
    return c + 50000 == 174653

def g():
    return 174653 - 50000
```

```
def f(x: str, chars=['Hello', 'there', 'you!'], n=4600):
    return x == x[::-1] and all([x.count(c) == n for c in chars])

def g(chars=['Hello', 'there', 'you!'], n=4600):
    s = "".join([c*n for c in chars])
    return s + s[::-1]
```

```
def f(graph: List[List[int]], start=0, target=37):
    def dfs(node):
        nums.add(node)
        for next_node in graph[node]:
            if next_node not in nums:
                dfs(next_node)
    nums = set()
    dfs(start)
    return len(nums) == 37

...
```

Figure 1: Illustrative puzzles and solutions that were synthesized by the language models: first is a simple equation; the second requires finding a palindrome (string same forwards and backwards) with exactly `n` copies of each of a given list of substrings; and the third is a simple graph-theory puzzle asking for a di-graph where exactly 37 nodes can be reached from node 0 (solution not shown here). All puzzles have verified synthetic solutions. Weaknesses include the fact that `f3` does not use the `target` variable and the poor variable name `chars` for a list of strings.

solution needs to be synthesized for each generated English problem description and predicting whether synthesized code solves problems as described would be yet another challenge.

Programming puzzles [Schuster et al., 2021] represent programming problems using code only,[1] emphasizing problem-solving and understanding code (rather than English). A puzzle is specified by a function $f(y, x)$ together with zero or more inputs x. The goal is to find y such that $f(y, x) =$ `True`. A puzzle f is an output verifier that validates y as a solution to f on input x. The answer $y = g(x)$ is the output of a synthesized program g. Examples of (synthetic) puzzles are given in Figure 1. In order to find a solution g, a code synthesizer is given the source code of f (and x), with the goal of generating a program g such that $f(g(x), x) =$ `True`. Even if the ultimate goal is solving problems in English, disentangling problem-solving from English-understanding can be useful for evaluating progress in learning to code.

The open-source P3 dataset[2] of Python Programming Puzzles demonstrates that programming puzzles can capture a wide range of challenges from various domains, from trivial string manipulation to longstanding open problems in algorithms and mathematics. Recursion, dynamic programming, and other fundamental programming techniques are all useful in solving the puzzles in P3. Puzzles circumvent the natural-language issues mentioned above, because the validity of puzzles and solutions can be directly verified by simply executing code. Puzzles can be used in a RL-like fashion without requiring matching human-authored solutions, by exploring the space of code problems and solutions, synthesizing problem-solution pairs and reinforcing the model with the pairs that correctly evaluate. In our work, we use human-authored puzzles but not human-authored solutions.

[1] Code may contain helpful natural-language comments, but correctness is evaluated based solely on code.
[2] https://GitHub.com/microsoft/PythonProgrammingPuzzles

Generating new puzzles and solutions. This paper investigates data generation using LMs, by generating synthetic programming puzzles with verified solutions to improve performance on solving future puzzles. While Schuster et al. [2021] showed how to use LMs to solve puzzles, we show how to use LMs to generate puzzles as well, selecting random subsets of P3 puzzles to create few-shot prompts to prime the LM to generate more puzzles. An LM is also used to synthesize the corresponding code solutions g for each programming problem f. The correctness of $f(g(x), x)$ is programatically evaluated, thus allowing for the creation of a large training dataset of verified-correct puzzles and solutions to fine-tune on.

We measure the utility of this synthetic dataset based on how well an LM performs at solving new, held-out test puzzles, after being fine-tuned on the dataset. Such an approach may fail for numerous reasons. The generated puzzles may be too hard and thus discarded because no solutions were found, or too easy with trivial solutions that are not instructive. They may also be too similar to one another, in which case a million synthetic puzzles may not be better than a few. Nonetheless, we hypothesized that fine-tuning on synthetic puzzles and solutions would improve P3 test performance.

To test this hypothesis, we use the publicly available API for the Codex LM [Chen et al., 2021] to synthesize nearly one million puzzle-solution pairs that are verified correct. Codex is a GPT3-like transformer model [Brown et al., 2020] that has been trained on a large corpus of code and a smaller corpus of standalone programming problems. P3 provides hundreds of hand-written training and test puzzles. We synthesize a set of 950k verified-correct synthetic puzzle-solution pairs, by running Codex on prompts derived from random P3 training puzzles. There is no limit on the number of puzzles and solutions that could be synthesized in this manner. Since fine-tuning on the large variants of Codex is not yet publicly available, we instead fine-tune the smaller EleutherAI open-source GPT-Neo model [Black et al., 2021] on the synthetic dataset. GPT-Neo is another GPT3-like model which has been pre-trained on the Pile [Gao et al., 2020], a dataset containing a large sample of code from GitHub repositories among other natural language data. Finally, GPT-Neo is used to solve test puzzles in a few-shot manner, following Schuster et al. [2021].

As a baseline, we compare to GPT-Neo with no fine-tuning, which solves few of P3's test puzzles. GPT-Neo, once fine-tuned on the 950k verified synthetic puzzle-solution pairs, solves approximately five times as many puzzles as the baseline model. We also evaluate two alternative LMs. The first is GPT-Neo fine-tuned on the P3 training puzzles with synthetic solutions. Second, we fine-tune GPT-Neo on a set of 950k *unverified* synthetic puzzle-solution pairs (without correctness filtering). Both alternatives performed significantly worse than the verified synthetic puzzle-solution pairs.

The main contribution of this work is demonstrating how one can use LMs, together with a Python interpreter, to generate verified programming challenges which can then be used to further improve LMs at solving such challenges. A key ingredient is using the programming puzzle framework to circumvent the ambiguities that plague natural language and input-output-based specifications. The puzzles generated and code used in this project will be available at the P3 repository. This work complements the growing body of work on extracting standalone tests from human-generated code.

The paper is organized as follows. Section 2 describes our approach for generating puzzles and verified solutions, and for using these to improve an LM's ability to solve puzzles. Section 3 presents the evaluation of our approach. Section 4 discusses related work. Finally, we conclude and discuss future work. Appendix A gives further implementation details, while Appendices B-C provide comparisons between synthesized and human-generated puzzles.

2 Learning Pipeline

We first present an overview of our pipeline and then detail the generation, fine-tuning, and evaluation. A high-level view of our pipeline and experiments can be seen in Figure 2. We reiterate that no human hand-written solutions are used for learning or evaluation (other than the five illustrative examples used in the prompt to solve puzzles, as shown in Figure 7).

2.1 Overview

The first step is generating a set of verified puzzle-solution pairs. We created prompts composed of puzzles sampled from the 155 training puzzles of the P3 dataset. Each prompt consists of a series

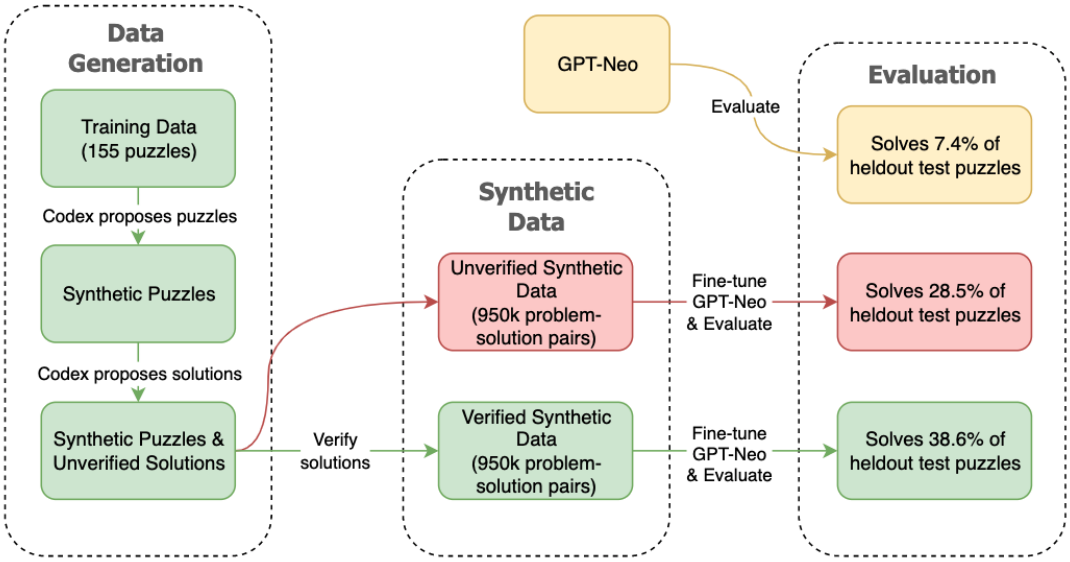


Figure 2: Overview of experiments and results. Our pipeline, which generates and fine-tunes on verified synthetic puzzles and solutions, is shown in green. The verification ablation study is shown in red. A GPT-Neo baseline is shown in yellow. All performance results are from the 2.7B model after one epoch of fine-tuning.

```
def f1(inds: List[int], li=[42, 18, 21, 103, 2, 11], target=[2, 21, 42]):
    i, j, k = inds
    return li[i:j:k] == target

def f2(path: List[List[int]], m=8, n=8, target=35):
    def legal_move(m):
        (a, b), (i, j) = m
        return {abs(i - a), abs(j - b)} == {1, 2}
    ...

def f44(
```

Figure 3: An example prompt for generating puzzles. For each request for a prompt completion, the Codex model would generate a new puzzle.

of puzzles one after another (see Fig. 3). The prompts were fed to Codex in the few-shot learning approach to generate additional puzzles. These generated puzzles formed a large set of synthetic puzzles, for which we then generated synthetic solutions. Solving was performed by prompting Codex with a few-shot prompt demonstrating several problem-solution pairs. Finally, we verified the Codex-generated solutions for correctness through execution in a Python interpreter, discarding any incorrect solutions. Further details are in Section 2.2.

The second step is fine-tuning an LM on the verified synthetic data (Section 2.3) and the final step is evaluating the fine-tuned model on the held-out test split of puzzles provided by P3 (Section 2.4). When solving puzzles, at all stages we use the same prompt as in Schuster et al. [2021], but the model varies.

2.2 Generating Puzzles and Solutions

This section describes how the puzzles and solutions were generated. Further details of the process appear in Appendix A. At the time of writing, the P3 repository contained 397 programming puzzles with a training and test split, as well as Codex-generated solutions to those puzzles that were solved based on 1,000 attempts.

Name	Puzzle source	Num. puzzles	Num. human solutions
Train	P3	155	0
Test	P3	228	0

Table 1: Statistics from the training and test set used from the P3 dataset.

Fine-tune dataset	Verified	Puzzles	Solutions	# Tokens	pass@100
BASELINE	N/A	No puzzles	No solutions	0	7.4%
HUMAN	Yes	Human	Synthetic (635)	74K	7.4%
UNVERIFIED	No	Synthetic	Synthetic (950k)	132M	28.5%
VERIFIED	Yes	Synthetic	Synthetic (950k)	92M	38.6%

Table 2: Statistics for the dataset used for fine-tuning in our experiments. The same puzzle may be repeated in a single dataset with multiple (fewer than 10) distinct solutions. pass@100 are after 1 epoch of fine-tuning on GPT-Neo2.7B.

In order to generate puzzles, we created a simple prompt which consisted of a sample of training puzzles (as large as possible that fit within the Codex API limit, median 43 puzzles) as illustrated in Figure 3. We then applied filtering, eliminating duplicate puzzles, puzzles with an invalid argument type hint[3], puzzles which did not parse in Python, and puzzles which had a "trivial" solution. For example, if a puzzle took an **`int`** solution, we tested to ensure that it did not have a solution in $\{-10, -9, \ldots, 100\}$. In total, approximately half of the generated puzzles were eliminated during this pre-filtering process. We then used Codex to attempt to solve each of these puzzles, in turn, using the same few-shot tutorial prompt used in P3 (see Figure 7), which consists of five sample puzzles *and solutions*–these same five samples were given to humans in their user study. For each puzzle, 128 candidate solutions were generated. Each of these solutions was judged as correct or incorrect based on whether it solved the generated puzzle, using a one-second timeout as in the P3 evaluation. We then take the solutions judged to be correct (up to a maximum of 8 distinct solutions per puzzle, taking the shortest 8 for the puzzles that had more than 8). Because each puzzle requires one completion to generate but is solved 128 times, we use the larger Davinci Codex model for generating puzzles and the smaller and faster Cushman Codex model for solving puzzles.

2.3 FINE-TUNING

After generating synthetic data, the models were fine-tuned at the task of solving puzzles. The format of the fine-tuning data mirrors that of the few-shot solving prompt shown in Figure 1, excluding puzzle numbers because there were so many puzzles. A breakdown of the datasets used for fine-tuning is given in Table 2. The BASELINE dataset is included for comparison and doesn't involve any fine-tuning. The HUMAN dataset consists of correct solutions to 95 puzzles out of the 155 puzzle P3 training. These solutions were generated by Codex and verified in the same fashion as described above (Section 2.2) for solving the synthetic puzzles. The UNVERIFIED dataset was created by the procedure in Section 2.2 but all Codex-proposed solutions to the synthetic puzzles were used, without discarding the problem-solution pairs that were not correct. The VERIFIED dataset consisted of 950K puzzles that were generated from the 155 puzzle train set, along with their generated solutions that passed verification as detailed in Section 2.2. In each case, the puzzle-solution pairs were shuffled (keeping each solution following its corresponding puzzle) and concatenated into a single corpus for fine-tuning.

None of the human annotated solutions for any of the P3 programming puzzles were used in the construction of any fine-tuning datasets, as the objective in this paper is to measure the ability to bootstrap the system to solve more puzzles through synthetic data with minimal human-labelling.

The experiments on fine-tuning LMs to improve their performance on solving programming puzzles were done with the GPT-Neo model [Black et al., 2021], as described in the introduction. GPT-Neo is

[3]A valid puzzle has a single required argument with a type that must be a **`bool`**, **`float`**, **`int`**, **`str`**, or `List[]`'s thereof, nested to arbitrary depth.

publicly available pre-trained in 3 different sizes of 125 million, 1.3 billion, and 2.9 billion parameters. These pre-trained models were used as the baseline models for these experiments, and fine-tuning was done starting from these models. Experiments were run across all 3 model sizes, yielding qualitatively similar results across the different model sizes. In Figure 5 we compare different amounts of fine-tuning, and in all other cases models were fine-tuned for one epoch (92 million tokens). Fine-tuning was done at a learning rate of $5 \cdot 10^{-6}$ with the AdamW variant [Loshchilov & Hutter, 2018] of the Adam optimizer [Kingma & Ba, 2014] with $\beta_1 = .9$ and $\beta_2 = .99$.

2.4 SOLVING TEST PUZZLES

The same few-shot approach used for solving synthetic puzzles was used to solve test puzzles, as described in Section 2.2. The same tutorial prompt used in P3 was also used, as shown in Figure 7, and a 1-second timeout for evaluation. The models used to solve test puzzles were the fine-tuned models discussed above. It is also worth noting that, while a significant amount of time was spent generating synthetic puzzles and fine-tuning the models, this is a one-time cost where the model could be used to solve many future puzzles, more than we presently have in our test set. Hence, amortizing the cost, generating synthetic puzzles would be less computationally expensive.

3 EXPERIMENTAL EVALUATIONS

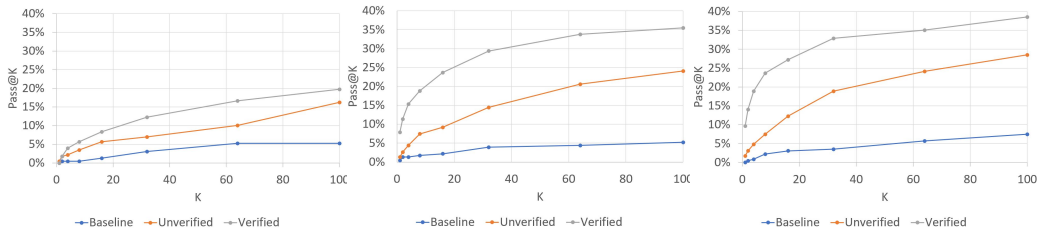
For solving test puzzles, GPT-Neo was given 100 attempts to generate a solution at temperature of 0.8 as used by Chen et al. [2021] using the few-shot prompt as described in 2.2. A temperature sweep was performed as shown in Figure 6. In future work the temperature sweep could use a validation set of puzzles, a data split which is not presently included in P3.

3.1 PASS@K SOLVING METRIC

Consistent with prior work, results are presented using the Pass@K metric [Chen et al., 2021]. In generating the 100 solutions for each of the 228 problems, the index of the first correct solution obtained for each problem is recorded. Pass@K indicates how many problems had a correct solution generated within the first K solutions. Higher values for K result in solving more problems, as shown in Figure 4. A refined N@K metric was introduced in Li et al. [2022], where K solutions are generated and $N \leq K$ of them must be chosen for "submission." This metric is inspired by settings such as competitive programming where a system must choose to submit a limited number of candidate solutions for evaluation. Thus in general, Pass@K is equivalent to K@K, and AlphaCode used 10@K for large values of K (e.g., millions). For puzzles, since at most one submission is need, Pass@K is in fact equivalent to the 1@K metric, because no solver need ever submit an incorrect solution. In some competitions such as those on the popular `codeforces.com` website, there is a cost for incorrect submissions. For puzzles, one should never incur such a cost.

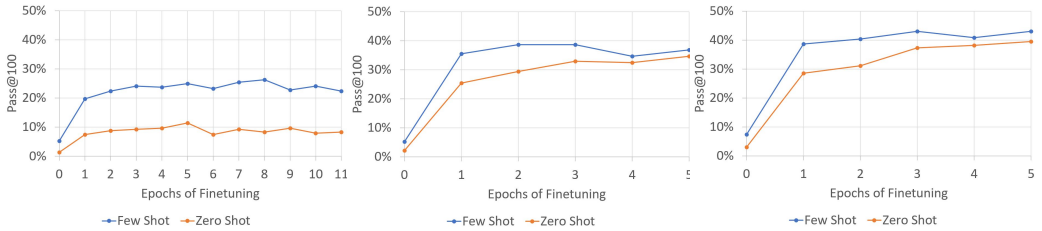
3.2 RESULTS

We measured how successfully GPT-Neo could solve the 228 test programming puzzles in the few-shot regime using the Pass@K metric. The first experiment involved fine-tuning GPT-Neo on the small HUMAN dataset described in Section 2.3, which was constructed from 635 synthesized solutions to the 155 handwritten puzzles in the training set. Fine-tuning on that small dataset didn't yield noticeable improvement in the accuracy of GPT-Neo. Next, we compared the GPT-Neo baseline to GPT-Neo fine-tuned on the UNVERIFIED dataset and to GPT-Neo fine-tuned on VERIFIED dataset. As shown in Figure 4, fine-tuning on the unverified data improved the Pass@K performance across all models, and verified data gave an additional considerable boost to performance. The importance of using the few-shot learning prompt for solving was analyzed and the results are displayed in Figure 5. Even after extensive fine-tuning on the puzzle problem format for over 1 billion tokens, the LM still performed better when prompted with the 5 examples of puzzles/solutions to prime the model.



(a) GPT-Neo 125M (b) GPT-Neo 1.3B (c) GPT-Neo 2.7B

Figure 4: Pass@K for the three models. On the test set across all 3 model sizes increasing the number of K solutions generated by GPT-Neo results in an increase in Pass@K. Pass@K corresponds to the number of puzzle problems that had at least one correct solution generated in K attempts. The graph shows how using verified generated data dominates using unverified generated data in fine-tuning, and that fine-tuning dominates using just the baseline model.



(a) GPT-Neo 125M (b) GPT-Neo 1.3B (c) GPT-Neo 2.7B

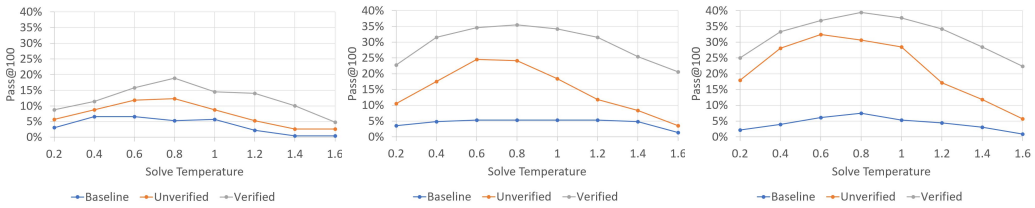
Figure 5: Few-shot vs. zero-shot and fine-tuning epochs. Across all 3 model sizes, testing GPT-Neo in few-shot beats zero-shot significantly even after 11 epochs of fine-tuning which is over 1 billion tokens of generated puzzle-problem/solution pairs. The LM still benefits from providing the P3 tutorial puzzle prompt.

4 Related Work

Until recently, much work in program synthesis is on Programming by Example (PBE) in Domain-Specific Languages (DSLs), where problems are specified by input-output pairs. This has proven useful in applications such as string manipulation [see, e.g., the survey by Gulwani et al., 2017]. Like English descriptions, PBE is inherently ambiguous. Recent work on massive transformer based LMs [Chen et al., 2021; Schuster et al., 2021; Austin et al., 2021; Li et al., 2022] has enabled synthesis in general-purpose programming languages like Python. Many works have studied data augmentation by synthesizing input-output examples [e.g., Balog et al., 2017; Shin et al., 2019; Alet et al., 2021; Li et al., 2022]. Bootstrapping has also been studied in example-based program synthesis [e.g., Menon et al., 2013; Ellis et al., 2021].

To facilitate evaluation, many related datasets of programming problems have been curated, including especially relevant standalone programming challenges described in English and code [Zavershynskyi et al., 2018; Hendrycks et al., 2021; Austin et al., 2021; Chen et al., 2021; Li et al., 2022]. Schuster et al. [2021] and similarly Li et al. [2022] make an important distinction between two types of programming problems: those that only involve *translation* and those that require *problem-solving*. Translation problems, such as "Add up all the odd numbers in array x," require the LM to translate a procedure from natural language to code. *Problem-solving* is required when the description does not state *how* to solve the problem. For example, "Find a path of length at most 17 between nodes 1 and 2 in graph x" conveys the problem to solve but not how to go about finding a path. Puzzles focus on problem-solving rather than translation.

This work can also be viewed through the lens of knowledge distillation (see Gou et al. [2021] for a survey). In knowledge distillation a student model is trained to imitate the behavior of a teacher model on some data, and in the *data-free* paradigm the training data itself is synthetically generated. This paper can be viewed as knowledge distillation from the teacher that generates and solves synthetic problems to the student that is fine-tuned and evaluated, with the additional innovation that we



(a) GPT-Neo 125M (b) GPT-Neo 1.3B (c) GPT-Neo 2.7B

Figure 6: Temperature controls the amount of diversity in the code solutions generated by GPT-Neo. All experiments in our paper were done with a fixed temperature of 0.8, based on the recommendation for Pass@100 in Chen et al. [2021]. A hyper-parameter sweep on temperature across all 3 model sizes verified that 0.8 was also optimal for our model and dataset at a 0.2 search step size for maximizing Pass@100 which is the percentage of problems solved at least once with 100 generated code solutions per problem. GPT-Neo was finetuned for 1 epoch, $\sim$92 million tokens in these graphs.

verify solutions before fine-tuning on them. The verification is key as it increases the data quality above what the teacher model produced. It is not particularly important that we use separate teacher and student models in this work, and even with a single model this pipeline would essentially be a form of *self-distillation*, where the model is using the verifier to self-improve (discussed in Section **??**). Recent work in commonsense knowledge graphs by West et al. [2021] has explored filtering language model outputs for quality during knowledge distillation using a neural filter. This shares the filtering aspect of our work, but given the ambiguity of their natural language task they can't evaluate correctness directly, unlike in the programming puzzle paradigm.

5 Conclusions and Future work

This paper describes a new method for generating programming problems and verified solutions automatically without the need for any human-written solutions. By leveraging the programming puzzle paradigm, this paper eliminates the ambiguity in judging whether a solution to a synthetically generated English problem description is correct, as puzzle solutions can be verified with a Python interpreter. We show that fine-tuning language models on generated problems and solutions which have been verified for correctness enhances the ability of the model to solve new problems. We find that fine-tuning the LM on verified generated data is superior to using unverified generated data, using the limited human-authored data set, and to the baseline LM. We also analyze the importance of the few-shot prompt in puzzle solving which yields a significant performance boost even after extensive fine-tuning. This work sets the stage for future work in which one could iterate the pipeline of puzzle generation, solving, and fine-tuning, allowing a language model to improve in a bootstrapping manner to solve increasingly difficult problems.

In future work, it would be interesting to generate puzzles using GPT-Neo rather than Codex to explore how solution verification enables a language model to improve it's own solving performance. Similarly, one could fine-tune Codex on the verified synthetic data to see if it improves its own solving performance. The method in this paper uses different LMs for generation and solving. One could use the same LM for both generation and solving to reduce the total model size and explore whether fine-tuning yields benefits for generation as well. As noted earlier, one could loop the pipeline described in this paper. Specifically, after generating puzzles, solving and verifying solutions, and fine-tuning the LM on the verified dataset, one could begin the pipeline again using the fine-tuned model by generating puzzles and using the improved solving ability to solve even more complex synthetic problems.

References

Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine*

Learning Research, pp. 175–186. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/alet21a.html.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Representation Learning (ICLR)*, 2017.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5551208.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. *DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning*, pp. 835–850. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383912. URL https://doi.org/10.1145/3453483.3454080.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge Distillation: A Survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021. ISSN 1573-1405. doi: 10.1007/s11263-021-01453-z. URL https://doi.org/10.1007/s11263-021-01453-z.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, Feb 2022. URL https://storage.googleapis.com/deepmind-media/AlphaCode/competition_level_code_generation_with_alphacode.pdf.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. 2018.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pp. 187–195, 2013.

Tal Schuster, Ashwin Kalyan, Alex Polozov, and Adam Tauman Kalai. Programming puzzles. In *Thirty-fifth Conference on Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=fe_hCc4RBrg.

Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. 2019. URL https://openreview.net/pdf?id=ryeOSnAqYm.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena D. Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. Symbolic knowledge distillation: from general language models to commonsense models. 2021.

Maksym Zavershynskyi, Alexander Skidanov, and Illia Polosukhin. NAPS: natural program synthesis dataset. *CoRR*, abs/1807.03168, 2018. URL http://arxiv.org/abs/1807.03168.

A Details of puzzle generation and solving

Figure 7 shows the prompt used to solve puzzles: the same prompt used (a) in P3 to solve the training puzzles, (b) to solve the generated puzzles, and (c) to solve the test puzzles. It is worth noting that fewer than 1% of puzzles were duplicates. The fixed temperature of 0.9 from prior work Schuster et al. [2021] was used in all puzzle-solving for generating fine-tuning data, where temperature of 0.8 was used for testing the fine-tuned model per Chen et al. [2021].

In solving puzzles, both synthetic puzzles and P3 puzzles, we use the same judging code from the P3 repository.[4] Their evaluation identifies syntax errors and aborts infinite loops using timeouts. Their judge prevents some malicious instructions from being executed by automated code checks, though other judging systems perform full sand-boxing of the computation to prevent a generated code sample from doing harm like deleting files.

B Further examples of generated puzzles

A hand examination was performed on a subset of the generated puzzles, where we attempted to understand how the puzzles may originate. We found several concepts repeated from the training, other human concepts such as days of the week, and other puzzles that appear to be derived from programming challenges on the web. We found many human concepts misused, such as the perimeter of a triangle being confused with its side. Additionally input variables were sometimes unused, or puzzles did not test what they appeared like they should test because of certain issues they contained. Finally, comments were sometimes generated of varying quality.

For several puzzles, we attempted to delve deeper to understand the origin for the puzzle. For instance, `f2` from Figure 1 seems similar in spirit (but not identical) to several of the training problems. Here is a P3 training problem that is somewhat related:

```
def train(s: str, substrings=['foo', 'bar', 'baz']):
    return all(sub in s and sub[::-1] in s for sub in substrings)
```

[4]We additionally set the PYTHONHASHSEED environment variable to 0 to make Python **set** functions deterministic.

```
from typing import List

def f1(s: str):
    return "Hello " + s == "Hello world"

def g1():
    return "world"

assert f1(g1())

def f2(s: str):
    return "Hello " + s[::-1] == "Hello world"

def g2():
    return "world"[::-1]

assert f2(g2())

def f3(x: List[int]):
    return len(x) == 2 and sum(x) == 3

def g3():
    return [1, 2]

assert f3(g3())

def f4(s: List[str]):
    return len(set(s)) == 1000 and all((x.count("a") > x.count("b")) and
    ('b' in x) for x in s)

def g4():
    return ["a"*(i+2)+"b" for i in range(1000)]

assert f4(g4())

def f5(n: int):
    return str(n * n).startswith("123456789")

def g5():
    return int(int("123456789" + "0"*9) ** 0.5) + 1

assert f5(g5())

def f6(inds: List[int], string="Sssuubbstrissiingg"):
    return inds == sorted(inds) and "".join(string[i] for i in inds) == "
    substring"

def g6(string="Sssuubbstrissiingg"):
```

Figure 7: An example of the prompt used for solving puzzles, identical to the "medium prompt" of P3 [Schuster et al., 2021, Figure C.3]. The first five example puzzles `f1-f5` are always the same. The puzzle to be solved is also provided in the prompt as `f6`, and the solution function signature is provided as `g6`.

Both involve testing palindromes and substrings.

More surprisingly, the following sophisticated problem was generated:

```
def f(n: int, target=20151120):
    assert 0 <= n <= 1e14
    next = lambda x: (x * 252533) % 33554393
    seen = set()
    now = 20151120
    while now not in seen:
        seen.add(now)
        now = next(now)
        if now == target:
            return n == 0
        n -= 1
    return False
    next = lambda x: (x * 252533) % 33554393
    now = 20151120
    n = 0
    while next(now) != target:
        n += 1
        now = next(now)
    return n
```

This problem requires computing a discrete log. While the discrete log problem is notoriously difficult and is the basis of numerous cryptography systems, the number is small enough that it can be solved by a simple loop. The P3 dataset does contain a discrete log problem but it is in the test set. While we could not find the exact code above, the problem itself does appear to be equivalent to the English challenge stated on this programming challenge website: https://adventofcode.com/2015/day/25. It is still unclear how exactly the system generated this code.

The following puzzle asks for a list of triangles of *perimeter* 5, but uses the variable name *side*, suggesting that it may not understand the difference between perimeter and side. The puzzle has an additional constraint which is clearly poor programming as it refers to undefined variables a1 and a2. Consequently, solving this requires finding a list of a single triangle of perimeter 5, such as `[[2,2,1]`.

```
def f(ls: List[List[int]], a=24, b=16, c=24, target=None, side=5):
    for a, b, c in ls:
        assert a <= side and b <= side and c <= side and a + b + c ==
    side, "Invalid triangle"
    if not target:
        target = ls[-1]

    def legal_move(m):
        (a, b, c), (i, j, k) = m
        return ((a == side or a == b + c) and a == a1 and a != a2) or a
    == a2 and a != a1 and a != b + c

    a1, a2, a3 = target
    moves = list(zip(ls, ls[1:]))
    return all(legal_move(m) for m in moves)


    return [[a,b,c] for a in range(side+1)
                for b in range(side-a+1)
                for c in range(side-a-b+1)
                if a + b + c == side
                and (a == side or a == b + c)]
```

Several puzzles included concepts (like vowels) and specific strings (like the famous pangram below) that appeared in the training data.

```
def f(w: str, z="The quick brown fox jumps over the lazy dog", n=2):
    return w.count("a") + w.count("e") + w.count("i") + w.count("o") + w.
    count("u") == n and w in z and w != z
```

Many puzzles were not particularly interesting such as the two below, which involve finding a string of a given length containing a given substring, and finding a list of 21 numbers between 1-9 that sum to 100.

```
def f(s: str, t="rome", length=14):
    return len(s) == length == len(set(s.upper())) and t.upper() in s.
    upper()

def f(li: List[int]):
    return len(li) == 21 and all(i in li for i in range(1, 10)) and sum(
    li) == 100
```

Other puzzles involved very human-like strings:

```
def f(m: str):
    assert m.startswith("Hello, Salif")
    assert "But, but..." in m
    assert m.endswith("You're great!")
    return len(m) == 282

def g():
    return "Hello, Salif. But, but... If a friend ever said hello to me,
    I wonder where are you from? A freaky fellow? Are you from a freaky
    galaxy?" + \
    " or are you from a freaky universe or a freaky planet? The answer is
     no: I'm megalomaniac!" + \
    " I know because I don't translate meaning. You're great!"
```

Other puzzles involved human concepts such as the day of week which did not appear in the training data:

```
def f(days: List[str], x="tue", k=3, n=4):
    nums = {"mon": 0, "tue": 1, "wed": 2, "thu": 3, "fri": 4, "sat": 5, "
    sun": 6}
    numx = nums[x]
    return (len(set(days)) <= k and (n - len(set(days))) * n >= n * (1 +
    (n - 1) // k) and numx <= n // 2 and
            numx != nums[days[n // 2]] and numx > nums[days[0]] and numx
    < nums[
        days[-1]])   # right half of week is weekdays
        days[:n//2]   # left half of week is weekends
```

The comments that are generated are sometimes useful and sometimes incorrect.

C TRAINING PUZZLES

This section presents the 155 training puzzles used to generate synthetic puzzles.

```
# Training problem 1
def f(s: str):
    return s.count('o') == 1000 and s.count('oo') == 0

# Training problem 2
def f(li: List[int]):
    return sorted(li) == list(range(999)) and all(li[i] != i for i in
    range(len(li)))

# Training problem 3
def f(li: List[int]):
```

```
    return all([li.count(i) == i for i in range(10)])

# Training problem 4
def f(s: str):
    return str(8 ** 2888).count(s) > 8 and len(s) == 3

# Training problem 5
def f(li: List[int]):
    return ["The quick brown fox jumps over the lazy dog"[i] for i in li]
     == list(
        "The five boxing wizards jump quickly")

# Training problem 6
def f(ls: List[str]):
    return min(ls) == max(ls) == str(len(ls))

# Training problem 7
def f(x: float):
    return str(x - 3.1415).startswith("123.456")

# Training problem 8
def f(li: List[int]):
    return all(sum(li[:i]) == 2 ** i - 1 for i in range(20))

# Training problem 9
def f(i: int):
    return len(str(i + 1000)) > len(str(i + 1001))

# Training problem 10
def f(li: List[int]):
    return {i + j for i in li for j in li} == {0, 1, 2, 3, 4, 5, 6, 17,
    18, 19, 20, 34}

# Training problem 11
def f(li: List[int]):
    return all([li[i] != li[i + 1] for i in range(10)]) and len(set(li))
    == 3

# Training problem 12
def f(ls: List[str]):
    return tuple(ls) in zip('dee', 'doo', 'dah!')

# Training problem 13
def f(s: str):
    return sorted(s) == sorted('Permute me true') and s == s[::-1]

# Training problem 14
def f(li: List[int]):
    return li[li[0]] != li[li[1]] and li[li[li[0]]] == li[li[li[1]]]

# Training problem 15
def f(l: List[int]):
    return all(i in range(1000) and abs(i * i - j * j) >= 10 for i in l
    for j in l if i != j) and len(set(l)) > 995

# Training problem 16
def f(hands: List[int], target_angle=45):
    h, m = hands
    assert 0 < h <= 12 and 0 <= m < 60
    hour_angle = 30 * h + m / 2
    minute_angle = 6 * m
    return abs(hour_angle - minute_angle) in [target_angle, 360 -
    target_angle]

# Training problem 17
```

```
def f(daygroups: List[List[List[int]]]):
    assert len(daygroups) == 7
    assert all(len(groups) == 5 and {i for g in groups for i in g} == set
    (range(15)) for groups in daygroups)
    assert all(len(g) == 3 for groups in daygroups for g in groups)
    return len({(i, j) for groups in daygroups for g in groups for i in g
     for j in g}) == 15 * 15

# Training problem 18
def f(n: int):
    for i in range(5):
        assert n % 5 == 1
        n -= 1 + (n - 1) // 5
    return n > 0 and n % 5 == 1

# Training problem 19
def f(coords: List[List[int]], side=10, num_points=20):
    for i1 in range(len(coords)):
        x1, y1 = coords[i1]
        assert 0 <= x1 < side and 0 <= y1 < side
        for i2 in range(i1):
            x2, y2 = coords[i2]
            for i3 in range(i2):
                x3, y3 = coords[i3]
                assert x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)
    != 0
    return len({(a, b) for a, b in coords}) == len(coords) >= num_points

# Training problem 20
def f(stamps: List[int], target=80, max_stamps=4, options=[10, 32, 8]):
    for s in stamps:
        assert s in options
    return len(stamps) <= max_stamps and sum(stamps) == target

# Training problem 21
def f(xy_sides: List[List[int]]):
    n = max(x + side for x, y, side in xy_sides)
    assert len({side for x, y, side in xy_sides}) == len(xy_sides) > 1
    for x, y, s in xy_sides:
        assert 0 <= y < y + s <= n and 0 <= x
        for x2, y2, s2 in xy_sides:
            assert s2 <= s or x2 >= x + s or x2 + s2 <= x or y2 >= y + s
    or y2 + s2 <= y

    return sum(side ** 2 for x, y, side in xy_sides) == n ** 2

# Training problem 22
def f(n: int, lace="
    bbrbrbbbbbbrrrrrrrbrrrrbbbrbrrbbbrbrrrbrrbrrbrbbrrrrrbrbbbrrrbbbrbbrbbbbrbrbb
    "):
    sub = lace[n: n + len(lace) // 2]
    return n >= 0 and lace.count("r") == 2 * sub.count("r") and lace.
    count("b") == 2 * sub.count("b")

# Training problem 23
def f(n: int):
    s = str(n * n)
    for i in "0123456789":
        assert s.count(i) == 1
    return True

# Training problem 24
def f(s: str):
    return set(s) <= set("18-+*/") and s.count("8") == 2 and s.count("1")
     == 1 and eval(s) == 63
```

```
# Training problem 25
def f(s: str):
    return set(s) <= set("18-+*/") and s.count("8") == 3 and s.count("1")
     == 1 and eval(s) == 63

# Training problem 26
def f(moves: List[List[int]], capacities=[8, 5, 3], init=[8, 0, 0], goal
    =[4, 4, 0]):
    state = init.copy()

    for [i, j] in moves:
        assert min(i, j) >= 0, "Indices must be non-negative"
        assert i != j, "Cannot pour from same state to itself"
        n = min(capacities[j], state[i] + state[j])
        state[i], state[j] = state[i] + state[j] - n, n

    return state == goal

# Training problem 27
def f(li: List[int], words=['SEND', 'MORE', 'MONEY']):
    assert len(li) == len(words) and all(i > 0 and len(str(i)) == len(w)
    for i, w in zip(li, words))
    assert len({c for w in words for c in w}) == len({(d, c) for i, w in
    zip(li, words) for d, c in zip(str(i), w)})
    return sum(li[:-1]) == li[-1]

# Training problem 28
def f(s: str, word="antidisestablishmentarianism", max_len=10):
    if len(word) <= max_len:
        return word == s
    return int(s[1:-1]) == len(word[1:-1]) and word[0] == s[0] and word
    [-1] == s[-1]

# Training problem 29
def f(corners: List[List[int]], m=10, n=9, a=5, target=4):
    covered = {(i + x, j + y) for i, j in corners for x in range(a) for y
     in range(a)}
    assert len(covered) == len(corners) * a * a, "Double coverage"
    return len(corners) <= target and covered.issuperset({(x, y) for x in
     range(m) for y in range(n)})

# Training problem 30
def f(lb: List[bool], trips=[[1, 1, 0], [1, 0, 0], [0, 0, 0], [0, 1, 1],
    [0, 1, 1], [1, 1, 1], [1, 0, 1]]):
    return len(lb) == len(trips) and all(
        (b is True) if sum(s) >= 2 else (b is False) for b, s in zip(lb,
    trips))

# Training problem 31
def f(n: int, scores=[100, 95, 80, 70, 65, 9, 9, 9, 4, 2, 1], k=6):
    assert all(scores[i] >= scores[i + 1] for i in range(len(scores) - 1)
    ), "Hint: scores are non-decreasing"
    return all(s >= scores[k] and s > 0 for s in scores[:n]) and all(s <
    scores[k] or s <= 0 for s in scores[n:])

# Training problem 32
def f(t: str, s="Problems"):
    i = 0
    for c in s.lower():
        if c in "aeiouy":
            continue
        assert t[i] == ".", f"expecting `.` at position {i}"
        i += 1
        assert t[i] == c, f"expecting `{c}`"
```

```
        i += 1
    return i == len(t)

# Training problem 33
def f(squares: List[List[int]], m=10, n=5, target=50):
    covered = []
    for i1, j1, i2, j2 in squares:
        assert (0 <= i1 <= i2 < m) and (0 <= j1 <= j2 < n) and (j2 - j1 +
     i2 - i1 == 1)
        covered += [(i1, j1), (i2, j2)]
    return len(set(covered)) == len(covered) == target

# Training problem 34
def f(n: int, ops=['x++', '--x', '--x'], target=19143212):
    for op in ops:
        if op in ["++x", "x++"]:
            n += 1
        else:
            assert op in ["--x", "x--"]
            n -= 1
    return n == target

# Training problem 35
def f(n: int, s="aaAab", t="aAaaB"):
    if n == 0:
        return s.lower() == t.lower()
    if n == 1:
        return s.lower() > t.lower()
    if n == -1:
        return s.lower() < t.lower()
    return False

# Training problem 36
def f(s: str, word="konjac"):
    for i in range(len(word)):
        if i == 0:
            if s[i] != word[i].upper():
                return False
        else:
            if s[i] != word[i]:
                return False
    return True

# Training problem 37
def f(t: str, s="abbbcabbac", target=7):
    i = 0
    for c in t:
        while c != s[i]:
            i += 1
        i += 1
    return len(t) >= target and all(t[i] != t[i + 1] for i in range(len(t
    ) - 1))

# Training problem 38
def f(delta: List[int], nums=[[1, 2, 3], [9, -2, 8], [17, 2, 50]]):
    return all(sum(vec[i] for vec in nums) + delta[i] == 0 for i in range
    (3))

# Training problem 39
def f(n: int, a=17, b=100, c=20):
    return n + a == sum([b * i for i in range(c)])

# Training problem 40
def f(n: int, v=17, w=100):
    for i in range(n):
```

```
        assert v <= w
        v *= 3
        w *= 2
    return v > w

# Training problem 41
def f(res: int, m=1234578987654321, n=4):
    for i in range(n):
        m = (m - 1 if m % 10 else m // 10)
    return res == m

# Training problem 42
def f(n: int, pairs=[[3, 0], [17, 1], [9254359, 19], [123, 9254359], [0,
    123]]):
    assert sum(p - m for p, m in pairs) == 0, "oo"
    tot = 0
    success = False
    for p, m in pairs:
        tot -= m
        tot += p
        assert tot <= n
        if tot == n:
            success = True
    return success

# Training problem 43
def f(s_case: str, s="CanYouTellIfItHASmoreCAPITALS"):
    caps = 0
    for c in s:
        if c != c.lower():
            caps += 1
    return s_case == (s.upper() if caps > len(s) // 2 else s.lower())

# Training problem 44
def f(inds: List[int], string="enlightenment"):
    return inds == sorted(inds) and "".join(string[i] for i in inds) == "
    intelligent"

# Training problem 45
def f(d: int, n=123456789):
    return d > n and all(i in "47" for i in str(str(d).count("4") + str(d
    ).count("7")))

# Training problem 46
def f(s: str, target="reverse me", reverse=True):
    return (s[::-1] == target) == reverse

# Training problem 47
def f(s: str, a=5129, d=17):
    return s.count("a") == a and s.count("d") == d and len(s) == a + d

# Training problem 48
def f(nums: List[int], a=100, b=1000, count=648):
    assert all(len(str(n)) == len(set(str(n))) and a <= n <= b for n in
    nums)
    return len(set(nums)) >= count

# Training problem 49
def f(tot: int, nums=[2, 8, 25, 18, 99, 11, 17, 16], thresh=17):
    return tot == sum(1 if i < thresh else 2 for i in nums)

# Training problem 50
def f(s: str, chars=['o', 'h', 'e', 'l', ' ', 'w', '!', 'r', 'd']):
    for c in chars:
        if c not in s:
```

```
            return False
    return True

# Training problem 51
def f(ans: List[List[int]], target=17):
    for i in range(len(ans)):
        a, b = ans[i]
        if b - a >= 2:
            target -= 1
    return target == 0

# Training problem 52
def f(indexes: List[int], target=[1, 3, 4, 2, 5, 6, 7, 13, 12, 11, 9, 10,
     8]):
    for i in range(1, len(target) + 1):
        if target[indexes[i - 1] - 1] != i:
            return False
    return True

# Training problem 53
def f(s: str, n=7012):
    return int(str(5 ** n)[:-2] + s) == 5 ** n

# Training problem 54
def f(states: List[str], start="424", combo="778", target_len=12):
    assert all(len(s) == len(start) for s in states) and all(c in "
    0123456789" for s in states for c in s)
    for a, b in zip([start] + states, states + [combo]):
        assert sum(i != j for i, j in zip(a, b)) == 1
        assert all(abs(int(i) - int(j)) in {0, 1, 9} for i, j in zip(a, b
    ))

    return len(states) <= target_len

# Training problem 55
def f(states: List[str], start="424", combo="778", target_len=12):
    return all(sum((int(a[i]) - int(b[i])) ** 2 % 10 for i in range(len(
    start))) == 1
               for a, b in zip([start] + states, states[:target_len] + [
    combo]))

# Training problem 56
def f(s: str, perm="qwertyuiopasdfghjklzxcvbnm", target="hello are you
    there?"):
    return "".join((perm[(perm.index(c) + 1) % len(perm)] if c in perm
    else c) for c in s) == target

# Training problem 57
def f(lists: List[List[int]], items=[5, 4, 9, 4, 5, 5, 5, 1, 5, 5],
    length=4):
    a, b = lists
    assert len(a) == len(b) == length
    assert len(set(a)) == len(a)
    assert len(set(b)) == 1
    for i in a + b:
        assert (a + b).count(i) <= items.count(i)
    return True

# Training problem 58
def f(seq: List[int], n=10000, length=5017):
    return all(i in [1, 2] for i in seq) and sum(seq) == n and len(seq)
    == length

# Training problem 59
```

```
def f(start: int, k=3, upper=6, seq=[17, 1, 2, 65, 18, 91, -30, 100, 3, 1,
     2]):
    return 0 <= start <= len(seq) - k and sum(seq[start:start + k]) <=
    upper

# Training problem 60
def f(start: int, k=3, lower=150, seq=[3, 1, 2, 65, 18, 91, -30, 100, 0,
    19, 52]):
    return 0 <= start <= len(seq) - k and sum(seq[start:start + k]) >=
    lower

# Training problem 61
def f(start: int, k=3, lower=100000, seq=[91, 1, 2, 64, 18, 91, -30, 100,
     3, 65, 18]):
    prod = 1
    for i in range(start, start + k):
        prod *= seq[i]
    return prod >= lower

# Training problem 62
def f(nums: List[int], tot=12345, n=5):
    return len(nums) == len(set(nums)) == n and sum(nums) == tot and all(
    i >= i % 2 > 0 for i in nums)

# Training problem 63
def f(rotations: List[int], target="wonderful", upper=69):
    s = "abcdefghijklmnopqrstuvwxyz"
    assert len(rotations) == len(target)
    for r, c in zip(rotations, target):
        s = s[r:] + s[:r]
        assert s[0] == c

    return sum(abs(r) for r in rotations) <= upper

# Training problem 64
def f(bills: List[int], denominations=[1, 25, 35, 84], n=980, max_len=14):

    return sum(bills) == n and all(b in denominations for b in bills) and
     len(bills) <= max_len

# Training problem 65
def f(sides: List[int], options=[2, 512, 1024], n
    =340282366920938463463374607431768211456, max_dim=13):
    prod = 1
    for b in sides:
        prod *= b
    return prod == n and set(sides) <= set(options) and len(sides) <=
    max_dim

# Training problem 66
def f(x: float, coeffs=[2.5, 1.3, -0.5]):
    a, b, c = coeffs
    return abs(a * x ** 2 + b * x + c) < 1e-6

# Training problem 67
def f(roots: List[float], coeffs=[1.3, -0.5]):
    b, c = coeffs
    r1, r2 = roots
    return abs(r1 + r2 + b) + abs(r1 * r2 - c) < 1e-6

# Training problem 68
def f(x: str, s=679):
    return s == sum([int(d) for d in x])

# Training problem 69
```

```
def f(z: float, v=9, d=0.0001):
    return int(z * 1 / d % 10) == v

# Training problem 70
def f(x: List[int], a=7, s=5, e=200):
    return x[0] == a and x[-1] <= e and (x[-1] + s > e) and all([x[i] + s
     == x[i + 1] for i in range(len(x) - 1)])

# Training problem 71
def f(e: List[int], a=2, b=-1, c=1, d=2021):
    x = e[0] / e[1]
    return abs(a * x + b - c * x - d) < 10 ** -5

# Training problem 72
def f(x: int, a=253532, b=1230200):
    if x > 0 or a > 50:
        return x - a == b
    else:
        return x + a == b

# Training problem 73
def f(x: int, a=4, b=54368639):
    if a == 1:
        return x % 2 == 0
    elif a == -1:
        return x % 2 == 1
    else:
        return x + a == b

# Training problem 74
def f(x: List[int], n=5, s=19):
    return len(x) == n and sum(x) == s and all([a > 0 for a in x])

# Training problem 75
def f(x: List[int], n=4, s=2021):
    return len(x) == n and sum(x) == s and len(set(x)) == n

# Training problem 76
def f(x: str, s=['a', 'b', 'c', 'd', 'e', 'f'], n=4):
    return len(x) == n and all([x[i] == s[i] for i in range(n)])

# Training problem 77
def f(x: List[int], t=677, a=43, e=125, s=10):
    non_zero = [z for z in x if z != 0]
    return t == sum([x[i] for i in range(a, e, s)]) and len(set(non_zero)
    ) == len(non_zero) and all(
        [x[i] != 0 for i in range(a, e, s)])

# Training problem 78
def f(x: List[int], t=50, n=10):
    assert all([v > 0 for v in x])
    s = 0
    i = 0
    for v in sorted(x):
        s += v
        if s > t:
            return i == n
        i += 1
    return i == n

# Training problem 79
def f(s: str, s1="a", s2="b", count1=50, count2=30):
    return s.count(s1) == count1 and s.count(s2) == count2 and s[:10] ==
    s[-10:]
```

```
# Training problem 80
def f(s: str, substrings=['foo', 'bar', 'baz', 'oddball']):
    return all(sub in s[i::len(substrings)] for i, sub in enumerate(
    substrings))

# Training problem 81
def f(s: str, substrings=['foo', 'bar', 'baz']):
    return all(sub in s and sub[::-1] in s for sub in substrings)

# Training problem 82
def f(s: str, strings=['cat', 'dog', 'bird', 'fly', 'moose']):
    return s in strings and sum(t > s for t in strings) == 1

# Training problem 83
def f(s: str, strings=['cat', 'dog', 'bird', 'fly', 'moose']):
    return s[::-1] in strings and sum(t < s[::-1] for t in strings) == 1

# Training problem 84
def f(s: str, target="foobarbazwow", length=6):
    return target[(len(target) - length) // 2:(len(target) + length) //
    2] == s

# Training problem 85
def f(substring: str, string="mooobooofasd", count=2):
    return string.count(substring) == count

# Training problem 86
def f(t: str, s="))(Add)some))parens()to()(balance(()(()(me!)(((("):
    for i in range(len(t) + 1):
        depth = t[:i].count("(") - t[:i].count(")")
        assert depth >= 0
    return depth == 0 and s in t

# Training problem 87
def f(squares: List[List[int]], m=8, n=8):
    k = min(m, n)
    assert all(i in range(m) and j in range(n) for i, j in squares) and
    len(squares) == k
    return 4 * k == len({t for i, j in squares for t in [('row', i), ('
    col', j), ('SE', i + j), ('NE', i - j)]})

# Training problem 88
def f(tour: List[List[int]], m=8, n=8):
    assert all({abs(i1 - i2), abs(j1 - j2)} == {1, 2} for [i1, j1], [i2,
    j2] in zip(tour, tour[1:])), 'legal moves'
    return sorted(tour) == [[i, j] for i in range(m) for j in range(n)]
    # cover every square
    once

# Training problem 89
def f(path: List[List[int]], m=8, n=8, target=35):
    def legal_move(m):
        (a, b), (i, j) = m
        return {abs(i - a), abs(j - b)} == {1, 2}

    def legal_quad(m1, m2):   # non-overlapping test: parallel or bounding box has (width - 1) *
    (height - 1) >= 5
        (i1, j1), (i2, j2) = m1
        (a1, b1), (a2, b2) = m2
        return (len({(i1, j1), (i2, j2), (a1, b1), (a2, b2)}) < 4  #
    adjacent edges in path, ignore
                or (i1 - i2) * (b1 - b2) == (j1 - j2) * (a1 - a2)  # parallel
                or (max(a1, a2, i1, i2) - min(a1, a2, i1, i2)) * (max(b1,
     b2, j1, j2) - min(b1, b2, j1, j2)) >= 5
                # far
```

```
                )

    assert all(i in range(m) and j in range(n) for i, j in path), "move
    off board"
    assert len({(i, j) for i, j in path}) == len(path), "visited same
    square twice"

    moves = list(zip(path, path[1:]))
    assert all(legal_move(m) for m in moves), "illegal move"
    assert all(legal_quad(m1, m2) for m1 in moves for m2 in moves), "
    intersecting move pair"

    return len(path) >= target

# Training problem 90
def f(path: List[List[int]], m=10, n=10, target=62):
    def legal_move(m):
        (a, b), (i, j) = m
        return {abs(i - a), abs(j - b)} == {1, 2}

    def legal_quad(m1, m2):   # non-overlapping test: parallel or bounding box has (width - 1) *
    (height - 1) >= 5
        (i1, j1), (i2, j2) = m1
        (a1, b1), (a2, b2) = m2
        return (len({(i1, j1), (i2, j2), (a1, b1), (a2, b2)}) < 4  #
    adjacent edges in path, ignore
                or (i1 - i2) * (b1 - b2) == (j1 - j2) * (a1 - a2)   # parallel
                or (max(a1, a2, i1, i2) - min(a1, a2, i1, i2)) * (max(b1,
     b2, j1, j2) - min(b1, b2, j1, j2)) >= 5
                # far
                )

    assert all(i in range(m) and j in range(n) for i, j in path), "move
    off board"
    assert len({(i, j) for i, j in path}) == len(path), "visited same
    square twice"

    moves = list(zip(path, path[1:]))
    assert all(legal_move(m) for m in moves), "illegal move"
    assert all(legal_quad(m1, m2) for m1 in moves for m2 in moves), "
    intersecting move pair"

    return len(path) >= target

# Training problem 91
def f(position: List[List[int]], target=[[1, 3], [1, 4], [2, 5]]):
    live = {x + y * 1j for x, y in position}   # complex numbers encode live cells
    deltas = (1j, -1j, 1, -1, 1 + 1j, 1 - 1j, -1 + 1j, -1 - 1j)
    visible = {z + d for z in live for d in deltas}
    next_step = {z for z in visible if sum(z + d in live for d in deltas)
     in ([2, 3] if z in live else [3])}
    return next_step == {x + y * 1j for x, y in target}

# Training problem 92
def f(init: List[List[int]], period=4):
    live = {x + y * 1j for x, y in init}   # use complex numbers
    init_tot = sum(live)
    target = {z * len(live) - init_tot for z in live}
    deltas = (1j, -1j, 1, -1, 1 + 1j, 1 - 1j, -1 + 1j, -1 - 1j)

    for t in range(period):
        visible = {z + d for z in live for d in deltas}
        live = {z for z in visible if 3 - (z in live) <= sum(z + d in
    live for d in deltas) <= 3}
        tot = sum(live)
```

```
        if {z * len(live) - tot for z in live} == target:
            return t + 1 == period and tot != init_tot

# Training problem 93
def f(moves: List[List[int]], initial_state=[5, 9, 3, 11, 18, 25, 1, 2, 4,
     1]):

    def bot_move():   # bot takes objects from the largest heap to make it match the second largest
    heap
        vals = sorted(state, reverse=True)
        i_largest = state.index(vals[0])   # largest heap
        state[i_largest] -= max(vals[0] - vals[1], 1)   # must take some, take 1 in
    case of tie

    state = initial_state[:]   # copy
    for i, n in moves:
        assert 0 < n <= state[i], "Illegal move"
        state[i] -= n
        if set(state) == {0}:
            return True   # you won!
        assert any(state), "You lost!"
        bot_move()

# Training problem 94
def f(probs: List[float]):
    assert len(probs) == 3 and abs(sum(probs) - 1) < 1e-6
    return max(probs[(i + 2) % 3] - probs[(i + 1) % 3] for i in range(3))
     < 1e-6

# Training problem 95
def f(strategies: List[List[float]], A=[[1.0, -1.0], [-1.3, 0.8]], B
    =[[-0.9, 1.1], [0.7, -0.8]], eps=0.01):
    m, n = len(A), len(A[0])
    p, q = strategies
    assert len(B) == m and all(len(row) == n for row in A + B), "inputs
    are a bimatrix game"
    assert len(p) == m and len(q) == n, "solution is a pair of strategies
    "
    assert sum(p) == sum(q) == 1.0 and min(p + q) >= 0.0, "strategies
    must be non-negative and sum to 1"
    v = sum(A[i][j] * p[i] * q[j] for i in range(m) for j in range(n))
    w = sum(B[i][j] * p[i] * q[j] for i in range(m) for j in range(n))
    return (all(sum(A[i][j] * q[j] for j in range(n)) <= v + eps for i in
     range(m)) and
            all(sum(B[i][j] * p[i] for i in range(m)) <= w + eps for j in
     range(n)))

# Training problem 96
def f(edges: List[List[int]]):
    # first compute neighbors sets, N:
    N = {i: {j for j in range(99) if j != i and ([i, j] in edges or [j, i
    ] in edges)} for i in range(99)}
    return all(len(N[i].intersection(N[j])) == (1 if j in N[i] else 2)
    for i in range(99) for j in range(i))

# Training problem 97
def f(tri: List[int], edges=[[0, 17], [0, 22], [17, 22], [17, 31], [22,
    31], [31, 17]]):
    a, b, c = tri
    return [a, b] in edges and [b, c] in edges and [c, a] in edges and a
    != b != c != a

# Training problem 98
def f(path: List[int], weights=[{1: 20, 2: 1}, {2: 2, 3: 5}, {1: 10}],
    bound=11):
```

```
    return path[0] == 0 and path[-1] == 1 and sum(weights[a][b] for a, b
    in zip(path, path[1:])) <= bound

# Training problem 99
def f(path: List[int], edges=[[0, 11], [0, 7], [7, 5], [0, 22], [11, 22],
     [11, 33], [22, 33]], u=0, v=33, bound=3):
    assert path[0] == u and path[-1] == v and all([i, j] in edges for i,
    j in zip(path, path[1:]))
    return len(path) <= bound

# Training problem 100
def f(path: List[int], edges=[[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [3,
     4], [5, 6], [6, 7], [1, 2]]):
    for i in range(len(path) - 1):
        assert [path[i], path[i + 1]] in edges
    assert path[0] == 0
    assert path[-1] == max(max(edge) for edge in edges)
    return True

# Training problem 101
def f(path: List[int], edges=[[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [3,
     4], [5, 6], [6, 7], [1, 2]]):
    assert path[0] == 0 and path[-1] == max(max(e) for e in edges)
    assert all([[a, b] in edges for a, b in zip(path, path[1:])])
    return len(path) % 2 == 0

# Training problem 102
def f(p: List[int], edges=[[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [3, 4],
     [5, 6], [6, 7], [6, 1]]):
    return p[0] == 0 and p[-1] == 1 == len(p) % 2 and all([[a, b] in
    edges for a, b in zip(p, p[1:])])

# Training problem 103
def f(bi: List[int], g1=[[0, 1], [1, 2], [2, 3], [3, 4], [2, 5]], g2=[[0,
     4], [1, 5], [4, 1], [1, 2], [2, 3]]):
    return len(bi) == len(set(bi)) and {(i, j) for i, j in g1} == {(bi[i],
     bi[j]) for i, j in g2}

# Training problem 104
def f(position: List[List[int]], transcript=[[[3, 3], [5, 5], [3, 7]],
    [[5, 3], [6, 4]]]):
    board = {(x, y): 0 for x in range(8) for y in range(8) if (x + y) % 2
     == 0}   # empty board, 0 =
    empty
    for x, y, p in position:
        assert -2 <= p <= 2 and board[x, y] == 0   # -1, 1 is regular piece, -2, 2 is king
        board[x, y] = p

    def has_a_jump(x, y):
        p = board[x, y]   # piece to move
        deltas = [(dx, dy) for dx in [-1, 1] for dy in [-1, 1] if dy != -
    p]   # don't check backwards for
    non-kings
        return any(board.get((x + 2 * dx, y + 2 * dy)) == 0 and board[x +
     dx, y + dy] * p < 0 for dx, dy in deltas)

    sign = 1   # player 1 moves first
    for move in transcript:
        start, end = tuple(move[0]), tuple(move[-1])
        p = board[start]   # piece to move
        assert p * sign > 0, "Moving square must be non-empty and players
     must be alternate signs"
        assert all(board[x, y] == 0 for x, y in move if [x, y] != move
    [0]), "Moved to an occupied square"
```

```
        for (x1, y1), (x2, y2) in zip(move, move[1:]):
            assert abs(p) != 1 or (y2 - y1) * p > 0, "Non-kings can only
    move forward (in direction of sign)"
            if abs(x2 - x1) == 1:  # non-jump
                assert not any(has_a_jump(*a) for a in board if board[a]
    * p > 0), "Must make a jump if possible"
                break
            mid = ((x1 + x2) // 2, (y1 + y2) // 2)
            assert board[mid] * p < 0, "Can only jump over piece of
    opposite sign"
            board[mid] = 0
        board[start], board[end] = 0, p
        assert abs(x2 - x1) == 1 or not has_a_jump(*end)
        if abs(p) == 1 and any(y in {0, 7} for x, y in move[1:]):
            board[end] *= 2  # king me at the end of turn after any jumps are done!
        sign *= -1

    return True

# Training problem 105
def f(cut_position: int, ring="
    yRrsmOkLCHSDJywpVDEDsjgCwSUmtvHMefxxPFdmBIpM", lower=5):
    line = ring[cut_position:] + ring[:cut_position]
    matches = {c: 0 for c in line.lower()}
    for c in line:
        if c.islower():
            matches[c] -= (1 if matches[c] > 0 else len(line))
        else:
            matches[c.lower()] += 1
    return sum(i == 0 for i in matches.values()) >= lower

# Training problem 106
def f(nums: List[int], b=7, m=6):
    assert len(nums) == len(set(nums)) == m and min(nums) >= 0

    def gcd(i, j):
        r, s = max(i, j), min(i, j)
        while s >= 1:
            r, s = s, (r % s)
        return r

    for a in nums:
        nums = [(a + i + 1) ** 2 + (a + i + 1) + 1 for i in range(b)]
        assert all(any(i != j and gcd(i, j) > 1 for j in nums) for i in
    nums)

    return True

# Training problem 107
def f(indices: List[int], a0=123):
    assert a0 >= 0 and a0 % 3 == 0, "Hint: a_0 is a multiple of 3."
    s = [a0]
    for i in range(max(indices)):
        s.append(int(s[-1] ** 0.5) if int(s[-1] ** 0.5) ** 2 == s[-1]
    else s[-1] + 3)
    return len(indices) == len(set(indices)) == 1000 and min(indices) >=
    0 and len({s[i] for i in indices}) == 1

# Training problem 108
def f(li: List[int], n=18):
    assert n % 3 == 0, "Hint: n is a multiple of 3"
    return len(li) == n and all(li[(i + 2) % n] == 1 + li[(i + 1) % n] *
    li[i] for i in range(n))

# Training problem 109
```

```
def f(inds: List[int], vecs=[169, 203, 409, 50, 37, 479, 370, 133, 53,
    159, 161, 367, 474, 107, 82, 447, 385]):
    return all(sum((v >> i) & 1 for i in inds) % 2 == 1 for v in vecs)

# Training problem 110
def f(inds: List[int], vecs=[26, 5, 32, 3, 15, 18, 31, 13, 24, 25, 34, 5,
     15, 24, 16, 13, 0, 27, 37]):
    return sum(sum((v >> i) & 1 for i in inds) % 2 for v in vecs) >= len(
    vecs) * 3 / 4

# Training problem 111
def f(nums: List[int]):
    a, b, c, n = nums
    return (a ** n + b ** n == c ** n) and min(a, b, c) > 0 and n > 2

# Training problem 112
def f(n: int, a=15482, b=23223, lower_bound=5):
    return a % n == 0 and b % n == 0 and n >= lower_bound

# Training problem 113
def f(n: int, nums=[77410, 23223, 54187], lower_bound=2):
    return all(i % n == 0 for i in nums) and n >= lower_bound

# Training problem 114
def f(n: int, a=15, b=27, upper_bound=150):
    return n % a == 0 and n % b == 0 and 0 < n <= upper_bound

# Training problem 115
def f(n: int, nums=[15, 27, 102], upper_bound=5000):
    return all(n % i == 0 for i in nums) and 0 < n <= upper_bound

# Training problem 116
def f(nums: List[int], n=12345):
    return len(nums) <= 4 and sum(i ** 2 for i in nums) == n

# Training problem 117
def f(li: List[int], k=5):
    def prod(nums):
        ans = 1
        for i in nums:
            ans *= i
        return ans

    return min(li) > 1 and len(li) == k and all((1 + prod(li[:i] + li[i +
     1:])) % li[i] == 0 for i in range(k))

# Training problem 118
def f(n: int):
    m = n
    while n > 4:
        n = 3 * n + 1 if n % 2 else n // 2
        if n == m:
            return True

# Training problem 119
def f(start: int):
    n = start   # could be positive or negative ...
    while abs(n) > 1000:
        n = 3 * n + 1 if n % 2 else n // 2
        if n == start:
            return True

# Training problem 120
def f(n: int, t=197, upper=20):
    m = n
```

```
    for i in range(t):
        if n <= 1:
            return False
        n = 3 * n + 1 if n % 2 else n // 2
    return n == 1 and m <= 2 ** upper

# Training problem 121
def f(n: int):
    return pow(2, n, n) == 3

# Training problem 122
def f(n: int, year_len=365):
    import random
    random.seed(0)
    K = 1000  # number of samples
    prob = sum(len({random.randrange(year_len) for i in range(n)}) < n
    for j in range(K)) / K
    return (prob - 0.5) ** 2 <= year_len

# Training problem 123
def f(counts: List[int], target_prob=0.5):
    m, n = counts  # m = num 1's, n = num -1's
    probs = [1.0] + [0.0] * n  # probs[n] is probability for current m, starting with m = 1
    for i in range(2, m + 1):  # compute probs using dynamic programming for m = i
        old_probs = probs
        probs = [1.0] + [0.0] * n
        for j in range(1, min(n + 1, i)):
            probs[j] = (
                    j / (i + j) * probs[j - 1]  # last element is a -1 so use probs
                    +
                    i / (i + j) * old_probs[j]  # last element is a 1 so use old_probs,
    m = i - 1
            )
    return abs(probs[n] - target_prob) < 1e-6

# Training problem 124
def f(s: str):
    return s + 'world' == 'Hello world'

# Training problem 125
def f(st: str, a="world", b="Hello world"):
    return st + a == b

# Training problem 126
def f(s: str, n=1000):
    return len(s) == n

# Training problem 127
def f(inds: List[int], s="hello world", target="do"):
    i, j, k = inds
    return s[i:j:k] == target

# Training problem 128
def f(s: str, big_str="foobar", index=2):
    return big_str.index(s) == index

# Training problem 129
def f(big_str: str, sub_str="foobar", index=2):
    return big_str.index(sub_str) == index

# Training problem 130
def f(s: str, a="hello", b="yellow", length=4):
    return len(s) == length and s in a and s in b

# Training problem 131
```

```
def f(substrings: List[str], s="hello", count=15):
    return len(substrings) == len(set(substrings)) >= count and all(sub
    in s for sub in substrings)

# Training problem 132
def f(string: str, substring="a", count=10, length=100):
    return string.count(substring) == count and len(string) == length

# Training problem 133
def f(x: str, parts=['I!!', '!love', 'dumplings', '!', ''], string="I
    !!!!!love!!dumplings!!!!!"):
    return x.join(parts) == string

# Training problem 134
def f(parts: List[str], sep="!!", string="I!!!!!love!!dumplings!!!!!"):
    return sep.join(parts) == string and all(sep not in p for p in parts)

# Training problem 135
def f(li: List[int], dups=42155):
    return len(set(li)) == len(li) - dups

# Training problem 136
def f(li: List[int], target=[17, 9, -1, 17, 9, -1], n=2):
    return li * n == target

# Training problem 137
def f(i: int, li=[17, 31, 91, 18, 42, 1, 9], target=18):
    return li[i] == target

# Training problem 138
def f(i: int, li=[17, 31, 91, 18, 42, 1, 9], target=91):
    return li[i] == target and i < 0

# Training problem 139
def f(inds: List[int], li=[42, 18, 21, 103, -2, 11], target=[-2, 21, 42]):

    i, j, k = inds
    return li[i:j:k] == target

# Training problem 140
def f(s: str, a=['cat', 'dot', 'bird'], b=['tree', 'fly', 'dot']):
    return s in a and s in b

# Training problem 141
def f(x: int, a=93252338):
    return -x == a

# Training problem 142
def f(x: int, a=1073258, b=72352549):
    return a + x == b

# Training problem 143
def f(x: int, a=-382, b=14546310):
    return x - a == b

# Training problem 144
def f(x: int, a=8665464, b=-93206):
    return a - x == b

# Training problem 145
def f(n: int, a=14302, b=5):
    return b * n + (a % b) == a

# Training problem 146
def f(n: int, a=3, b=23463462):
```

```
    return b // n == a

# Training problem 147
def f(n: int, a=345346363, b=10):
    return n // b == a

# Training problem 148
def f(x: int, a=10201202001):
    return x ** 2 == a

# Training problem 149
def f(x: float, a=1020):
    return abs(x ** 2 - a) < 10 ** -3

# Training problem 150
def f(x: float, a=1020):
    return abs(x ** 2 - a) < 10 ** -3 and x < 0

# Training problem 151
def f(s: str):
    return "Hello " + s == "Hello world"

# Training problem 152
def f(s: str):
    return "Hello " + s[::-1] == "Hello world"

# Training problem 153
def f(x: List[int]):
    return len(x) == 2 and sum(x) == 3

# Training problem 154
def f(s: List[str]):
    return len(set(s)) == 1000 and all((x.count("a") > x.count("b")) and
    ('b' in x) for x in s)

# Training problem 155
def f(n: int):
    return str(n * n).startswith("123456789")
```