

---

# Programming Puzzles

---

Tal Schuster  
MIT

Ashwin Kalyan  
Allen Inst. for AI

Oleksandr Polozov  
Microsoft Research

Adam Tauman Kalai  
Microsoft Research

## Abstract

We introduce a new type of programming challenge called programming *puzzles*, as an objective and comprehensive evaluation of program synthesis, and release an open-source dataset of Python Programming Puzzles (P3). Each puzzle is defined by a short Python program  $f$ , and the goal is to find an input  $x$  which makes  $f$  output True. The puzzles are objective in that each one is specified entirely by the source code of its verifier  $f$ , so evaluating  $f(x)$  is all that is needed to test a candidate solution  $x$ . They do not require an answer key or input/output examples, nor do they depend on natural language understanding. The dataset is comprehensive in that it spans problems of a range of difficulties and domains, ranging from trivial string manipulation problems that are immediately obvious to human programmers (but not necessarily to AI), to classic programming puzzles (e.g., Towers of Hanoi), to interview/competitive-programming problems (e.g., dynamic programming), to longstanding open problems in algorithms and mathematics (e.g., factoring). The objective nature of P3 readily supports self-supervised bootstrapping. We develop baseline enumerative program synthesis and GPT-3 solvers that are capable of solving easy puzzles—even without access to any reference solutions—by learning from their own past solutions. Based on a small user study, we find puzzle difficulty to correlate between human programmers and the baseline AI solvers.

## 1 Introduction

Puzzles are often used to teach and evaluate human programmers. Classic puzzles such as the Towers of Hanoi teach fundamental concepts such as recursion. Programming competition problems, also referred to as puzzles [29], evaluate a participant’s ability to apply these concepts to artificial tasks. Puzzles are also used to evaluate programmers in job interviews, and harder puzzles—such as the RSA-factoring challenge—can even test the limits of state-of-the-art algorithms. Each of these types of puzzles is described in its own format, often in a natural language such as English, and the solutions are evaluated through a variety of means.

We introduce a novel puzzle representation called a programming puzzle or simply a *puzzle*, which captures the essence of these puzzles in a form convenient for machines and programmers. A puzzle is specified by a short program  $f$ , and an *answer* is an input  $x$  which makes  $f$  output True, i.e., a valid answer  $x$  satisfies  $f(x) == \text{True}$ . Puzzles make for an objective programming evaluation based solely on source code with no need for any natural language descriptions, input/output examples, or reference solutions. We also release a growing open-source Python Programming Puzzles dataset,<sup>1</sup> called P3, which is already comprehensive in terms of difficulty, domain, and algorithmic tools. This dataset unifies many of the types of puzzles mentioned above.

To illustrate, `(lambda s: "Hello " + s == "Hello world")`<sup>2</sup> is an example of a Python programming puzzle, with the answer `"world"`. Some puzzles have multiple answers and some puzzles, even if very short, are extremely challenging. For example, Figure 1 illustrates three short puzzles

<sup>1</sup><https://github.com/microsoft/PythonProgrammingPuzzles>

<sup>2</sup>In Python, `lambda s` defines a function of variable `s` and string addition corresponds to string concatenation.

```

def f1(s: str): #find a string with 1000 o's but no consecutive o's.
    return s.count("o") == 1000 and s.count("oo") == 0

def f2(x: List[int]): #find the *indices* of the longest monotonic subsequence
    s = "Dynamic programming solves this classic job-interview puzzle!!!"
    return all(s[x[i]] <= s[x[i+1]] and x[i] < x[i+1] for i in range(25))

def f3(d: int): #find a non-trivial integer factor
    n = 100433627766186892221372630609062766858404681029709092356097
    return 0 < d < n and n % d == 0

```

Figure 1: Programming puzzles ranging from trivial to longstanding open algorithmic challenges in multiple domains. `f1("ox" * 1000) == True` where `s * n` means `n` repetitions of string `s`; dynamic programming efficiently finds a list of 26 increasing indexes to solve `f2` while brute force would take exponentially long; and `f3` requires advanced computational number theory algorithms.

that are diverse in domain, difficulty, and algorithmic tools. The first puzzle is an easy (for humans) puzzle that tests one’s understanding of basic properties of strings. The second one requires dynamic programming (typically taught in college-level programming courses), and the last is a hard problem requiring advanced computational number theory algorithms.

As describe in §3, the dataset also contains numerous classic puzzles; game-playing puzzles; optimization puzzles such as solving a linear programming; graph puzzles such as shortest path; and competitive-programming problems. The most difficult puzzles involve longstanding open problems such as learning parity with noise [8]; factoring [20, 26]; or finding a cycle in the  $3n+1$  process which would disprove the Collatz conjecture [30]. Thus, if AI were to surpasses human-level performance on this dataset, it would lead to breakthroughs on major open problems. The P3 puzzles were inspired by sources such as Wikipedia, algorithms books, and programming competitions. We plan to grow this dataset as an open-source project, and anyone can add a puzzle by simply writing a function `f`.

Puzzles allow *objective* evaluation, meaning that it is easy to test whether any candidate answer is valid without consulting an answer key. A typical IQ test problem may give a sequence of numbers and ask to predict the next. However, one can typically justify many different answers and thus one needs to learn how to reason like the problem authors. Similarly, in Programming by Example (PBE), there may be multiple functions mapping a given sequence of inputs  $x_i$  to outputs  $y_i$ . In an objective evaluation, there is no additional burden to learn the answer-key bias. Objective evaluation also makes bootstrapping easy, even on a set of test puzzles provided without any answers. Given a set of puzzles, one can attempt to solve them, determine with certainty which solutions are correct, and use those solutions to improve one’s ability to solve the remaining puzzles [15]. Inspired by success in playing games [41, 43], self-training has also proven useful in program synthesis [see, e.g., 3, 11].

From a theoretical point of view, as we shall discuss, objectivity can be formalized as the complexity class NP of non-deterministic polynomial-time decision problems. Moreover, the puzzle decision problem is NP-complete, meaning puzzles can readily express any NP problem, including polynomial-time problems and other NP-complete problems such as Boolean satisfiability.

Our experiments compare different parametric enumerative top-down solvers based on random forests and Transformers, and different types of GPT-3 prompts, e.g., zero/few-shot and with/without English descriptions. Without access to any reference solutions, only utilizing self-training bootstrapping, our enumerative models solved up to 43% more P3 problems than a naive brute force baseline.

To address the questions of whether puzzles measure programming proficiency and how puzzle difficulty compares between humans and computers, we performed a small user study. Puzzles were accessible and enjoyable for programmers with varying levels of experience. While both GPT-3 and enumerative techniques can solve a fraction of the puzzles, human programmers outperform them. For example, bootstrapping GPT-3 with up to 10K tries solved 60% of the puzzles, lower than both beginner and experienced participants that solved 76% and 87% puzzles on average, respectively. Overall, we find perceived puzzle difficulty to scale similarly for both humans and AI.

The main contributions of this paper are introducing: (a) programming puzzles: a new type of problem suitable for algorithmic problem-solving (for both machines and humans); (b) P3, an open-source dataset of puzzles covering diverse domains and difficulties; and (c) an evaluation of humans and baselines demonstrating that puzzles can be used to measure algorithmic problem-solving progress.

## 2 Problem formulation

Formally, both puzzles and answers can be represented as strings, where  $\Sigma^*$  is the set of finite strings over alphabet  $\Sigma$ . The set of puzzles is denoted by  $\mathcal{F} \subseteq \Sigma^*$ , with answers  $\mathcal{X} \subseteq \Sigma^*$ . A puzzle  $f \in \mathcal{F}$  is defined entirely by its source code string and, with a slight abuse of notation, the result of running it on input  $x \in \mathcal{X}$  is denoted as  $f(x) \in \{0, 1\}$ . Answer  $x \in \mathcal{X}$  is *valid* if it *satisfies*  $f(x) = 1$ , i.e.,  $f$  outputs 1 when run on  $x$ , within a specified amount of time. To ensure that puzzles can be *quickly* verified, it is necessary to upper-bound the time required for puzzle verification. This ensures that the puzzle decision problem, namely the problem of determining whether a given a puzzle has a valid answer, is in the complexity class NP. Thus formally, the puzzle decision problem is, given a string  $f$  denoting the puzzle (represented as, say, a Turing machine) and a timeout  $t$ , does the puzzle output 1 in time  $\leq t$ . See Appendix D for further details and comparison to other NP-complete problems.

A **solver** takes  $n$  puzzles  $f_1, f_2, \dots, f_n$  with timeouts  $t_1, \dots, t_n$ , and outputs answers to as many puzzles as it can within a time bound  $T$ . Of course  $T \gg \sum t_i$  is significantly larger than the verification timeouts. Formally, the *score* of solver  $S : \mathcal{F}^n \rightarrow \mathcal{X}^n$  is the number of puzzles  $f_i$  for which  $f_i(x_i)$  outputs 1 in time  $\leq t_i$ . Although we do not study it in this paper, it would be natural to assign different *values* to different puzzles. For example, solving open problems such as finding a Collatz cycle or factoring the largest RSA challenge integer (currently unsolved, with a \$200,000 prize offered), should be of greater value than solving a simple hello-world puzzle.

It is convenient, though not required, to solve puzzles by outputting a program which, when run, computes the answer. Such a program is called a **solution**  $g$  to distinguish it from the answer  $x$  that  $g$  outputs (i.e.,  $x=g()$ ). Puzzles may have short solutions but long answers, e.g., the puzzle  $f$ : `lambda s: len(s) == 10 ** 6` that asks for a string of length one million is solved by the solution program  $g$ : `'a' * (10 ** 6)`. In this example, solution  $g$  generates a valid answer  $x$  of length one million. Of course, another solution would be to explicitly write a string of length one million in the code, though this implementation may not pass a human code review. In the dataset and this paper, we provide solutions since they may be significantly shorter.

Many puzzles fit a single **problem** template. For example, the factoring puzzles for different  $n$  are all instances of the factoring problem (Figure 1). The parameters that vary—in this case,  $n$ —are called *puzzle parameters*. The parameters of a shortest-path problem might be the graph, the source and target nodes, and an upper bound on the path length. Many problems, such as the [Collatz problem](#), have no parameters and thus the Collatz problem consists of a single puzzle.

## 3 The P3 dataset

P3 uses Python, the de facto language of ML research, as the programming language for specifying puzzles. P3 is generated from the 200 problems summarized in Table 1 by running `make_dataset.py` in the repository with a maximum of 1,000 puzzles per problem. A larger number of puzzles may be generated by increasing this maximum. Every puzzle is described by a function with a single typed argument (i.e., the candidate answer) that returns True upon success. Since Python is not type-safe, we add an assertion to ensure that answers match the declared type.

We also provide code for serializing Python objects to and from strings in a json-like format, so that programs implemented in any language can output potential answers. Moreover, strings are universal in that they can encode arbitrary Python objects including functions, as in the [Quine](#) puzzle (`lambda quine: eval(quine) == quine`)<sup>3</sup> motivated by the classic challenge of writing a program that outputs its own source code. As evaluation of the string quine can lead to an infinite loop, this puzzle illustrates the necessity of the evaluation timeout  $t$  for attempted solutions.

While not necessary,<sup>4</sup> we follow the common practice of programming competitions and provide a reference solution to most (over 90%) of the puzzles. Some puzzles have more than one solution. A handful of puzzles represent major open problems in computer science and mathematics including [Factoring](#) (and [Discrete Log](#)), [Planted Clique](#), [Learning Parity with Noise](#), [Graph Isomorphism](#), and finding a [Collatz cycle](#), as described in Appendix E. We also provide English descriptions for

<sup>3</sup>GPT-3 generated a 5-character solution to the quine puzzle while the authors’ solution was 88-characters.

<sup>4</sup>Reference solutions are neither necessary nor used in our experiments, because puzzles are self-contained.

each puzzle in the dataset to support research involving natural language. Appendix F compares programming competition problems to puzzles.

**Creation process.** The following sources were used for identifying possible puzzles:

- Wikipedia, specifically the [Logic puzzles](#) category, the [List of unsolved problems in mathematics](#), and the [List of algorithms](#).
- Competitions, primarily the competitive programming website [codeforces.com](#) but also a handful of problems from the [International Collegiate Programming Contest](#) and the [International Mathematical Olympiad](#) (IMO)—a high school mathematics competition.
- The Python programming language itself, with trivial puzzles created to test understanding of basic functions, such as the the hello-world puzzle which tests string concatenation.

P3 is organized topically into files listed in Table 1. These topics include domains such as number theory, graph theory, chess puzzles, game theory, etc., as well as puzzles inspired by a specific source such as a specific programming competition. One finding in this paper is that many types of puzzles can be captured in spirit, if not exactly, as succinct puzzles. Common patterns include:

- Problems that are *naturally* puzzles. For instance, the [TowersOfHanoi](#) and [SlidingPuzzle](#) puzzles simply test the sequence of moves to see if they lead to the goal state.
- Problems that have an equivalent natural puzzle. For instance, the standard definition of the factoring problem, namely factorizing an integer into its prime factors would require a puzzle that tests primality. However the problem of [finding any non-trivial integer factor](#), f3 in Figure 1, can be recursively called to solve the prime factorization problem.
- Optimization problems. Some such problems have equivalent natural puzzles, e.g., linear programming is well-known [14] to be equivalent to solving a zero-sum game which is the [ZeroSum](#) puzzle. For others, such as [LongestMonotonicSubstring](#) (f2 of Figure 1) or [ShortestPath](#), we specify a bound  $\theta$  on the objective, and the goal is to find a feasible  $x$  with objective better than  $\theta$ . In order to generate  $\theta$ , we must solve the optimization problem ourselves, but the puzzle generation code is not provided to the solvers.
- Problems that ask how many items in a certain set satisfy a given property, may be converted to problems that require an explicit enumeration of all such items. See for example the [AllPandigitalSquares](#) puzzle that requires all 174 roots of pandigital perfect squares as input.
- Problems that involve game-playing can often be converted to puzzles. In chess, this includes the classic Eight Queens and Knights Tour puzzles. Our puzzles for the games of Nim and Mastermind involve exhibiting a winning strategy.

In order to ensure that each puzzle is achieving its goals, the puzzle design process has a step in which we test for unintended trivial solutions that are small integers or common strings.

**Exclusions.** Many programming challenges do not make as good puzzles. First, some challenges require a long program to describe. For instance, testing a Rubik’s cube solution requires significantly more code than the sliding 15 puzzle (which is partly why sliding puzzles are used so often in AI courses). Second, sometimes specifying the puzzle would give away the solution. For instance, consider finding the smallest prime number greater than  $10^6$ . This makes a reasonable contest problem with an answer key, but the puzzle form would include code to test primality which would give away the solution. Third, “soft” challenges involving natural language or images are not in NP and not easily verifiable. This includes challenges involving human common-sense knowledge about names, dates, or image classification. Finally, *interactive* programming challenges do not make for good puzzles. Fortunately, several other datasets cover these latter two types of exclusions.

**Growth process.** The focus of this paper is in creating a framework with an initial dataset; and demonstrating its utility for developing and evaluating AI solvers. As a GitHub repository, the dataset can grow over time in a standard manner with the ability to reference previous versions.<sup>5</sup> We plan to continue adding puzzles and hope that others will as well. The popular competitive-programming website [codeforces](#) was found to be a potential source of innumerable puzzles which we plan to use

<sup>5</sup>This paper’s version is <https://github.com/microsoft/PythonProgrammingPuzzles/tree/v0.1>

Table 1: Number of problems (and how many of them have at least one reference solution) per domain in P3 v0.1. Puzzles are generated by altering problem parameters. The right two columns show the average size of puzzles and solutions, measured by the number of nodes in the Python AST.

Domain	Problems	Solutions	Puzzles	$ f $	$ g $
Algebra	4	4	4000	98	173
Basic	21	21	21000	79	43
Chess	5	3	4858	277	186
Classic puzzles	22	22	11370	132	194
CodeForces	24	24	23025	111	67
Compression	3	3	3000	160	118
Conways game of life	2	1	2000	248	371
Game Theory	2	2	2000	417	506
Games	5	5	1006	223	189
Graphs	11	10	9002	151	151
ICPC	3	3	3000	433	663
IMO	6	6	5012	209	253
Lattices	2	2	2000	108	222
Number Theory	16	12	10762	66	68
Probability	5	5	5000	107	72
Study	30	30	30	65	21
Trivial inverse	34	33	32002	46	26
Tutorial	5	5	5	46	13
Total # / Average size	200	191	139,072	109	99

to grow the dataset. We found that almost all level-800 problems (their easiest problems) could be encoded as short puzzles.

## 4 Solvers

In this section, we describe the models we develop as baselines for the dataset. We consider both solving problems independently and joint solvers that bootstrap from previously obtained solutions to find new ones. We also consider both enumerative solvers that use standard techniques from program synthesis and a Language-Model (LM) solver that uses GPT-3 to solve puzzles. While a direct comparison between these two different approaches is difficult because they run on very different hardware (the LM calls an API), we can still compare the relative difficulty with which they solve different puzzles, and also to human difficulty rankings among puzzles.

### 4.1 Enumerative solvers

Following prior work [3, 11, 34], we develop models to guide the search for  $g$  over the space of all possible functions. In particular, we implement a grammar that generates Abstract Syntax Trees (ASTs) for a large subset of Python. The grammar covers basic Python functionality and is described in Appendix B.1. Specifically, each Python function is translated to an AST using a given set  $\mathcal{R}$  of rules. Based on the puzzle, a context-specific distribution over rule probabilities is computed. To facilitate efficient top-down search, the context of a rule is defined to be the rule used by the parent node and the index of the current node among the parent’s children. Thus if the parent node was a division binary operator, then the two children would each have different contexts, but if two such divisions were present in the same program, both numerators would share the same context.

Each puzzle  $f$  is represented by a feature vector  $\phi(f)$  and each context is represented by a vector  $c(p, i)$  where  $p$  is the parent rule and  $i$  is the child index. Each rule  $r \in \mathcal{R}$  is also associated with a feature vector  $\rho(r)$ . The probability distribution over  $\mathcal{R}$  is determined based on  $\rho(r)$ ,  $\phi(f)$ ,  $c(p, i)$ , and the likelihood of a solution  $g$  is the product of all rules constituting its AST. Naturally, this scoring mechanism introduces a bias towards shorter programs (i.e., smaller trees), which is desirable as a short solution is easy to inspect.

**COPY rules.** Solutions often reuse constants or puzzle parameters, for example the constant 25 or the variable `s` in example `f2` in Figure 1. As in prior work [34], for each puzzle, the global rules



bank is expanded to include COPY rules for constants and parameters of the examined puzzle.<sup>6</sup> When composing solutions, this rule can reduce the complexity of the solution by simply learning to copy part of the puzzle rather than having to generate it from scratch. For simplicity, we create copy rules for each of the supported types and assign the probabilities uniformly across all the puzzle’s constants of that type. In other words, our models learn when a certain type should be copied from the puzzle, and rank all available constants and parameters of that type the same.

To solve a new puzzle, we perform a top-down search. Specifically, at each node, we apply a selected model over all rules in  $\mathcal{R}$  whose type matches the context, and re-normalize the scores to create a valid probability distribution. The solver enumerates solutions in order of decreasing likelihood until it finds a solution  $g$  such that  $\mathbf{f}(g())$  evaluates to `True` in time  $\leq t$ , for a maximum number of tries  $M$ . See Appendix B for details on the search and rules. Next, we briefly describe our models.

**Uniform.** The first model is a simple uniform rule that assigns the same probability to all rules. The only exception is COPY rules, which have a larger, fixed probability in order to bias the solver towards utilizing this option. As we score programs by their joint probability, this bias effectively favors shorter programs. We use this model to find solutions to the easier problems, satisfied by a simple and short answer, and use these to bootstrap the learning of the parametric models. This model also provides a naive brute force baseline to compare the parametric models with, testing if they can guide the solution search better.

The remaining two models have parameters that are fit based on *bootstrapping*. Namely, given previously obtained solutions, we collect all parent-child rule pairs as self-supervision and fit the model’s parameters on them. The training size for this learning problem is the total number of nodes in all the trees among solutions discovered up until that point. We implement two bigram parametric models to predict  $\mathbb{P}(r \mid \rho(r), \phi(f), c(p, i))$ , where  $r$  is a candidate rule to appear in  $g$ ’s tree under  $p$  as its  $i$ ’s argument.

**Random forest.** In this model, we represent  $f$  as a bag-of-rules  $\cup\{r_k \in f\}$ . Specifically,  $\phi(f)$  is a vector of length  $|\mathcal{R}|$  representing the number of occurrences of each rule in  $f$ .  $p$  and  $i$  are encoded as a one-hot vector and concatenated to  $f$ ’s representation to construct the input to the model. Given past solution trees, we train the model to predict the index of  $r$  out of  $|\mathcal{R}|$  given  $f, p, i$  examples.

**Transformer.** Following the recent success of Transformer models [44] in encoding source code [17, 27, 42, *inter alia*], we turn to these encoders for richer representations. We use a BERT-based [16] Transformer to encode puzzles and rules directly from their code. The probability of a rule  $r$  being the  $i$ ’s child of  $p$  in  $g$  is proportional to the dot product of the deep joint representation of  $f, p, i$  and the Transformer encoding  $\rho(r)$ . We pretrain the Transformer with a masked language model task on Python GitHub repositories [23]. Then, our solver concatenates the Transformer encodings  $\phi(f)$  and  $\rho(p)$  with a learned embedding for  $i$ , following by non-linear layers to compute the joint representation. We fine-tune the solver on parent-child rule pairs from previously acquired solutions. See Appendix B.2 for extended details, and Figure B.1 on page 17 for a model diagram.

## 4.2 Autoregressive language model solvers

We experiment with the massive Transformer-based GPT-3 language model [9]. We follow the recent strategy of using GPT-3 by designing a prompt that directs the text generation to our desired task. This approach has recently shown to be useful in converting natural language descriptions to programming code [22] and guide theorem proving [38]. Interestingly, our prompts are mostly programs. Unlike our enumerative models that build an AST, GPT-3 generates the solution as a string that is directly evaluated as Python code.

We consider four different prompts: (a) A *short* zero-shot prompt based solely on the puzzle at hand (illustrated in Figure 2); (b) a *medium* 5-shot prompt that includes the five example puzzles that had been shown to (human) programmers during our user study (Figure C.1 on page 18); (c) a *long* prompt with the same five puzzles augmented by English descriptions of the tasks in comments (Figure C.2 on page 19); and (d) a *bootstrapping* prompt which uses only solutions to problems that it has already solved (Figure C.3 on page 20). The bootstrapping prompt begins like the zero-shot

<sup>6</sup>When executing a solution, COPY rules are simply the identity function (COPY = `lambda x: x` in Python).

```
def f(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

assert True == f(...
```

Figure 2: A short prompt for a puzzle requesting a list of ten integers where the fourth item occurs exactly twice. A valid completion would be `...[1, 2, 3, 4, 5] * 2)`.

prompt but quickly exceeds GPT-3’s API maximum length as more puzzles are solved. At that point, previously solved puzzles are randomly sampled to form the prompt.

The completions which parse as valid Python expressions are then evaluated. Appendix C gives further details of the execution environment, the API parameters and other prompts we investigated.

## 5 Experiments

We use our P3 dataset to evaluate the performance of the solvers from §4. We assume no access to reference solutions<sup>7</sup> and measure how many puzzles are solved by each solver with up to  $M$  tries per puzzle, where each try is a potential solution that is evaluated. For the enumerative solvers, this is equivalent to having a valid solution ranked in the top  $M$ . For autoregressive solvers, it reflects the probability of obtaining a solution within  $M$  outputs. First, we test the solvers bootstrapping efficacy in leveraging previously solved problems to solve new ones. Then, once solutions to a single instance of certain problems are found, we test whether solvers also succeed on other problem instances (i.e., puzzles originated from the same problem). Finally, in §5.1, we present our user study results that compares human’s performance with AI solvers.

**Learning from past solutions.** In this experiment, we use a single puzzle instance per problem. For the parametric enumerative solvers, we first run the uniform solver with  $M = 10^4$  on all 138 problems supported by our grammar (see Appendix B.1), solving 38 of them. These solutions contain a total of 2,475 rules that we use to train the parametric models. In the bootstrapping variant, we repeat the training for 6 cycles, each time adding the new solutions found with  $M = 10^4$ . In the final round, we allow up to  $M = 10^6$  solving tries (including tries from previous cycles). The Bootstrapping GPT-3 starts with the zero-shot (short) prompt and appends to it valid solutions as they are found. Table 2 shows the number of achieved solutions per domain with  $M$  tries.

Figure 3a shows the number of puzzles solved by each enumerative solver, with and without the self-training bootstrapping cycles. We see that the parametric models quickly improve over the naive uniform search and that the bootstrapping process facilitates solving many new problems. At  $M = 10^6$ , the random forest and Transformer-based enumerative models solved a total of 73 and 76 problems, respectively, which is 38% and 43% more than the uniform solver.

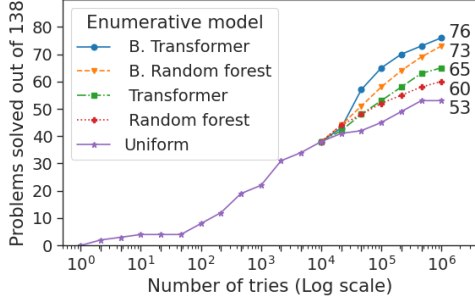
The GPT-3 solver also improves by learning from previously found solutions. As Figure 3b shows, few-shot settings with tutorial examples perform better than zero-shot (Short) and solve new problems. Including natural language descriptions (Long) helps for solving five more puzzles, with up to  $10^4$  tries. The best strategy, however, is the bootstrapping one that starts without any reference and adds solutions to the prompt as they are found. Appendix H presents the 86 solutions found by this model.

**Generalizing to other problem instances.** In the previous experiment, we attempted to solve the *default* single puzzle instance of each problem. Next, we examine whether our solvers can also solve other puzzle instances, originating from the same problems. We collect a set of 700 puzzles that are random instances of 35 problems for which both our bootstrapping enumerative models solved the default puzzle. At  $M = 10^4$ , the random forest and Transformer models solved 75% and 79%, respectively. As a reference, the uniform model solves only 62% of these puzzles.

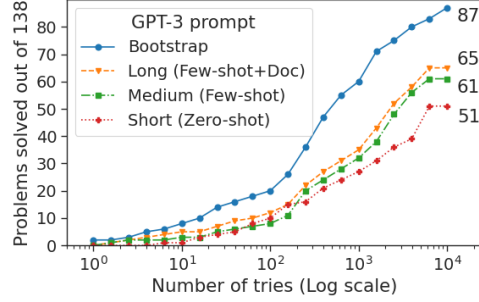
### 5.1 User study

In a small user study, 21 participants with varying experience in Python programming completed 30 puzzles. Each puzzle was allocated a maximum of 6 minutes to solve, and the study was conducted virtually using Jupyter notebooks. Participants were employees at a major software company and

<sup>7</sup>With the exception of the GPT-3 medium-long prompts that including five Tutorial problems and solutions.



(a) Enumerative solvers.



(b) GPT-3 solvers.

Figure 3: Increasing the number of tries allows solving new problems. Better solvers, though, solve new problems significantly faster by learning from past experience. (a) Parametric enumerative solvers initialized with the solutions of the uniform solver at  $M = 10^4$  accelerate the solution search. Additional self-training bootstrapping cycles (marked with B.) solve even more problems. (b) Due to limited budget, GPT-3 solvers are evaluated with up to  $10^4$  attempts. The natural language description (Long) allows small improvement. Adding previously found solutions to the prompt (Bootstrap) provides significant improvement and solves the most puzzles across baselines.

Table 2: Solved problems per domain with up to 1M tries per puzzle for enumerative and 10K for GPT-3. The first row also shows the number of available P3 problems in that domain. Bootstrapping models, that learn from new solutions as they are found, are in grayed lines.

Model	Algebra	Basic	Chess	Classic	CodeForces	Compression	Conways	Game Theory	Games
Uniform	0/4	7/21	0/5	1/22	7/24	0/3	0/2	0/2	0/5
Random forest	0	10	0	4	6	0	0	0	0
B. Random forest	0	13	0	5	7	0	0	0	0
Transformer	0	9	0	4	7	0	0	0	0
B. Transformer	0	13	0	3	7	0	0	0	0
GPT-3 (Medium)	0	6	0	1	7	0	0	0	1
B. GPT-3	0	13	0	2	7	0	0	0	1

Model	Graphs	ICPC	IMO	Lattices	Number Theory	Probability	Study	Trivial inverse	Tutorial
Uniform	1/11	0/3	0/6	0/2	3/16	1/5	9/30	22/34	2/5
Random forest	2	0	0	0	4	1	7	23	3
B. Random forest	2	0	0	0	6	1	10	26	3
Transformer	1	0	0	0	6	1	10	24	3
B. Transformer	3	0	0	0	6	1	14	26	3
GPT-3 (Medium)	4	0	0	0	1	1	17	18	5
B. GPT-3	6	0	0	0	5	1	18	30	4

were recruited by email and at a hackathon. No compensation was offered. Participants were first given a short tutorial about puzzles and how to submit solutions. The user study files are available in the open-source dataset, and further details (including the 30 puzzles) can be found in Appendix G.

The first finding is that success in puzzles correlates with programming experience. For our retrospective study analysis, we split the participants by the median years of Python programming experience. We had 10 *beginners* with less than three years of experience, and 11 *experienced* participants with at least three years. We find that 9 of the 30 puzzles were solved by all beginners, while 17 of the puzzles were solved by all experienced participants. Also, beginners spent on average 194 seconds per puzzle, while experienced spent only 149 seconds on average. The average solving time provides a useful proxy to the perceived difficulty of each puzzle. Overall, we see that puzzles are easier for experienced programmers, indicating their value for evaluating programming proficiency.

The second finding is that experienced human programmers outperformed our AI baselines. There were over 10 puzzles that none of our baselines solved, while the average number of puzzles solved by experienced programmers was greater than 25, and one participant solved all 30. However, bear in mind that these results are with a 6 minute limit. No human would solve the puzzles within seconds, and we would expect experienced programmers to eventually solve virtually all study puzzles.

Finally, we find that difficult puzzles for humans are also harder for AI. Figure 4 shows that most of the puzzles solved by AI solvers are the ones that are easier for humans (i.e., solved faster). To compare



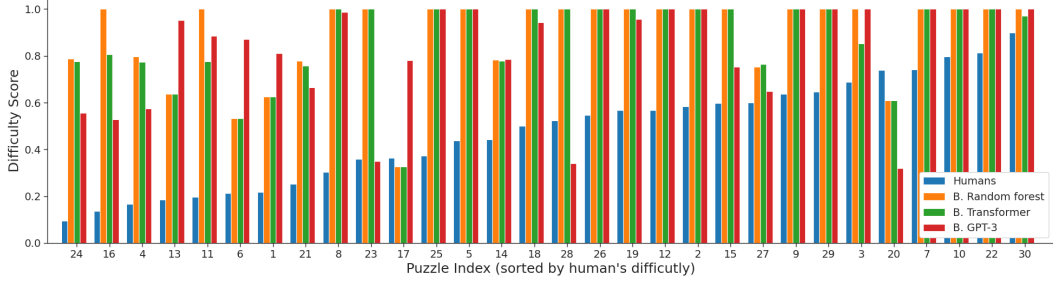


Figure 4: The difficulty score per study puzzle for both humans and AI solvers, sorted by the human’s scores. The difficulty score for humans is measured by the average fraction of solving time out of the maximum allowed. For AI, the number of required attempts is used. and of attempts for AI. Most of the puzzles solved by AI (difficulty  $< 1$ ) are also easier for humans (left hand side of the plot).

the two, we define a puzzle’s perceived difficulty score as the average solving time for humans and the number of required tries for machines. For unsolved puzzles, we set the difficulty to the maximum value, and then normalize to  $[0, 1]$ . The results show a strong correlation between the two. Specifically, the Spearman’s rank coefficient of humans with B. Transformer is 0.443, and with B. GPT-3 is 0.512. The correlation is even higher (0.493 and 0.541, respectively) when comparing only to beginner programmers. On the one hand, this suggests that additional computational power might allow AI solvers to match humans. However, as Figure 3 shows, this improvement is logarithmic, leading to diminishing returns. Encouragingly, we see that carefully designed solvers can utilize past experience to significantly reduce the required attempts. We hope that our puzzles will support the research and development of new AI solvers that will solve more puzzles with less computational effort.

## 6 Related Work

Program synthesis problems take drastically different forms for different applications, often resulting in one-off evaluations rather than common datasets. A major paradigm is Programming by Example (PBE) where problems are specified by input-output examples. For instance, several studies focus on text processing [18] or robot navigation [36]. While convenient for end user applications (e.g., many in [37]), PBE alone is inadequate to objectively describe most algorithmic programming challenges. A recent ARC dataset [10] adopts PBE for evaluating abstraction and reasoning in AI, but, like in all PBE applications, the examples’ inherent ambiguity makes the evaluation non-objective.

English descriptions, often mixed with examples, are becoming an increasingly popular problem representation as LMs improve [24, 28, 46]. In independent work, Hendrycks et al. [22] created a large dataset of English programming problems with examples on which they fine-tuned GPT models. This representation is more natural for people, but conflates understanding of natural language with programming ability (see Appendix F for a comparison of English competition problems and out puzzles). Several neighboring fields that have made substantial progress in reasoning include theorem proving [5], two-player game playing [41], and SAT-solving [6]. In all of these fields, important progress has been made by encoding the problems, be they theorems, game rules, or optimization problems, in machine-readable formats that do not involve the ambiguities of natural language.

The last paradigm, program synthesis from formal specifications, has a long history of study. See Gulwani et al. [19] for a survey of methods, benchmarked by e.g., the SyGuS competition [1]. In this setting, however, the AI system has to synthesize an algorithm that correctly and efficiently solves a problem on all inputs (and often prove correctness as well), and writing and testing the formal specification is often non-trivial.

To our knowledge, our work has the first *controlled evaluation* of GPT-3 for synthesizing programs, though it has been studied carefully for other purposes [9]. Prototypes and deployed applications of GPT-3 driven code generation exist [31, 40] but they lack public systematic evaluation. In concurrent work [22], GPT-3 was found to perform poorly at the task of synthesizing competitive-programming problem solutions without fine-tuning. However, with fine-tuning, GPT-based networks did solve such challenges. Also, compared to many domain-specific languages used in other work on program synthesis, Python is a significantly more complex language, though it is much simpler than English.

## 7 Conclusions

We introduce Python Programming Puzzles (P3), an open-source dataset with puzzles described only in source code. As discussed in §3, the puzzle framework is limited to capturing only NP challenges. Yet, puzzles cover a wide range of interesting challenges, and allow fast and objective evaluation, thereby supporting test-time bootstrapping. We implemented and evaluated several enumerative program-synthesis and autoregressive baselines, and found a positive correlation between their per-puzzle performance and the difficulty for human programmers. Also, we found bootstrapping using past solutions to improve performance more than simply using additional computational effort. In future work, we plan to use and extend the dataset ourselves, and we hope others will as well.

**Acknowledgments.** We would like to thank Mariia Mykhailova for suggesting doing a Python Programming Puzzles Hackathon. We are especially grateful to the participants in our user study and hackathon. We are grateful to the creators of GPT-3 and to Nicolás Fusi for suggesting using it. We would like to thank David Alvarez Melis and Alec Helbing for suggesting quine puzzles. We are grateful to Ana-Roxana Pop for helpful discussions and feedback. We also thank Tianxiao Shen and the rest of the MIT NLP group members for valuable feedback on the writing.

## References

- [1] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. SyGuS-Comp 2018: Results and analysis. 2019.
- [2] Sanjeev Arora and B. Barak. Computational complexity: A modern approach. 2009.
- [3] 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.
- [4] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978. doi: 10.1109/TIT.1978.1055873.
- [5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2004.
- [6] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
- [7] N. Biggs, P.M.L.S.E.N.L. Biggs, London Mathematical Society, London Mathematical Society lecture note series, and S.P.G.N.J. Hitchin. *Finite Groups of Automorphisms: Course Given at the University of Southampton, October-December 1969*. Lecture note series. Cambridge University Press, 1971. ISBN 9780521082150. URL <https://books.google.com/books?id=f1A4AAAAIAAJ>.
- [8] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, July 2003. ISSN 0004-5411. doi: 10.1145/792538.792543. URL <https://doi.org/10.1145/792538.792543>.
- [9] 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, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfbcb4967418bfb8ac142f64a-Paper.pdf>.
- [10] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

- [11] Konstantina Christakopoulou and Adam Tauman Kalai. Glass-box program synthesis: A machine learning approach. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [12] John Horton Conway. Five \$1,000 problems (update 2017). 2017. URL <https://oeis.org/A248380/a248380.pdf>. [Online; accessed 12/15/2020].
- [13] Valentina Dagienė and Gerald Futschek. Bebras international contest on informatics and computer literacy: Criteria for good tasks. In *International conference on informatics in secondary schools-evolution and perspectives*, pages 19–30. Springer, 2008.
- [14] George B Dantzig. A proof of the equivalence of the programming problem and the game problem. *Activity analysis of production and allocation*, 13:330–338, 1951.
- [15] Eyal Dechter, Jonathan Malmaud, Ryan P Adams, and Joshua B Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, pages 1302–1309, 2013.
- [16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL <https://www.aclweb.org/anthology/N19-1423>.
- [17] 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*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL <https://www.aclweb.org/anthology/2020.findings-emnlp.139>.
- [18] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, page 317–330, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926423. URL <https://doi.org/10.1145/1926385.1926423>.
- [19] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.
- [20] G. H. Hardy, E. M. Wright, D. R. Heath-Brown, and Joseph H. Silverman. *An introduction to the theory of numbers*. Oxford University Press, Oxford; New York, 2008. ISBN 9780199219858 0199219850 9780199219865 0199219869.
- [21] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). 2016.
- [22] 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.
- [23] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- [24] Alexander Skidanov Illia Polosukhin. Neural program search: Solving data processing tasks from description and examples. In *ICLR Workshop Acceptance Decision*, 2018. URL <https://openreview.net/forum?id=B1KJJf-R->.
- [25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.
- [26] Burt Kaliski. *RSA factoring challenge*, pages 531–532. Springer US, Boston, MA, 2005. ISBN 978-0-387-23483-0. doi: 10.1007/0-387-23483-7\_362. URL [https://doi.org/10.1007/0-387-23483-7\\_362](https://doi.org/10.1007/0-387-23483-7_362).

- [27] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.
- [28] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. SPoC: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL <https://proceedings.neurips.cc/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf>.
- [29] A. Laaksonen. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Undergraduate Topics in Computer Science. Springer International Publishing, 2020. ISBN 9783030393571. URL <https://books.google.com/books?id=3JbiDwAAQBAJ>.
- [30] Jeffrey C. Lagarias. The  $3x + 1$  problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985. ISSN 00029890, 19300972. URL <http://www.jstor.org/stable/2322189>.
- [31] Jennifer Langston. From conversation to code: Microsoft introduces its first product features powered by GPT-3, May 2021. URL <https://blogs.microsoft.com/ai/from-conversation-to-code-microsoft-introduces-its-first-product-features-powered-by-gpt-3/>
- [32] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 2019.
- [33] C. Mayer. *Python One-Liners: Write Concise, Eloquent Python Like a Professional*. No Starch Press, Incorporated, 2020. ISBN 9781718500501. URL <https://books.google.com/books?id=jVv6DwAAQBAJ>.
- [34] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.
- [35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [36] Chris Piech and Eric Roberts. Karel the robot learns python. <https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html>, 2020. [Online; accessed 07-Jun-2021].
- [37] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.
- [38] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL <https://arxiv.org/abs/2009.03393>.
- [39] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/schuster>.
- [40] Sharif Shameem. Tweet, 2020. URL <https://twitter.com/sharifshameem/status/1282676454690451457>. [Online; accessed 15-May-2021].
- [41] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

- [42] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. IntelliCode Compose: code generation using Transformers. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov 2020. doi: 10.1145/3368089.3417058. URL <http://dx.doi.org/10.1145/3368089.3417058>.
- [43] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.
- [44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.
- [45] 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*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- [46] Maksym Zavershynskiy, Alexander Skidanov, and Illia Polosukhin. NAPS: natural program synthesis dataset. *CoRR*, abs/1807.03168, 2018. URL <http://arxiv.org/abs/1807.03168>.



## A Example solutions by enumerative models

We provide examples of our examined enumerative solvers to three P3 puzzle in Figure A.1 on page 15 (Examples of GPT-3 solutions are given in Appendix H). The solution to the first puzzle is general and will work for any other instance of this problem. For the two other puzzles, the obtained solutions are instance-specific and don't even use the input variables. Yet, it is possible that the logical steps to achieve the answer are implicitly executed by the model. To test this, we evaluate the solvers on other problem instances (i.e., puzzles originated from the same problem).

The solvers' solutions to the first puzzle in Figure A.1 are simpler than the one created by humans (though less efficient in terms of input length). This illustrates another potential use case of AI solvers: debugging puzzles by finding easy solutions.

## B Enumerative solvers details

We train our random forest solver with the Python Skikit-learn library [35]. The Transformer model is implemented on top of the Hugging Face repository [45]. We use GPUs for training the Transformer and for querying it for rule probabilities. All other computations are performed with CPUs. Making up to  $10^4$  solution tries takes only a few seconds to a few tens of seconds, depending on the puzzle and the attempted solutions. Running up to  $10^6$  solution tries usually takes less than an hour but for some puzzles can take longer. We run the solver in parallel on multiple puzzles to reduce the global computation time.

**Solution validation.** Given the AST, a solution is generated in the form of a Python program (possibly multiple lines) that is evaluated by the Python interpreter to get an answer that is tested by the puzzle. To address long-running programs and infinite loops, timeout checks are added to the puzzles and to the solution during conversion from AST to Python. Alternatively, the programs could be evaluated in a sandbox as is done in programming competitions and as we did for the LM generators, though a sandbox imposes an additional overhead.

### B.1 Vocabulary

We use a grammar for a subset of Python covering the following basic objects: Booleans, unlimited-precision integers, floats, strings, lists, sets, dictionaries, generators, and tuples. Table B.1 summarizes the grammar. These rules occur multiply, for instance the addition rule has instantiations for adding two strings, two integers, an integer and a float, etc., where each Python type corresponds to a non-terminal in our grammar. However, because Python is a duck-typed language, in several cases a variable can be used with multiple different types. To handle such programs, we also have a generic non-terminal which can correspond to any Python object, and this makes our grammar ambiguous. For instance, the program `1+1` can be parsed either as the sum of two integers or as the sum of two Python objects, also using a rule mapping an object to an integer. This latter program is a larger AST and hence will typically have lower probability, hence we have the advantages of types when possible but the flexibility to generate fully duck-typed code. In this manner we are able to parse puzzles from 138 of our 200 problems. We also use this grammar to generate timed and safe Python code. In particular, we inject timing checks into comprehensions and loops, and we also add timing checks to potentially time-consuming operations such as exponentiation or string multiplication. This grammar is available upon request for researchers who wish to use it in further projects.

### B.2 Transformer implementation

We use the RoBERTa-base 12-layers Transformer [32] pretrained on English text and fine-tune it on Python code using the Hugging Face library [45]. For fine-tuning data, we use Python functions with their documentation text from GitHub repositories [23]. In order to better adjust the tokenizer to Python code, we retrain a Byte-level BPE tokenizer on our Python fine-tuning data. We use the same vocabulary size as the original tokenizer and keep the token embeddings of the overlapping ones (39%). For the other tokens, we initialize new token embeddings. Thereafter, we fine-tune RoBERTa with a masked language modeling task for 30 epochs. This model, which we denote by  $T_P$ , achieved an impressive 3.3 perplexity score on held-out evaluation data, indicating its success in learning Python's syntax.

```

# Sum of digits.
def sat1(x: str, s: int=679):
    return s == sum([int(d) for d in x])

# B. Random forest solution.
def sol(s):
    return ((chr(49))*(COPY(s)))

# B. Transformer solution.
def sol(s):
    return ((COPY(s))*(str(1)))

# Human-written solution.
def sol(s):
    return int(s/9) * '9' + str(s%9)

----

# Line intersection.
def sat2(e: List[int], a: int=2, b: int=-1, c: int=1, d: int=2021):
    x = e[0] / e[1]
    return abs(a * x + b - c * x - d) < 10 ** -5

# B. Random forest and B. Transformer solution (identical).
def sol(a, b, c, d):
    return ([2022, 1, ])

# Human-written solution.
def sol(a, b, c, d):
    return [d - b, a - c]

---

# Find the three slice indices that give the specific target in string s.
def sat3(inds: List[int], s: str="hello world", target: str="do"):
    i, j, k = inds
    return s[i:j:k] == target

# B. Random forest solution.
def sol(s, target):
    return ([12, 5, -(3), ])

# B. Transformer solution.
def sol(s, target):
    return ([11, 1, -(6), ])

# Human-written solution.
def sol(s, target):
    from itertools import product
    for i, j, k in product(range(-len(s) - 1, len(s) + 1), repeat=3):
        try:
            if s[i:j:k] == target:
                return [i, j, k]
        except (IndexError, ValueError):
            pass

```

Figure A.1: Example of three P3 puzzles and the solutions found by our examined solvers. The natural language description of each problem is provided for ease of read, but is hidden to these models. Human-written solutions are provided here for reference, but are also hidden from AI solvers.

Table B.1: The grammar for a subset of Python.

Rule name	rule	Rule name	rule	Rule name	rule
!=	(_) != ( _ )	[list]	[ _ ]	is not	(_) is not ( _ )
&	(_) & ( _ )	%	(_) % ( _ )	issubset	(_) .issubset ( _ )
(tuple)	( _ , _ )	{set}	{ _ }	issuperset	(_) .issuperset ( _ )
(tuple)	( _ , _ , _ )	^	(_) ^ ( _ )	join	(_) .join ( _ )
(tuple)	( _ , _ , _ , _ )	abs	abs ( _ )	len	len ( _ )
*	(_) * ( _ )	all	all ( _ )	list	list ( _ )
**	(_) ** ( _ )	and	(_) and ( _ )	log	log ( _ )
**=	(_) **= ( _ )	any	any ( _ )	max	max ( _ )
**=	_ **= ( _ )	append	(_) .append ( _ )	min	min ( _ )
*args	* _	arg	_ , _	not	not ( _ )
*args	* _ , ** _	arg	_ : _ , _	not in	(_) not in ( _ )
+	(_) + ( _ )	assert	assert _	or	(_) or ( _ )
+=	(_) += ( _ )	assert	assert _ , _	ord	ord ( _ )
+=	_ += ( _ )	bool	bool ( _ )	range	range ( _ )
+unary	+( _ )	chr	chr ( _ )	range	range ( _ , _ )
-	(_) - ( _ )	cos	cos ( _ )	range	range ( _ , _ , _ )
-=	(_) -= ( _ )	count	(_) .count ( _ )	replace	(_) .replace ( _ , _ )
-unary	-( _ )	def	def _ ( _ ) : _	return	return ( _ )
/	(_) / ( _ )	def_ANY_tuple	( _ )	reversed	reversed ( _ )
//	(_) // ( _ )	default_arg	_ : _ = , _	revsorted	sorted ( _ , reverse=True )
//=	(_) // = ( _ )	default_arg	_ = , _	round	round ( _ )
:slice	_ : _ : _	endswith	(_) .endswith ( _ )	round	round ( _ , _ )
<	(_) < ( _ )	exp	exp ( _ )	set	set ( _ )
<<	(_) << ( _ )	f_string	f ' _ '	sin	sin ( _ )
<=	(_) <= ( _ )	float	float ( _ )	sorted	sorted ( _ )
=	(_) = ( _ )	float-const	_ =	split	(_) .split ( _ )
==	(_) == ( _ )	float-const-large	_ = _ e	split	(_) .split ( )
>	(_) > ( _ )	float-const-tiny	_ = _ e -	startswith	(_) .startswith ( _ )
>=	(_) >= ( _ )	for	for ( _ ) in ( _ ) : _	str	str ( _ )
COPY	COPY ( _ )	for	for ( _ , _ ) in ( _ ) : _	str-const	" _ "
[-1]	(_) [-1]	for_in_if	for _ in ( _ ) if _	sum	sum ( _ )
[-2]	(_) [-2]	formatted_value	{ _ : _ }	tuple	tuple ( _ )
[-3]	(_) [-3]	if	if _ : _	type	type ( _ )
[-4]	(_) [-4]	if	if _ : _ else : _	union	(_) .union ( _ )
[0]	(_) [0]	ifExp	(_) if ( _ ) else ( _ )	zip	zip ( _ , _ )
[1]	(_) [1]	in	(_) in ( _ )	zip	zip ( _ , _ , _ )
[2]	(_) [2]	index	(_) .index ( _ )		(_)   ( _ )
[3]	(_) [3]	int	int ( _ )		
[i]	(_) [ _ ]	is	(_) is ( _ )		

Next, we use  $T_P$  to encode dense embeddings  $e_r = T_P(r)$  for all the rules  $r$  in our vocabulary  $\mathcal{R}$ . As input to the Transformer, we use a string representation of the Python operation and types of each rule. For example,  $(x)/(y)$  , `' // -> FLOAT :: (x: INT, y: FLOAT)` is used to describe the rule for the `//` operation with an integer and float inputs, resulting in a float. Then, we take  $e_r$  as the average across the top-layer embeddings of all tokens.

Finally, we design a neural model on top of  $T_P$  to predict  $\mathbb{P}(r_j|\phi(f), p, i)$  for each puzzle  $f$  where  $p$  is the parent rule and  $i$  is the child index. The model computes a hidden representation of the puzzle with the parent rule as a concatenation  $h = [T_P(f), \mathbf{W}_1 e_p, e_i] \in \mathbb{R}^{d+d_r+d_i}$ , where  $e_i \in \mathbb{R}^{d_i}$  is a learned embedding for the child rule index,  $\mathbf{W}_1 \in \mathbb{R}^{d_r}$  is a learned linear projection, and  $d$  is the hidden dimension of  $T_P$ . To obtain  $\phi(f)$ , we duplicate  $T_P$  and further fine-tune it with the rest of the solver parameters, while keeping the rule Transformer fixed as  $T_P$ . Specifically, we use the [CLS] embedding of the top layer as  $\phi(f)$ . Fixing the rule encoder prevents overfitting to the rules seen in the puzzle-solution fine-tuning pairs.  $h$  is then passed through two non-linear layers, where the first also projects it to  $\mathbb{R}^{d_r}$ , with a gelu activation [21] and batch normalization [25] to get a joint puzzle and parent rule embedding  $e_{f,p,i}$ . The score of rule  $r_j$  then being the  $i$ 's argument of  $r$  in the solution to  $f$  is determined by the dot product of its projected embedding  $e_{r_j}$  with the parent's embedding:  $p_{r_j|\phi(f), e_p, e_i} \propto e_{f,p,i} \cdot (\mathbf{W}_2 e_{r_j})^T$ . Similar to the Random Forest fitting process, we use all parent-child rule pairs from the previously obtained solutions for fine-tuning. We use cross-entropy loss with an Adam optimizer. See Figure B.1 for a model diagram.

## C GPT-3 solver details

The GPT-3 API was used to generate completions based on prompts. The completions were generated in batches of  $n=32$  with `temp=0.9`, for a maximum of 150 tokens, stopping at the end of a new line,

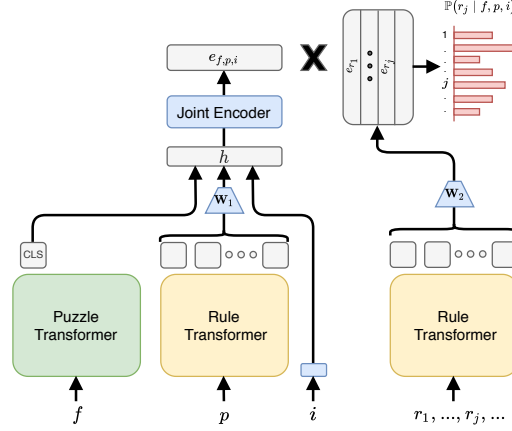


Figure B.1: An illustration of our Transformer-based enumerative solver. The rule strings are encoded with a Transformer pretrained on Python code. The puzzle Transformer is initialized the same, but is further fine-tuned on puzzles, together with the rest of the solver’s parameters shown in blue color. The left hand side of a diagram represents the encoding of the puzzle  $f$ , parent rule  $p$ , and child index  $i$ , each separately and then combined to a joint representation  $e_{f,p,i}$ . All rules  $r \in \mathcal{R}$  are also encoded with the Transformer and projected to the same dimension as  $e_{f,p,i}$ . The output probability of  $r$  being the  $i$ ’s child of  $p$  in the solution tree  $g$  to puzzle  $f$  is computed by a softmax over the product of  $e_{f,p,i}$  with all rule representations. Encoding the puzzle and the parent rule first separately, allows passing the puzzle only once during inference, and computing all rule embeddings in advance.

with default values of `top_p=1`, `presence_penalty=0`, `frequency_penalty=0`, and `best_of=1`. The resulting programs were evaluated in a sandbox limited to 1 second on Intel Xeon Platinum 8272CL CPUs at 2.60GHz. The timeout was necessary since a number of solution generators would take prohibitive resources such as `"a"*(10**(100))` which would generate a string of length googol. The solutions were also checked to be of the type requested in the problem, as was the case for the top-down solver. Without this additional requirement, for instance the puzzle f2 from Figure 1 would admit the solution `["o"] * 1000` which is invalid as it is a list of strings, though it does satisfy the puzzle as stated.

**Prompt programming.** The space of possible prompts is practically boundless. Our current prompt designs leverage the API without fine-tuning, but they are limited to one-line Python solutions. One benefit of single-line solutions is that they are short and we can use the API feature of stopping on a newline. We also experimented with prompts that would support multi-line solutions. While GPT-3 generated valid multi-line solutions, in our experiments there was not a single case in which a problem was solved with a multi-line solution which was not also solved by GPT-3 with a single-line solution. One-line Python programs are considered, by some, to be a useful form of programming with books dedicated to the topic [see, e.g., 33]. In future work, it will be exciting to see how large fine-tuned models can generate longer programs, but for now even one line programs pose an interesting challenge. As the [python.org](https://www.python.org) wiki states, “If you cannot read and write one-liner code snippets, how can you ever hope to read and write more complicated codebases? Python one-liners can be just as powerful as a long and tedious program written in another language designed to do the same thing.”

Numerous other prompts were considered. For instance, we tried adding a preface stating, “A Programming Puzzle is a short python function, and the goal is to find an input such that the function True. In other words, if program computes a function  $f$ , then the goal is to find  $x$  such that  $f(x)=\text{True}$ .” We also had prompts which encouraged the system to use variable names rather than having to copy the puzzle constants. The gains from such alternatives did not seem to be substantial, thus we chose the simpler prompt designs.

Interestingly, a handful of generations included potentially dangerous commands such as `eval` and `__import__("os").system(...)`, but a cursory inspection did not detect any that used them in malicious ways. We do advise caution in executing generated code, as malicious actors can

take advantage of such routine [39]. There are several libraries available for scoring programming competitions that serve this exact purpose. Also, some of the generated code seemed especially human-like, e.g.: `raise RuntimeError("this is a joke.")` which of course did not solve the puzzle at hand.

Figures 2, C.1, C.2, and C.3 show our prompts for the short, medium, long, and bootstrap prompts, respectively.

**Smoothing evaluation.** Rather than simply generating solutions until the first correct one is found, to evaluate the short, medium and long prompts, we generate 10,000 solutions for each puzzle. This gives us more than one solution for some puzzles, which we use for improved accuracy in estimating how many solutions are necessary (on average) to solve each puzzle shown in Figure 3b.

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

assert True == f1("world")

---

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

assert True == f2("world"[::-1])

---

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

assert True == f3([1, 2])

---

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)

assert True == f4(["a" * (i + 2) + "b" for i in range(1000)])

---

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

assert True == f5(int(int("123456789" + "0"*9) ** 0.5) + 1)

---

def f6(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

assert True == f6(...
```

Figure C.1: The medium-length prompt, used for GPT-3. The first five example puzzles f1–f5 were shown to people in the user study and f6 is the one that is being solved. GPT-3’s completion was ... [1,2,3,3,4,5,6,7,8,9])



```

def f1(s: str):
    """Find a string that when concatenated onto 'Hello ' gives 'Hello world'."""
    return "Hello " + s == "Hello world"

assert True == f1("world")

---

def f2(s: str):
    """Find a string that when reversed and concatenated onto 'Hello ' gives 'Hello world'."""
    return "Hello " + s[::-1] == "Hello world"

assert True == f2("world"[::-1])

---

def f3(x: List[int]):
    """Find a list of two integers whose sum is 3."""
    return len(x) == 2 and sum(x) == 3

assert True == f3([1, 2])

---

def f4(s: List[str]):
    """Find a list of 1000 distinct strings which each have more
    'a's than 'b's and at least one 'b'."""
    return len(set(s)) == 1000 and all(
        (x.count("a") > x.count("b")) and ('b' in x) for x in s)

assert True == f4(["a" * (i + 2) + "b" for i in range(1000)])

---

def f5(n: int):
    """Find an integer whose perfect square begins with 123456789 in its decimal
    representation ."""
    return str(n * n).startswith("123456789")

assert True == f5(int(int("123456789" + "0"*9) ** 0.5) + 1)

---

def f6(li: List[int]):
    """Find a list of length 10 where the fourth element occurs exactly twice."""
    return len(li) == 10 and li.count(li[3]) == 2

assert True == f6(...)

```

Figure C.2: An example long prompt which includes English descriptions in the Python docstrings. As in the medium-length prompts, the first five example puzzles f1-f5 were shown to people in the user study and f6 is the one that is being solved.

```

def f1(s: str, a: List[str]=['cat', 'dot', 'bird'], b: List[str]=['tree', 'fly', '
dot']):
    return s in a and s in b

assert True == f1('dot')

---

def f2(li: List[int]):
    return all([sum(li[:i]) == i for i in range(20)])

assert True == f2(list(map(lambda x: 1, range(100))))

#
# omitting 22 random puzzles that GPT-3 solved...
#

---

def f25(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

assert True == f25(

```

Figure C.3: An example bootstrapping prompt which includes 24 random solved puzzles among those that GPT-3 solved, truncated at 24 due to the 2048 tokens allowed by the GPT-3 API. This puzzle f2 asks for a strategy that guarantees a tie in rock-paper-scissors and the GPT-3 completion was `list((1.0 / 3.0) for i in range(3))`.

## D NP-completeness

Before formally proving that the puzzle decision problem is NP-complete, note that the Boolean Satisfiability problem (SAT) is NP-complete and any Boolean SAT formula such as  $(x_0 \vee \neg x_7 \vee x_{17}) \wedge \dots$  can trivially be rewritten as a puzzle, e.g.,

```
def f(x: List[bool]):  
    return (x[0] or not x[7] or x[17]) and ...
```

The size of  $f$  is linear in the formula size. Thus converting a SAT formula to a puzzle is natural and does not make the problem much bigger or harder.

However, a common misconception is that NP-complete problems are all equally intractable, but the theory of NP-completeness only speaks to the worst-case complexity of solving all puzzles. While any of our puzzles could theoretically be converted to a SAT formula, the resulting formula would be mammoth without any abstraction or intuition. For example, consider the following puzzle,

```
def f(d: int): # find a non-trivial integer factor  
    """Hint, try d = 618970019642690137449562111 :-)"""  
    n = 100433627766186892221372630609062766858404681029709092356097  
    return 0 < d < n and n % d == 0
```

This puzzle is identical to the factoring puzzle  $f_3$  from Figure 1 except that the answer is given away in a comment. Any natural compiler from Python to SAT would ignore comments so the SAT form of this trivial puzzle would be quite hard. While we are not aware of such a compiler, there are programs that convert a factoring problem to a SAT instance. We ran such a converter <http://cgi.cs.indiana.edu/~sabry/cnf.html> on this  $n$  and it generated a formula with 113,878 variables and 454,633 terms! This illustrates that not all polynomials are small, and that some easy puzzles may become hard puzzles in such a conversion. The theory of NP-completeness only guarantees that if one can efficiently solve *every* SAT instance one could efficiently solve every puzzle, but specific easy puzzles may become quite hard SAT formulas.

### D.1 Proof of NP-completeness

Formally, a puzzle  $f$  represents a Turing machine as a string, a timeout  $t$  is a positive integer represented in unary, and the decision problem is, given  $(f, t)$ , does there exist an input such that when the Turing machine  $f$  is run on that input, halts in fewer than  $t$  steps and outputs 1. The time constraint is necessary to ensure that the puzzle decision problem is in NP. It is well-known that this problem is in NP and, moreover is NP-complete:

**Observation 1.** *The puzzle decision problem is NP-complete.*

*Proof.* One can test whether a given puzzle string  $f$  encoding a Turing machine halts on a witness (which we call answer) string  $x$  in time  $\leq t$  by simulating running  $f$  on  $x$  for  $t$  steps. Since simulating a Turing machine of size  $|f|$  running for  $t$  steps can be done in  $\text{poly}(|f|, t)$  time, this can be done in time  $\text{poly}(|f|, t)$  as required for NP.

To see that the problem is complete, note that given any other NP problem defined by a Turing machine  $T(z, x)$  that runs on input  $z \in \Sigma^*$  and witness  $x \in \Sigma^*$  in polynomial time  $t = p(|z|)$ , the reduction simply creates a Turing machine puzzle  $T_z$  that takes  $x$  as an input and runs  $T(z, x)$ , and we set the timeout to be  $t = p(|z|)$ . This standard reduction can also be executed in polynomial time.  $\square$

## E Unsolved problems

The following five puzzles would each represent a major breakthrough in computer science or mathematics if solved.

1. **Factoring.** In the traditional version of this ancient problem, the goal is to efficiently find the prime factorization of a given integer. In the puzzle version, we state the equivalent

problem of finding any non-trivial factor of a given integer. The puzzle is equivalent in the sense that one can recursively call the puzzle on each of the factors found until one achieves the complete prime factorization. A number of factoring algorithms have been developed over decades that factor larger and larger numbers. The RSA Factoring Challenge [see, e.g., 26] has awarded tens of thousands of dollars in prize money and RSA offered \$200,000 for factoring the largest RSA challenge number with 617 digits. The closely related [Discrete Log](#) problem is also unsolved.

2. [Graph Isomorphism](#). Given two isomorphic graphs, find the bijection that relates the two of them. In a breakthrough, Babai has claimed a quasi-polynomial time for this problem, but no polynomial time algorithm is known.
3. [Planted Clique](#). In this classic graph-theory problem, an  $n$ -node Erdős–Rényi random graph is chosen and then  $k$  nodes are selected at random and the edges are added so that they form a clique. The problem is to find the clique. It is not known whether there is a polynomial-time algorithm for this problem [see, e.g., 2].
4. [Learning Parity with Noise](#). This is a binary classification problem in computational learning theory. Roughly speaking, the problem is to efficiently learn a parity function with random classification noise. The fastest known algorithm for this problem runs in time  $\tilde{O}(2^{n/\log n})$  [8]. The problem is also closely related to efficiently decoding random linear codes [4] and various assumptions in cryptography.
5. [Collatz cycle](#). The problem is to find a cycle in the famous  $3n + 1$  process, where you start with integer  $n > 0$  and repeatedly set  $n$  to  $n/2$  if  $n$  is even, otherwise  $3n + 1$ , until you reach 1. The Collatz cycle conjecture is that there are no cycles in this process. According to the [Wikipedia article](#) on the topic, Jeffrey Lagarias stated that it “is an extraordinarily difficult problem, completely out of reach of present day mathematics” and Paul Erdős said “Mathematics may not be ready for such problems.” He also offered \$500 for its solution.

Each of these problems is described by a short (1-5 line) python function. Now, for the algorithms problems 1-3, the puzzle involves solving given instances and not exactly with the open problem: coming up with a provably polynomial-time algorithm, and it is entirely possible that no polynomial algorithm exists. However, these are all problems that have been intensely studied and an improvement, even a practical one, would be a breakthrough. For the Collatz cycle, if the Collatz conjecture holds then there is no cycle. However, we give problems involving finding integers with large Collatz delays which could be used to, at least, break records. Also noteworthy but perhaps not as well-known is [Conway’s 99 puzzle](#), an unsolved problem in graph theory due to Conway and [7] (as cited by Wikipedia). The two-line puzzle describes finding an undirected graph with 99 vertices, in which each two adjacent vertices have exactly one common neighbor, and in which each two non-adjacent vertices have exactly two common neighbors. Conway [12] offered \$1,000 for its solution.

There are also several unsolved puzzles in terms of beating records, e.g., finding oscillators or spaceships of certain periods in Conway’s game of life and finding uncrossed knights tours on chess boards of various sizes.

## F Comparing puzzles to competitive-programming problems

Figure F.1 illustrates an elementary [codeforces.com](#) problem. As is typical in programming competitions, the authors have concocted an entertaining story to motivate the problem. Dagienė and Futschek [13] include “should be funny” and “should have pictures” among desirable criteria for competitive programming problems. Also, as is typical the first step is explaining how the input is formatted and how the output should be formatted. One difficulty in authoring such competitive-programming challenges is ensuring that the English description unambiguously matches with the hidden test cases. The [ICPC rules](#) state: “A contestant may submit a claim of ambiguity or error in a problem statement by submitting a clarification request. If the judges agree that an ambiguity or error exists, a clarification will be issued to all contestants.” With puzzles, this is not necessary—a mistake in a puzzle either means that the puzzle is unsolvable or that the puzzle has an unexpected (often trivial) solution, neither of which cause major problems as it would still be a fair comparison of different solvers.

The puzzle form [InvertPermutation](#)<sup>8</sup> has no story, no description of input/output format, and no examples. The input/output formatting is taken care of simply by the type hints.

The intention is for puzzles to isolate the essence of the part of the problem that involves reasoning. Other datasets already address natural language understanding and input/output string formatting.

### Codeforces problem 474 A. Keyboard

Our good friend Mole is trying to code a big message. He is typing on an unusual keyboard with characters arranged in following way:

```
qwertyuiop  
asdfghjkl;  
zxcvbnm,./
```

Unfortunately Mole is blind, so sometimes it is problem for him to put his hands accurately. He accidentally moved both his hands with one position to the left or to the right. That means that now he presses not a button he wants, but one neighboring button (left or right, as specified in input).

We have a sequence of characters he has typed and we want to find the original message.

#### Input

First line of the input contains one letter describing direction of shifting ('L' or 'R' respectively for left or right).

Second line contains a sequence of characters written by Mole. The size of this sequence will be no more than 100. Sequence contains only symbols that appear on Mole's keyboard. It doesn't contain spaces as there is no space on Mole's keyboard.

It is guaranteed that even though Mole hands are moved, he is still pressing buttons on keyboard and not hitting outside it.

#### Output

Print a line that contains the original message.

#### Examples

##### input

```
R  
s;;upimrrfod;pbr
```

##### output

```
allyouneedislove
```

```
def f(s: str, perm="qwertyuiopasdfghjkl;zxcvbnm,./", target="s;;upimrrfod;pbr"):  
    return "".join(perm[perm.index(c) + 1] for c in s) == "target"
```

Figure F.1: Example of an introductory competition problem <https://codeforces.com/problemset/problem/474/A> (top) and the respective puzzle version (bottom) that is only using code and is short to read. In this problem, there is a given permutation of characters  $\pi$ , and a given target string  $t$ , and one wants to find a source string  $s$  such that when each character of  $s$  has been permuted with  $\pi$ , the target is achieved. The puzzle has been simplified to always shift right.

<sup>8</sup>In P3, we have slightly modified the problem slightly so that it is only inspired by the codeforces problem and not a direct translation. The P3 problem is harder in that characters not in the permutation may also appear in the string unmodified.



## G User Study Details

The user study began with a short tutorial about puzzles, which included the puzzles shown in Figure C.1. The 30 puzzles (see Figures G.6-G.7) were divided into three parts of 10 puzzles each: numbers 1-10, 11-20, and 20-30. Since each puzzle took at maximum of 6 minutes, no part took more than one hour. In the internal IRB approval (July 22, 2020), the key discussion points were that we would not collect age, gender or any other PII since it was not relevant to our study.

### G.1 Provided instructions

Figures G.1-G.3 present the initial instructions that participants were given before starting the study. Figures G.4-G.5 show the interface that they used for retrieving puzzles and submitting solutions. We run implement a Python backend to store progress logs and to serve each puzzle in its turn, so participants won't accidentally be exposed to any of the puzzles in advance. We asked participants to follow the simple interface and not to attempt any sophisticated hacking techniques that will give them any personal advantage. We did not observe any such malicious behaviour and received positive feedback for the stability and clarity of the interface.

### G.2 Qualitative feedback.

Our Jupyter notebook interface also allowed users to submit qualitative feedback. As an example of this last point, participants mentioned that they were not familiar with functions such as `zip` or `all` but learned them in the course of the study. Overall, Three themes emerged in the feedback: participants enjoyed solving the puzzles, they felt that 6 minutes was not enough time to solve the puzzles, and they felt they learned Python from doing the puzzles.

### G.3 Results summary

We had a total of 21 participants completing the user study. Participants solved between 12-30 puzzles, with 6 participants solving more than 28 puzzles, and only a single participant solving all 30. As Figure G.8 shows, the participants Python experience ranged between a few months to 8 years, with a median of 3 years. For post study analysis purposes, we denote participants with less than 3 years of experience as *beginners* and the rest as *experienced*. Figure G.9 shows the number of participants that solved each puzzle, grouped by experience. 9 of the puzzles were solved by all beginners, whereas 17 puzzles were solved by all experienced. This positive correlation between the number of programming experience and number of puzzles solved, indicates the effectiveness of our puzzles as a proxy to evaluating programming proficiency.

We also notice that experienced programmers solve puzzles faster (149 seconds per puzzle on average, compared to 194 seconds for beginners). Figure G.10 shows the distribution of time spent by participants on each puzzle. We use the per puzzle average solving time as an indicator to its perceived difficulty. As discussed in the main paper (§5.1), we see a strong correlation between the perceived difficulty of different puzzles for humans and for our examined AI solvers.

## Getting started

A Python Programming Puzzle is simply a Boolean function `puzzle` and the goal is to find an input which makes `puzzle` return `True`. Let's start with a trivial example:

```
In [1]: def puzzle(s: str):  
        return "Hello " + s == "Hello world"
```

```
In [2]: puzzle("world")  
Out[2]: True
```

That's it, so easy!

```
In [3]: def puzzle(s: str):  
        return "Hello " + s[::-1] == "Hello world"
```

```
In [4]: # What's s[::-1]? Check stackoverflow.com or just try it out:  
        "Testing"[::-1]
```

```
Out[4]: 'gnitset'
```

```
In [5]: # Aha! Now that you know what s[::-1] does, can you solve the puzzle?  
        puzzle("your solution here")
```

```
Out[5]: False
```

Figure G.1: Instructions page provided to the study participants as a Jupyter notebook (part 1).

## No cheating on types!!!

The `: str` in the arg list above indicates that the input should be a string.

In this Hackathon, we will always define the required type for the input.

For example, in the following puzzle, the input should be a list of integers:

```
In [6]: from typing import List, Tuple, Callable, Set  
  
        def puzzle(x: List[int]):  
            return len(x) == 2 and sum(x) == 3
```

Our checker will verify the correctness of the type.

## Learning to comprehend it all

For background, a lot of puzzles involve list [comprehensions](#), python functions like `set` and `all`. It will be useful to be familiar with these. For example:

```
In [7]: def puzzle(s: List[str]):  
        """  
        Find 1000 *different* strings where each string has more a's than b's.  
        """  
        return len(set(s)) == 1000 and \  
            all((x.count("a") > x.count("b")) and ('b' in x) for x in s)  
  
        # Example of a valid solution:  
        solution = []  
        for i in range(1000):  
            solution.append('baa' + 'a' * i)  
  
        print(puzzle(solution))  
  
        # You can also use comprehensions in your solution (but you're not required to):  
        solution = ["baa" + "a" * i for i in range(1000)]  
        print(puzzle(solution))  
  
        True  
        True
```

Figure G.2: Instructions page provided to the study participants as a Jupyter notebook (part 2).

## Classic puzzle example

There is no limit to how easy or hard a short puzzle can be.

Many classic puzzles problem can be written as short puzzles. The point of the example below is just to illustrate how one classic programming problem (see the 7,000 words [Wikipedia article](#)) can be written in a couple of lines of Python. (No need to read or understand it.)

Don't be scared! We are interested in puzzles of all levels as simple puzzles can be helpful to computers and people just beginning to learn Python. The study puzzles range from easy to medium since each problem is limited to 6 minutes---we do not expect anyone to solve all puzzles but many can be solved faster.

```
In [8]: def puzzle(sol: List[Tuple[int, int]]):
        """
        @param sol: list of moves (i, j) meaning a move from stack i to j (i, j in [0, 1, 2])
        """
        s = (list(range(8)), [], [])
        return all(s[j].append(s[i].pop()) or sorted(s[j]) == s[j] for i, j in sol) and s[0] == s[1]

        # The type annotation List[Tuple[int, int]] means that the solution should be a list of pairs of integers like [(0, 2),
        # A great puzzle like this is easy to state but requires a trick (in this case, recursion) to solve.

        def solve_hanoi(n_disks, i=0, j=2):
            if n_disks==0:
                return []
            k = 3 - i - j
            return solve_hanoi(n_disks - 1, i, k) + [(i, j)] + solve_hanoi(n_disks - 1, k, j)

        solution = solve_hanoi(8)
        puzzle(solution)

Out[8]: True
```

Puzzles can also require a function for a solution!

```
In [9]: def puzzle(f: Callable[[int], int]):
        """Find a function f that maps integers to integers where f(0) is nonzero but f(f(0)) is 0."""
        return f(f(0)) == 0 and f(0) != 0
```

Figure G.3: Instructions page provided to the study participants as a Jupyter notebook (part 3).

```
In [ ]: # Please run this cell first.
        from study.study import next_puzzle, cur_puzzle, puzzle, give_up, submit_feedback, submit_years

In [ ]: # Please submit the approximate number of years you have been programming in python.
        years = _ # integer.
        submit_years(years)
```

## Instructions

Thank you so much for your participation! Please first complete [Study Consent.ipynb](#).

- The first 3 problems are "practice" and the time you take will not be used in the study. This is a good chance to see how the system works.
- Puzzles are defined by `def puzzle(...)`. For each puzzle, you will try to find an input `x` which makes `puzzle(x)` return `True`.
- Type `next_puzzle()` when you are ready to start the first problem or to advance to the next problem.
- There is **no option to revisit a puzzle** and once you call `next_puzzle()` the clock starts ticking (you have up to 6 minutes per puzzle).
- If you **get lost**, call `cur_puzzle()` to see the current puzzle you are on (and time bar).

## Timing

- Please solve the problems as quickly as you can. We are measuring the difficulty of the problems both in terms of how many people solve each problem and how long on average it takes them.
- If you do not solve a problem in **6 minutes**, move to the next puzzle by typing `next_puzzle()`.
- If you are sure that you won't be able to solve the puzzle in 6 minutes and would like to skip to the next puzzle without waiting, you can call `give_up()`. However, please avoid this option when possible.

We are evaluating the puzzle's difficulty and **not your ability**, so do not feel bad about the problems you do not solve. In fact, your not solving a problem is extremely useful information for us. Also, please do not discuss the specific puzzles with people who have not yet completed the study.

## Breaks

- Since problems are timed individually, feel free to take breaks between puzzles at your convenience.
- We are storing a state for each user, so you can restart the kernel or close and reopen the notebook if needed. After doing so, please run the top cell to reimport the functions and use `cur_puzzle()` or `next_puzzle()` to get back on track.

Figure G.4: The introduction of the study notebook given to participants.

### Summary of functions

Function	Description
<code>next_puzzle()</code>	Start the next puzzle (call only when you are ready to start! no revisiting)
<code>cur_puzzle()</code>	Present the current puzzle (useful if you got lost or accidentally overridden <code>puzzle()</code> )
<code>puzzle(...)</code>	Submit a solution to the current puzzle
<code>give_up()</code>	Give up and skip to the next puzzle before 6 minutes have passed. Please avoid this option if possible.
<code>submit_feedback(...)</code>	Send us feedback

```
In [ ]: # The first 3 puzzles are warmup. Begin the warmup part by running this cell.
next_puzzle()
```

```
In [ ]: # Solve the first puzzle by running this cell.
puzzle('world')
```

```
In [ ]: # when you are ready to continue, run this cell.
next_puzzle()
```

(a) Initial view.

```
In [4]: # The first 3 puzzles are warmup. Begin the warmup part by running this cell.
next_puzzle()
```

Time:

```
PUZZLE 1/3 (WARM UP)
=====

def puzzle(s: str):
    """
    Warmup problem.
    """
    return "Hello " + s == "Hello world"
```

```
In [ ]: # Solve the first puzzle by running this cell.
puzzle('world')
```

(b) View while solving a puzzle. The progress bar advances towards the 6 minutes limit.

```
In [5]: # Solve the first puzzle by running this cell.
puzzle('world')
```

CORRECT in 00:39 sec.

```
Out[5]: True
```

(c) View after submitting a successful solution.

```
In [6]: # when you are ready to continue, run this cell.
next_puzzle()
```

Out of time

```
PUZZLE 2/3 (WARM UP)
=====

def puzzle(n: int):
    """
    Hint: puzzle(11111111) works.
    """
    return str(n * n).startswith("123456789")
```

(d) View after 6 minutes have passed since viewing the puzzle without submitting a valid solution.

```
In [6]: puzzle(1234)
```

```
Out[6]: False
```

(e) View when submitting a wrong solution to a puzzle (before timeout is reached).

Figure G.5: The interface used by participants to solve puzzles during the study. Each sub-figure shows a different state of the notebook according to the user's interaction.

```

def f1(s: str):
    return s.count("o") == 1000 and s.count("oo") == 100 and s.count("ho") == 801

def f2(s: str):
    return s.count("o") == 1000 and s.count("oo") == 0

def f3(x: List[int]):
    return sorted(x) == list(range(999)) and all(x[i] != i for i in range(len(x)))

def f4(x: List[int]):
    return len(x) == 10 and x.count(x[3]) == 2

def f5(x: List[int]):
    return all([x.count(i) == i for i in range(10)])

def f6(n: int):
    return n % 123 == 4 and n > 10**10

def f7(s: str):
    return str(8**2888).count(s) > 8 and len(s) == 3

def f8(s: List[str]):
    return s[1234] in s[1235] and s[1234] != s[1235]

def f9(x: List[int]):
    return ["The quick brown fox jumps over the lazy dog"[i] for i in x] \
        == list("The five boxing wizards jump quickly")

def f10(s: str):
    return s in str(8**1818) and s==s[::-1] and len(s)>11

def f11(x: List[str]):
    return min(x) == max(x) == str(len(x))

def f12(x: List[int]):
    return all(a + b == 9 for a, b in zip([4] + x, x)) and len(x) == 1000

def f13(x: float):
    return str(x - 3.1415).startswith("123.456")

def f14(x: List[int]):
    return all([sum(x[:i]) == i for i in range(20)])

def f15(x: List[int]):
    return all(sum(x[:i]) == 2 ** i - 1 for i in range(20))

```

Figure G.6: The first 15 puzzles in the user study.



```

def f16(x: str):
    return float(x) + len(x) == 4.5

def f17(n: int):
    return len(str(n + 1000)) > len(str(n + 1001))

def f18(x: List[str]):
    return [s + t for s in x for t in x if s!=t] == 'berlin berger linber linger
gerber gerlin'.split()

def f19(x: Set[int]):
    return {i+j for i in x for j in x} == {0, 1, 2, 3, 4, 5, 6, 17, 18, 19, 20, 34}

def f20(x: List[int]):
    return all(b in {a-1, a+1, 3*a} for a, b in zip([0] + x, x + [128]))

def f21(x: List[int]):
    return all([x[i] != x[i + 1] for i in range(10)]) and len(set(x)) == 3

def f22(x: str):
    return x[:2] in x and len(set(x)) == 5

def f23(x: List[str]):
    return tuple(x) in zip('dee', 'doo', 'dah!')

def f24(x: List[int]):
    return x.count(17) == 3 and x.count(3) >= 2

def f25(s: str):
    return sorted(s)==sorted('Permute me true') and s==s[::-1]

def f26(x: List[str]):
    return "".join(x) == str(8**88) and all(len(s)==8 for s in x)

def f27(x: List[int]):
    return x[x[0]] != x[x[1]] and x[x[x[0]]] == x[x[x[1]]]

def f28(x: Set[int]):
    return all(i in range(1000) and abs(i-j) >= 10 for i in x for j in x if i != j) \
        and len(x)==100

def f29(x: Set[int]):
    return all(i in range(1000) and abs(i*i - j*j) >= 10 for i in x for j in x if i
        != j) and len(x) > 995

def f30(x: List[int]):
    return all([123*x[i] % 1000 < 123*x[i+1] % 1000 and x[i] in range(1000)
        for i in range(20)])

```

Figure G.7: The last 15 puzzles in the user study.

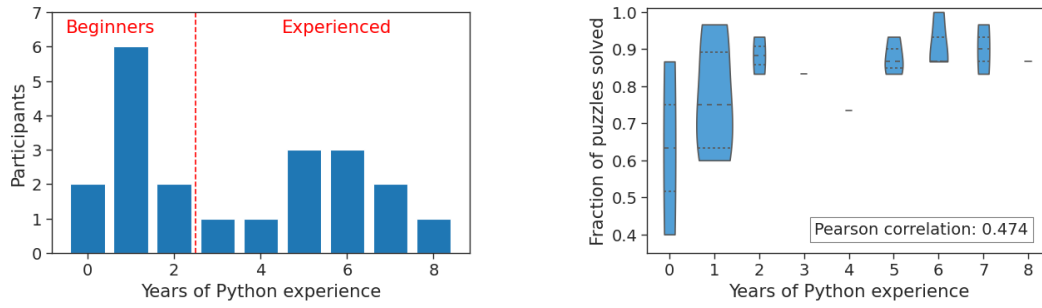


Figure G.8: Years of Python programming experience distribution of our study participants. For post study analysis purposes, we split the group by the median (3 years) to beginners and experienced programmers. The right violin plot shows the fraction of puzzles solved by participants with different years of experience. The lines in the violin show the four quartiles.

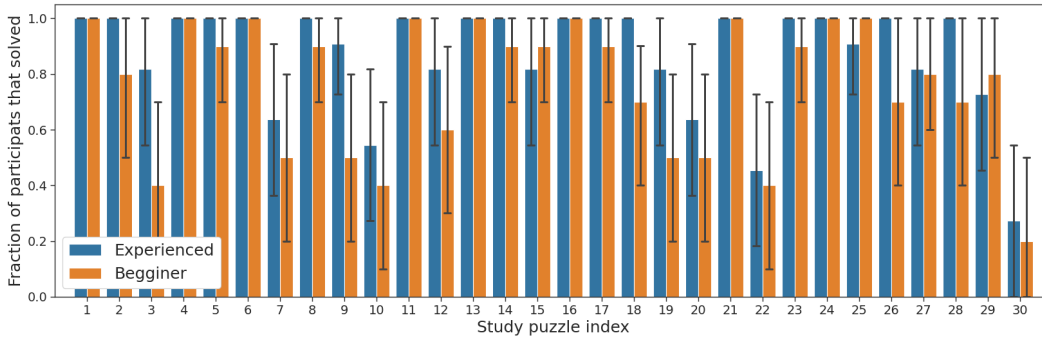


Figure G.9: Fraction of participants, divided to experienced and beginners, that solved each of the 30 puzzles in less than 6 minutes.

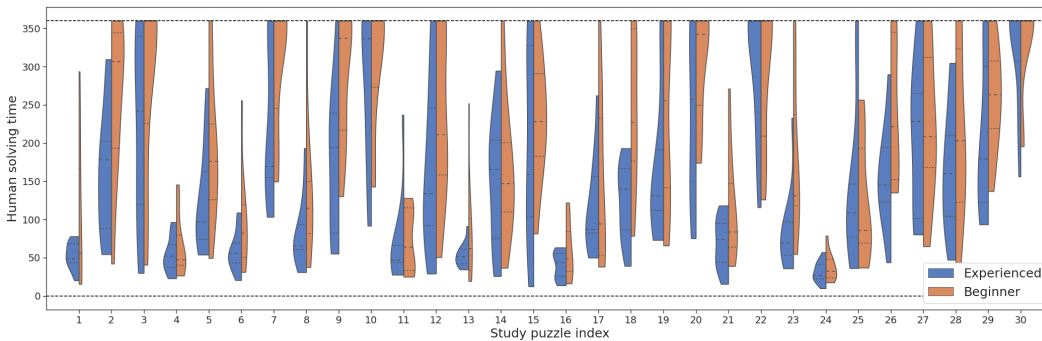


Figure G.10: The time that participants spent on each study puzzle, up to 360 seconds per puzzle. For unsolved puzzles, we count the time as using the full 6 minutes. The orange colored areas on the right show the time distribution for beginner Python coders with less than three years experience (10 out of 21). The blue colored areas on the left present the solving time distribution per puzzle for users with at least three years of experience in Python.

## H GPT-3 Bootstrap results

Below are the 87 puzzles solved by the GPT-3 bootstrap run for 10K tries per puzzle on the set of 138 puzzles.

*# Bootstrap solved after 1 tries:*

```
def f1(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  
SOL: "5"
```

*# Bootstrap solved after 1 tries:*

```
def f2(path: List[int], edges=[[0, 1], [0, 2], [1, 2], [1, 3], [2, 3]]):  
    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  
SOL: "list(range(4))"
```

*# Bootstrap solved after 2 tries:*

```
def f3(n: int, s='0000111111100000', k=5):  
    return s[n:n + k] == s[n] * k  
SOL: "4"
```

*# Bootstrap solved after 3 tries:*

```
def f4(z: float, v=9, d=0.0001):  
    return int(z * 1/d % 10) == v  
SOL: "2.3"
```

*# Bootstrap solved after 3 tries:*

```
def f5(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:])  
SOL: "8"
```

*# Bootstrap solved after 4 tries:*

```
def f6(n: int, lace='bbbrrrbrrbrbrrrrr'):  
    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")  
SOL: "4"
```

*# Bootstrap solved after 7 tries:*

```
def f7(i: int, s='cat', target='a'):  
    return s[i] == target  
SOL: "1"
```

*# Bootstrap solved after 10 tries:*

```
def f8(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  
SOL: "19143213"
```

*# Bootstrap solved after 11 tries:*

```
def f9(n: int, target='foolfoolfofoo', s='foolfo'):  
    return s * n == target  
SOL: "2"
```

```

# Bootstrap solved after 12 tries:
def f10(i: int, li=[17, 31, 91, 18, 42, 1, 9], target=18):
    return li[i] == target
SOL: "3"

# Bootstrap solved after 19 tries:
def f11(li: List[int]):
    return all(j in {i - 1, i + 1, 3 * i} for i, j in zip([0] + li, li + [128]))
SOL: "list(range(128))"

# Bootstrap solved after 20 tries:
def f12(n: int, year_len=365):
    prob = 1.0
    for i in range(n):
        prob *= (year_len - i) / year_len
    return (prob - 0.5) ** 2 <= 1/year_len
SOL: "24"

# Bootstrap solved after 23 tries:
def f13(si: Set[int]):
    return all(i in range(1000) and abs(i - j) >= 10 for i in si for j in si if i !=
j) and len(si) == 100
SOL: "set(range(1, 1000, 10))"

# Bootstrap solved after 25 tries:
def f14(ls: List[str]):
    return tuple(ls) in zip('dee', 'doo', 'dah!')
SOL: "['d', 'd', 'd']"

# Bootstrap solved after 30 tries:
def f15(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
SOL: "3"

# Bootstrap solved after 30 tries:
def f16(li: List[int], n=85012):
    return len(li) == n
SOL: "list(range(85012))"

# Bootstrap solved after 48 tries:
def f17(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)])
SOL: "list(range(7, 200, 5))"

# Bootstrap solved after 57 tries:
def f18(item: int, li=[17, 2, 3, 9, 11, 11], index=4):
    return li.index(item) == index
SOL: "11"

# Bootstrap solved after 64 tries:
def f19(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
SOL: "8"

# Bootstrap solved after 79 tries:
def f20(x: List[int]):
    return len(x) == 2 and sum(x) == 3
SOL: "[] + [1, 2]"

# Bootstrap solved after 119 tries:

```

```

def f21(s: str, n=7):
    return int(str(5 ** n)[:2] + s) == 5 ** n
SOL: "str(int(-5) ** 2)"

# Bootstrap solved after 129 tries:
def f22(s: str):
    return float(s) + len(s) == 4.5
SOL: "str('1.5')"
```

*# Bootstrap solved after 131 tries:*

```

def f23(s: str):
    return "Hello " + s == "Hello world"
SOL: "str('world')"
```

*# Bootstrap solved after 141 tries:*

```

def f24(s: str, target='foofoofoofoo', n=2):
    return s * n == target
SOL: "str('foofoo')"
```

*# Bootstrap solved after 145 tries:*

```

def f25(s: str, a=['cat', 'dot', 'bird'], b=['tree', 'fly', 'dot']):
    return s in a and s in b
SOL: "str('dot')"
```

*# Bootstrap solved after 151 tries:*

```

def f26(s: str, target='foobazwow', length=6):
    return target[(len(target) - length) // 2:(len(target) + length) // 2] == s
SOL: "str('barbaz')"
```

*# Bootstrap solved after 160 tries:*

```

def f27(s: str):
    return s + 'world' == 'Hello world'
SOL: "str('Hello ')"
```

*# Bootstrap solved after 162 tries:*

```

def f28(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)])
SOL: "str('abcd')"
```

*# Bootstrap solved after 162 tries:*

```

def f29(st: str, a='world', b='Hello world'):
    return st + a == b
SOL: "str('Hello ')"
```

*# Bootstrap solved after 166 tries:*

```

def f30(li: List[int]):
    return li.count(17) == 3 and li.count(3) >= 2
SOL: "list([17, 3, 17, 3, 17])"
```

*# Bootstrap solved after 177 tries:*

```

def f31(s: str, strings=['cat', 'dog', 'bird', 'fly', 'moose']):
    return s in strings and sum(t > s for t in strings) == 1
SOL: "str('fly')"
```

*# Bootstrap solved after 179 tries:*

```

def f32(s: str, a='hello', b='yellow', length=4):
    return len(s) == length and s in a and s in b
SOL: "str('ello')"
```

*# Bootstrap solved after 199 tries:*

```

def f33(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2
SOL: "list([0, 10, 0, 10, 0, 3, 1, 2, 6, 2])"
```

*# Bootstrap solved after 221 tries:*

```

def f34(n: int, nums=[15, 27, 102], upper_bound=5000):
    return all(n % i == 0 for i in nums) and n <= upper_bound
SOL: "0"

# Bootstrap solved after 231 tries:
def f35(s: str, n=1000):
    return len(s) == n
SOL: "str('f' * 1000)"

# Bootstrap solved after 237 tries:
def f36(x: int, a=93252338):
    return -x == a
SOL: "int(-93252338)"

# Bootstrap solved after 269 tries:
def f37(x: int, a=-382, b=14546310):
    return x - a == b
SOL: "int(14546310 - 382)"

# Bootstrap solved after 301 tries:
def f38(x: int, a=8665464, b=-93206):
    return a - x == b
SOL: "int(8665464 - -93206)"

# Bootstrap solved after 302 tries:
def f39(i: int, s='cat', target='a'):
    return s[i] == target and i < 0
SOL: "int(-2)"

# Bootstrap solved after 321 tries:
def f40(x: int, a=9384594, b=1343663):
    if x > 0 and a > 50:
        return x - a == b
    else:
        return x + a == b
SOL: "int(9384594 - -1343663)"

# Bootstrap solved after 325 tries:
def f41(i: int, li=[17, 31, 91, 18, 42, 1, 9], target=91):
    return li[i] == target and i < 0
SOL: "int(-5)"

# Bootstrap solved after 336 tries:
def f42(s: str):
    return s[::-1] + 'world' == 'Hello world'
SOL: "str('Hello ')[::-1]"

# Bootstrap solved after 341 tries:
def f43(s: str):
    return "Hello " + s[::-1] == "Hello world"
SOL: "str(\"dlrow\")"

# Bootstrap solved after 342 tries:
def f44(n: int, a=14302, b=5):
    return b * n + (a % b) == a
SOL: "int(14302 / 5)"

# Bootstrap solved after 350 tries:
def f45(x: int, a=253532, b=1230200):
    if x > 0 or a > 50:
        return x - a == b
    else:
        return x + a == b
SOL: "int(253532 + 1230200)"

```



```

# Bootstrap solved after 385 tries:
def f46(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
SOL: "int(54368639 - 4)"

# Bootstrap solved after 398 tries:
def f47(li: List[int]):
    return li[li[0]] != li[li[1]] and li[li[li[0]]] == li[li[li[1]]]
SOL: "list([3, 9, 9, 4, 1, 9, 8, 6, 1, 10, 1, 9])"

# Bootstrap solved after 402 tries:
def f48(n: int, a=3, b=23463462):
    return b // n == a
SOL: "int(23463463 / 3)"

# Bootstrap solved after 451 tries:
def f49(x: int, a=1073258, b=72352549):
    return a + x == b
SOL: "int(72352549 - 1073258)"

# Bootstrap solved after 462 tries:
def f50(li: List[int]):
    return all([li[i] != li[i + 1] for i in range(10)]) and len(set(li)) == 3
SOL: "list([9, 1, 0, 9, 1, 0, 9, 1, 0, 9, 1, 0])"

# Bootstrap solved after 488 tries:
def f51(n: int, a=15482, b=23223, lower_bound=5):
    return a % n == 0 and b % n == 0 and n >= lower_bound
SOL: "int(23223 / 3)"

# Bootstrap solved after 488 tries:
def f52(n: int, nums=[77410, 23223, 54187], lower_bound=2):
    return all(i % n == 0 for i in nums) and n >= lower_bound
SOL: "int(23223 / 3)"

# Bootstrap solved after 493 tries:
def f53(li: List[int], target=[17, 9, -1, 17, 9, -1], n=2):
    return li * n == target
SOL: "list([17, 9, -1])"

# Bootstrap solved after 504 tries:
def f54(path: List[int], edges=[[0, 2], [0, 1], [2, 1], [2, 3], [1, 3]]):
    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
SOL: "list([0, 2, 1, 3])"

# Bootstrap solved after 539 tries:
def f55(e: List[int], edges=[[0, 217], [40, 11], [17, 29], [11, 12], [31, 51]]):
    return e in edges
SOL: "list([40, 11])"

# Bootstrap solved after 652 tries:
def f56(n: int, a=345346363, b=10):
    return n // b == a
SOL: "int(3453463634)"

# Bootstrap solved after 758 tries:
def f57(path: List[int], edges=[[0, 11], [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
SOL: "list([0, 22, 33])"

# Bootstrap solved after 782 tries:
def f58(p: List[int], edges=[[0, 1], [0, 2], [1, 2], [3, 1], [2, 3]]):
    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:]))
SOL: "list([0, 1, 2, 3, 1])"

# Bootstrap solved after 787 tries:
def f59(s: str, dups=2021):
    return len(set(s)) == len(s) - dups
SOL: "str(2022 * 'S')"

# Bootstrap solved after 846 tries:
def f60(x: str, s=679):
    return s == sum([int(d) for d in x])
SOL: "str('1' * 679)"

# Bootstrap solved after 1039 tries:
def f61(li: List[int]):
    return all(sum(li[:i]) == 2 * i - 1 for i in range(20))
SOL: "list(map(lambda i: 2*i, range(20)))"

# Bootstrap solved after 1043 tries:
def f62(x: int, a=10201202001):
    return x ** 2 == a
SOL: "int(10201202001 ** 0.5)"

# Bootstrap solved after 1053 tries:
def f63(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
SOL: "list(map(lambda i: float((i + 2) / 3 - (i + 1) / 3), range(3)))"

# Bootstrap solved after 1059 tries:
def f64(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
SOL: "list([12, 4, 3, 1, 2, 1, 5, 94, 2, 2, 13])"

# Bootstrap solved after 1121 tries:
def f65(x: float, a=1020):
    return abs(x ** 2 - a) < 10 ** -3 and x < 0
SOL: "float(-1020 ** 0.5)"

# Bootstrap solved after 1147 tries:
def f66(x: float, a=1020):
    return abs(x ** 2 - a) < 10 ** -3
SOL: "float(1020 ** 0.5)"

# Bootstrap solved after 1282 tries:
def f67(n: int, a=10000200001):
    return a == n * n and n < 0
SOL: "int(-10000200001 ** 0.5)"

```

```

# Bootstrap solved after 1299 tries:
def f68(i: int, n=62710561):
    return 1 < i < n and n % i == 0
SOL: "int(62710561 ** 0.5)"

# Bootstrap solved after 1331 tries:
def f69(i: int):
    return len(str(i + 1000)) > len(str(i + 1001))
SOL: "int(-1001)"

# Bootstrap solved after 1375 tries:
def f70(li: List[int]):
    return all([sum(li[:i]) == i for i in range(20)])
SOL: "list([1] * 20)"

# Bootstrap solved after 1414 tries:
def f71(big_str: str, sub_str='foobar', index=2):
    return big_str.index(sub_str) == index
SOL: "str('f foobar foo f')"

# Bootstrap solved after 1669 tries:
def f72(x: int, a=324554, b=1345345):
    if a < 50:
        return x + a == b
    else:
        return x - 2 * a == b
SOL: "int(1345345 + 324554 * 2)"

# Bootstrap solved after 1752 tries:
def f73(s: str, strings=['cat', 'dog', 'bird', 'fly', 'moose']):
    return s[:-1] in strings and sum(t < s[:-1] for t in strings) == 1
SOL: "str('cat')[:-1]"

# Bootstrap solved after 1760 tries:
def f74(s: str):
    return s.count('o') == 1000 and s.count('oo') == 0
SOL: "str('ow' * 1000)"

# Bootstrap solved after 2456 tries:
def f75(nums: List[int], target=10):
    assert target % 9 not in [4, 5], "Hint"
    return len(nums) == 3 and sum([i ** 3 for i in nums]) == target
SOL: "list([1, 1, 2])"

# Bootstrap solved after 2517 tries:
def f76(x: List[int], n=5, s=19):
    return len(x) == n and sum(x) == s and all([a > 0 for a in x])
SOL: "list([2, 7, 3, 1, 6])"

# Bootstrap solved after 3027 tries:
def f77(i: int):
    return i % 123 == 4 and i > 10 ** 10
SOL: "int(10 ** 10 * 123 + 4)"

# Bootstrap solved after 3388 tries:
def f78(s: str, big_str='foobar', index=2):
    return big_str.index(s) == index
SOL: "str('ob')"

# Bootstrap solved after 3447 tries:
def f79(ls: List[str]):
    return min(ls) == max(ls) == str(len(ls))
SOL: "['2', '2']"

# Bootstrap solved after 3890 tries:

```

```

def f80(li: List[int], dups=42155):
    return len(set(li)) == len(li) - dups
SOL: "list(range(42155))*2"

# Bootstrap solved after 5560 tries:
def f81(n: int):
    return str(n * n).startswith("123456789")
SOL: "int(int(\"123456789\" * 2) ** 0.5)"

# Bootstrap solved after 5986 tries:
def f82(ls: List[str]):
    return [s + t for s in ls for t in ls if s != t] == 'berlin berger linber linger
gerber gerlin'.split()
SOL: "['ber', 'lin', 'ger']"

# Bootstrap solved after 6086 tries:
def f83(bi: List[int], g1=[[0, 1], [1, 2], [2, 3], [3, 4]], g2=[[0, 4], [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}
SOL: "list([0, 2, 3, 4, 1])"

# Bootstrap solved after 6408 tries:
def f84(x: float):
    return str(x - 3.1415).startswith("123.456")
SOL: "float(123.456 + 3.1415)"

# Bootstrap solved after 6686 tries:
def f85(si: Set[int]):
    return {i + j for i in si for j in si} == {0, 1, 2, 3, 4, 5, 6, 17, 18, 19, 20,
34}
SOL: "set([0, 1, 2, 3, 17])"

# Bootstrap solved after 6849 tries: ("The Monkey and the coconuts" ancient puzzle)
def f86(n: int):
    for i in range(5):
        assert n % 5 == 1
        n -= 1 + (n - 1) // 5
    return n > 0 and n % 5 == 1
SOL: "int(10 ** 10 * 105 - 4)"

# Bootstrap solved after 8845 tries:
def f87(ls: List[str]):
    return ls[1234] in ls[1235] and ls[1234] != ls[1235]
SOL: "['foo', 'bar', 'foobar']*1234"

```