We provide complementary details, analysis and results in the following sections:

A Example solutions by enumerative models

We provide examples of our examined enumerative solvers to three P3 puzzle in Figure A.1 on page 17 (examples of LM solutions are found in the P3 repository). 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 Skickit-learn library [41]. The Transformer model is implemented on top of the Hugging Face repository [59]. 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.

```
# 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(_)		

B.2 Transformer implementation

We use the RoBERTa-base 12-layers Transformer [36] pretrained on English text and fine-tune it on Python code using the Hugging Face library [59]. For fine-tuning data, we use Python functions with their documentation text from GitHub repositories [27]. 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.

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 [25] and batch normalization [29] to

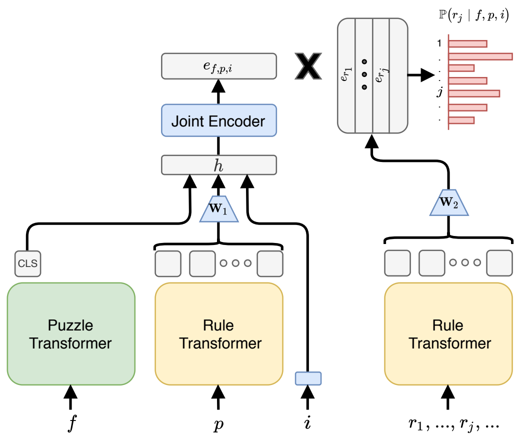


Figure B.1: An illustration of our Transformer-based enumerative solver. The rule strings are encoded with a Transformer pretrained on Python code. The puzzle Transformer is initialized the same, but is further fine-tuned on puzzles, together with the rest of the solver's parameters shown in blue color. 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.

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

Figure C.1: A puzzle where type-checking is important. A type-safe solution is computed by the program returning `"ox" * 1000`. However, `["o"] * 1000` would be considered invalid as it is a list of strings, though it does satisfy the puzzle as stated.

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 Language Model solvers details

The GPT-3 and Codex APIs were used to generate completions based on prompts. For all models, the completions were generated in batches of `n=32` with `temp=0.9`, for a maximum of 150 tokens, 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 where also checked to be of the type requested in the problem, as was the case for the top-down solver. Figure C.1 illustrates a puzzle where type checking matters.

Prompt programming. The space of possible prompts is practically boundless. Our current prompt designs leverage the API without fine-tuning. For GPT-3, among the prompts we experimented with, we found that the `assert` structure worked best but it was limited to one-line Python solutions. One-line Python programs are considered, by some, to be a useful form of programming with books dedicated to the topic [see, e.g., 39]. For Codex, we found a prompt that resembled a legal python file with a multi-line solution structure worked better.

```python
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.2: 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])`

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)=True."

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 [50]. There are several libraries available for scoring programming competitions to serve this 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 3, C.2, C.5-C.4, and C.7 show our prompts for the Short, Medium, Long, and Bootstrap prompts, respectively.

```python
from typing import List

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

def g1():
    return "world"

assert f1(g1())

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

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

assert f2(g2())

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

def g3():
    return [1, 2]

assert f3(g3())

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

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

assert f4(g4())

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

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

assert f5(g5())

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

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

Codex completed it successfully as:

```python
    inds = []
    ind = 0
    for c in "substring":
        while string[ind] != c:
            ind += 1
        inds.append(ind)
        ind += 1
    return inds
```

Figure C.3: The medium-length prompt, used for Codex. The first five example puzzles `f1-f5` were given in the tutorial to participants in the user study and `f6` is the puzzle that is being solved.

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 4b. We use the unbiased estimator of $\mathbf{pass}@k$ defined by Chen et al. [13].

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 $(x_0 \vee \neg x_7 \vee x_{17}) \wedge \ldots$ 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 `f3` 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, x, t), does there exist y such that when the Turing machine f is run on (y, x), it 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 y in time $\leq t$ by simulating running f on (y, x) for t steps. Since simulating a Turing machine of size $|f|$ running for t steps can be done in $\mathrm{poly}(|f|, t)$ time, this can be done in time $\mathrm{poly}(|f|, t)$ as required for NP.

To see that the problem is complete, note that given any other NP problem defined by a Turing machine $T(x, y)$ that runs on input $x \in \Sigma^*$ and witness $y \in \Sigma^*$ in polynomial time $t = p(|x|)$ is a type of puzzle itself for $f = T$ (with inputs swapped). $\square$

E Open 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., 30] 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 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., 4].
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 $\tilde{O}(2^{n/\log n})$ [11]. The problem is also closely related to efficiently decoding random linear codes [7] and various assumptions in cryptography. Note that some of the instances of this problem are small (and thus easy) while others are quite large.
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 poly-time 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 [10] (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 [16] 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 [17] 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[10] 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.

G User Study Details

The user study began with a short tutorial about puzzles, which included the puzzles shown in Figure C.2. 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

A total of 21 participants completed 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

[10] In P3, we have slightly modified the problem 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.

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.

H Solution to Tower of Hanoi

Codex's solution to the Tower of Hanoi puzzle is shown in Figure H.1. Even though the puzzle did not mention the word Hanoi, Codex's solution clearly knew the reference, in fact offering a link to the Wikipedia page. The first part of the URL is correct, but there is no "Advanced computer algorithm" section on the page, so the link simply resolves to the Wikipedia page. The Python code on the Wikipedia page is only similar in spirit, in part because the way the puzzle asks for the moves is somewhat different from the Wikipedia page. This is a difficult puzzle for which solutions are found at a rate of approximately 0.03%. Surprisingly, Codex was not able to solve the puzzle when we renamed the variable `num_disks` to `n` and removed the string `"bigger disk on top"`, possibly because the association with Tower of Hanoi was weaker.

```python
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.4: An example GPT-3 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.

```python
from typing import List

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

def g1():
    """Find a string that when concatenated onto 'Hello ' gives 'Hello world'."""
    return "world"

assert f1(g1())

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

def g2():
    """Find a string that when reversed and concatenated onto 'Hello ' gives 'Hello world'.
    """
    return "world"[::-1]

assert f2(g2())

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

def g3():
    """Find a list of two integers whose sum is 3."""
    return [1, 2]

assert f3(g3())

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

def g4():
    """Find a list of 1000 distinct strings which each have more 'a's than 'b's and at least
    one 'b'."""
    return ["a"*(i+2)+"b" for i in range(1000)]

assert f4(g4())

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

def g5():
    """Find an integer whose perfect square begins with 123456789 in its decimal
     representation."""
    return int(int("123456789" + "0"*9) ** 0.5) + 1

assert f5(g5())

def sat(d: int, n=6002685529):
        return n % d == 0 and all(i in "47" for i in str(d))

def sol(n=6002685529):
    """Find a integer factor of n whose decimal representation consists only of 7's and 4's.
    """
```

Figure C.5: An example Codex 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. The representation here supports multi-line solutions. Codex's successful completion is shown in Figure C.6.

```
# return the closest factor to n which is inferior either to n or to the sqrt(n)
    if n == 1:      # def.: n>1!
        return 1
    i = int(n ** 0.5)
    while True:
        if f6(i):
            return i
        elif i == 2:
            raise Exception("Could not find any 7 or 4 number factors
for n.")
        i -= 1
```

Figure C.6: An example Codex completion to the long prompt illustrated in C.5. The solution cleverly calls `f6`, a trick that some participants used to solve a couple of the user study puzzles. Codex also generated English comments.

```python
from typing import List

def f1(item: int, li=[17, 2, 3, 9, 11, 11], index=4):
    return li.index(item) == index

def g1(li=[17, 2, 3, 9, 11, 11], index=4):
    return li[index]

assert f1(g1())

def f2(s: str, word="antidisestablishmentarianism", max_len=10):
    if len(word) <= max_len:
        return word == s
    return int(s[1:-1]) == len(word[1:-1]) and word[0] == s[0] and word[-1] == s[-1]

def g2(word="antidisestablishmentarianism", max_len=10):
    if len(word) == max_len:
        return word
    return word[0] + str(len(word[1:-1])) + word[-1]

assert f2(g2())

#
# omitting 28 random puzzles that Codex solved...
#

assert f31(g31())

def f32(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)])

def g32(a=7, s=5, e=200):
```

```python
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.7: Example bootstrapping prompts for the Codex and GPT-3 LMs. The prompts includes random solved puzzles among those that the LM solved, truncated to the token limit of the API (2048 for GPT3 and 4096 for Codex).

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 π, 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 π, the target is achieved. The puzzle has been simplified to always shift right.

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 [::-1]? Check stackoverflow.com or just try it out:

        "Testing"[::-1]
Out[4]: 'gnitseT'
```

```
In [5]: # Aha! Now that you know what [::-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]:
```python
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]:
```python
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 [ ]:
```python
# Please run this cell first.
from study.study import next_puzzle, cur_puzzle, puzzle, give_up, submit_feedback, submit_years
```

In [ ]:
```python
# 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
`next_puzzle()`	Start the next puzzle (call only when you are ready to start! no revisiting)
`cur_puzzle()`	Present the current puzzle (useful if you got lost or accidentally overridden `puzzle()`)
`puzzle(...)`	Submit a solution to the current puzzle
`give_up()`	Give up and skip to the next puzzle *before* 6 minutes have passed. Please avoid this option if possible.
`submit_feedback(...)`	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:

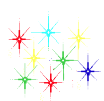
```
PUZZLE 1/3 (WARM UP)
============

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(111111111) 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.

```python
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.

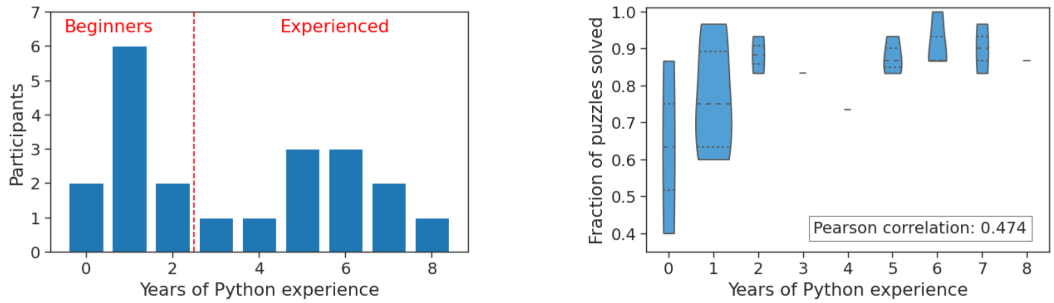


Figure G.8: Years of Python programming experience distribution of our study participants. For post study analysis purposes, we split the group by the median (3 years) to beginners and experienced programmers. The right violin plot shows the fraction of puzzles solved by participants with different years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

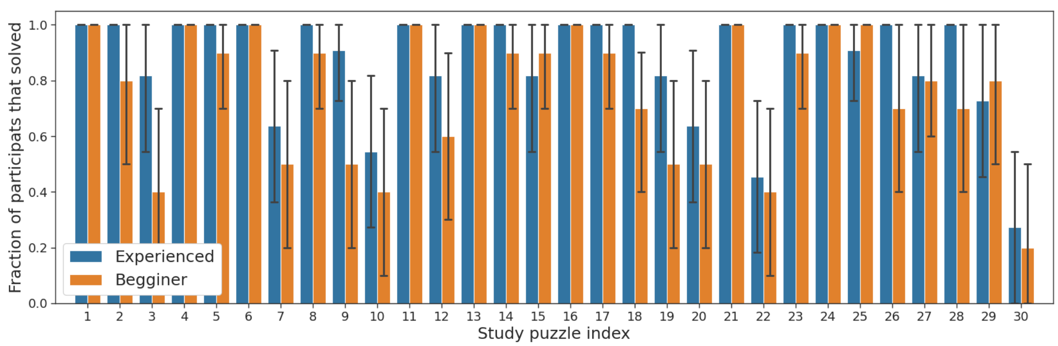


Figure G.9: Fraction of participants, divided to experienced and beginners, that solved each of the 30 puzzles in less than 6 minutes.

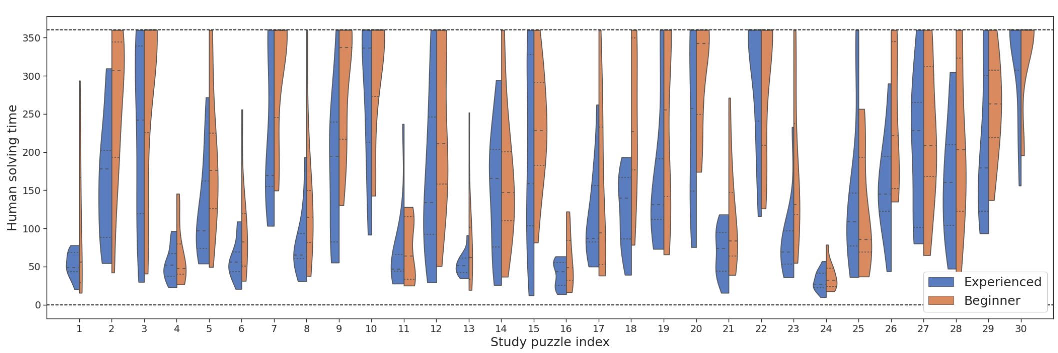


Figure G.10: The time that participants spent on each study puzzle, up to 360 seconds per puzzle. For unsolved puzzles, we count the time as using the full 6 minutes. The orange colored areas on the right show the time distribution for beginner Python coders with less than three years experience (10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

The prompt was the medium Codex prompt (without English descriptions):

```
from typing import List

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

def g1():
    return "world"

assert f1(g1())

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

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

assert f2(g2())


# skipping two puzzles, see Figure C.3

assert f5(g5())

def f6(moves: List[List[int]], num_disks=8):
    state = [1] * num_disks
    for [i, j] in moves:
        assert state.index(i) <= (state + [1, 2, 3]).index(j), "bigger disk
     on top"
        state[state.index(i)] = j
    return state == [3] * num_disks

def g6(num_disks=8):
```

And a valid completion from Codex, the smaller cushman-codex engine, was:

```
    # Algorithm is equivalent to moving all disks.
    # From https://en.wikipedia.org/wiki/Tower_of_Hanoi#Advanced_computer_algorithm
    def hanoi(n, p, q, r):
        if n > 0:
            hanoi(n - 1, p, r, q)
            moves.append([p, r])
            hanoi(n - 1, q, p, r)
    moves = []
    hanoi(num_disks, 1, 2, 3)
    assert f6(moves, num_disks)
    return moves
```

Figure H.1: A solution to the Tower of Hanoi puzzle found by the small Codex. The Wikipedia page link generated by the model indeed contains (though not in the *Advanced computer algorithm* section) a solution to a slightly different formulation of this puzzle, see Figure H.2. Note that the medium prompt doesn't mention the name of the puzzle. Codex made the correct association, and adjusted the solution code to the state-based representation of this puzzle as given in `f6`. Interestingly, replacing the use of *disks* in the puzzle's variable names with other non-descriptive options seems to prevent Codex from solving this puzzle.

```python
A = [3, 2, 1]
B = []
C = []

def move(n, source, target, auxiliary):
    if n > 0:
        # Move n - 1 disks from source to auxiliary, so they are out of the way
        move(n - 1, source, auxiliary, target)

        # Move the nth disk from source to target
        target.append(source.pop())

        # Display our progress
        print(A, B, C, '##############', sep='\n')

        # Move the n - 1 disks that we left on auxiliary onto target
        move(n - 1, auxiliary, target, source)

# Initiate call from source A to target C with auxiliary B
move(3, A, C, B)
```

Figure H.2: The algorithm from https://en.wikipedia.org/wiki/Tower_of_Hanoi#Recursive_implementation (November, 2021) that solves Tower of Hanoi for a representation in which the three towers are lists with disk indices.