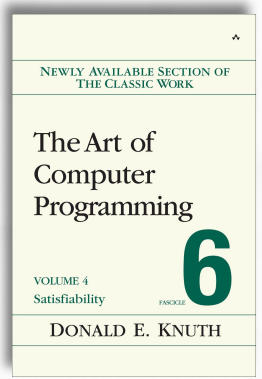
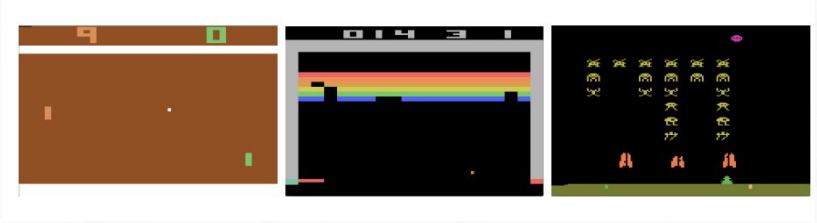
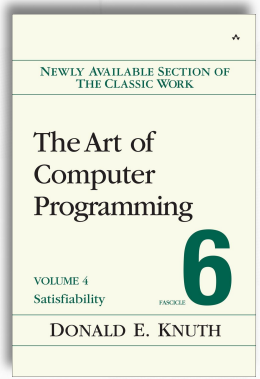
Programming Puzzles

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, Adam Tauman Kalai

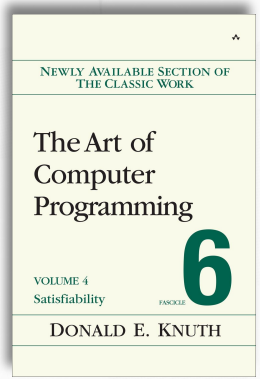
Puzzles for teaching and evaluation





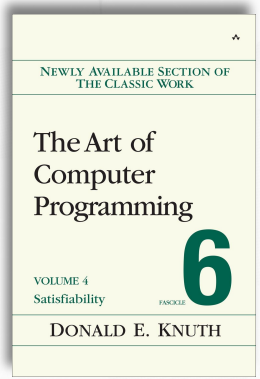


Single representation



Single representation

across many different challenge types

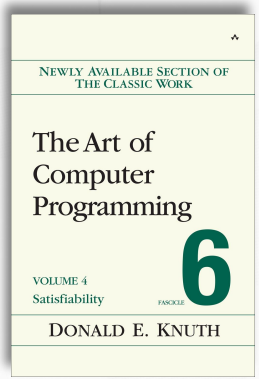


Single representation

across many different challenge types

for both humans and machines

The Guardian/ Stan Honda/AFP/Getty Images



Programming Puzzles

What is a programming puzzle?

- A **function** in any programming language (e.g., Python) that **returns a Boolean** value

- The challenge is to **find the input that satisfies the function** (i.e., makes it return **True**)

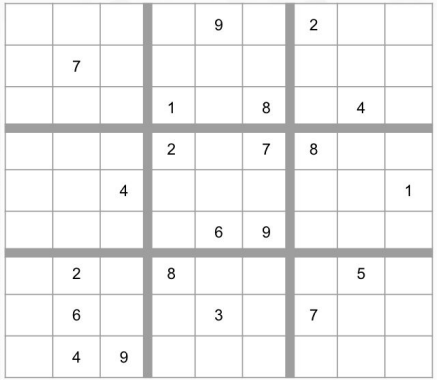
```
# Find y that solves the puzzle.
Def puzzle(y, x=[...]):
    # code defining the puzzle…
    Return solved  # Boolean.
```

Example - Sudoku

- Find the input that:
  - Is consistent with the given scheme
  - Contains only 1-9 values
  - No duplicates in rows
  - No duplicates in columns
  - No duplicates in 3x3 squares

				9		2		
	7							
			1		8		4	
			2		7	8		
		4						1
				6	9			
	2		8				5	
	6			3		7		
	4	9						

Example - Sudoku



Python Programming Puzzle

```
def sat(y: str, x="____9_2___7_________1_8_4____2_78____4_____1____69____2_8___5__6__3_7___49_____"):
    assert all(c == "_" or c == s for (c, s) in zip(x, y)), "incosistent with x"

    full = set('123456789')
    for i in range(9):
        assert {y[i] for i in range(9 * i, 9 * i + 9)} == full, "invalid row"
        assert {y[i] for i in range(i, i + 81, 9)} == full, "invalid column"
        assert {y[9 * a + b + i + 26 * (i % 3)] for a in range(3) for b in range(3)} == full, "invalid square"

    return True
```

Formal setting

- Both puzzles and answers are strings
  - Puzzle $f(x,y)$: function (source code) with a defined input type and arguments x
  - Answer y: object
  - Correct answer: $f(x,y)$ is True within time t
- Solution $g(x)$: source code to generate y from x

```
def f(y: str, x="Hello world"):   # find a string that will make the function return "True".
    return "Hello " + y == x

def g(x):   # solution.
    y = x.split(" ")[-1]
    return y   # answer.

assert f(g())
```

- **Solver**: takes **n** puzzles and timeouts and predicts solutions

Why is it important?

- AI models are getting better at code completion and generation

GitHub Copilot

- Need objective evaluation of coding proficiency to measure and spur progress

Why is it important?

- AI models are getting better at code completion and generation

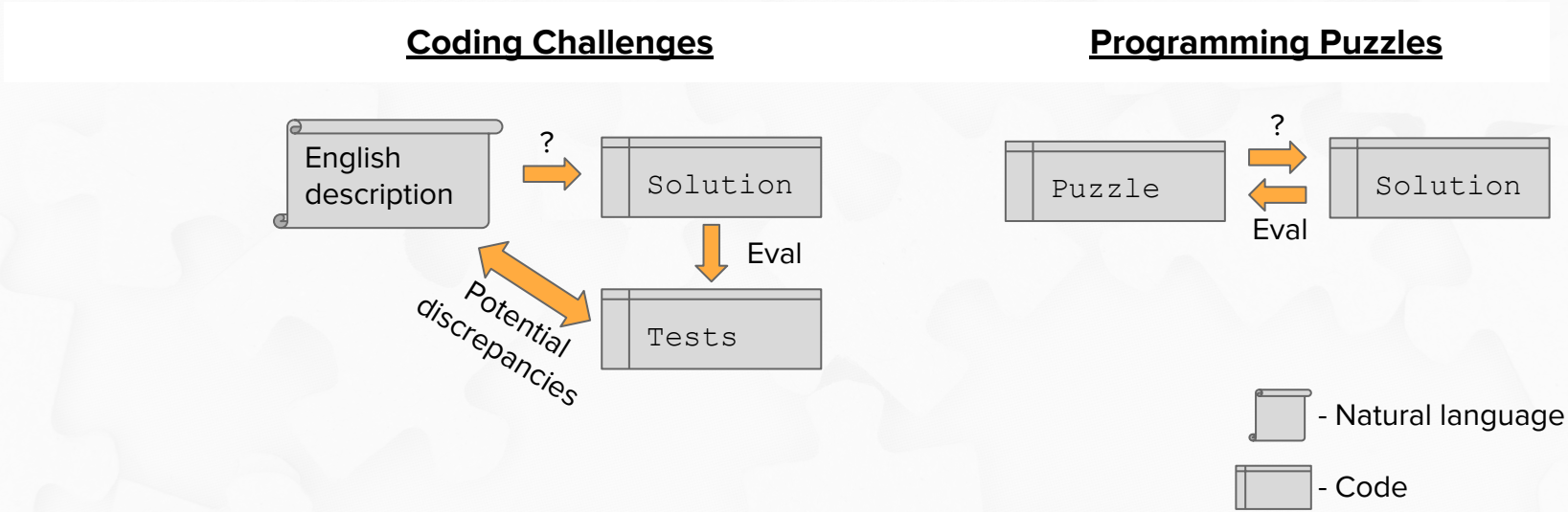
Ambiguous description

Errors indicating lack of basic understanding

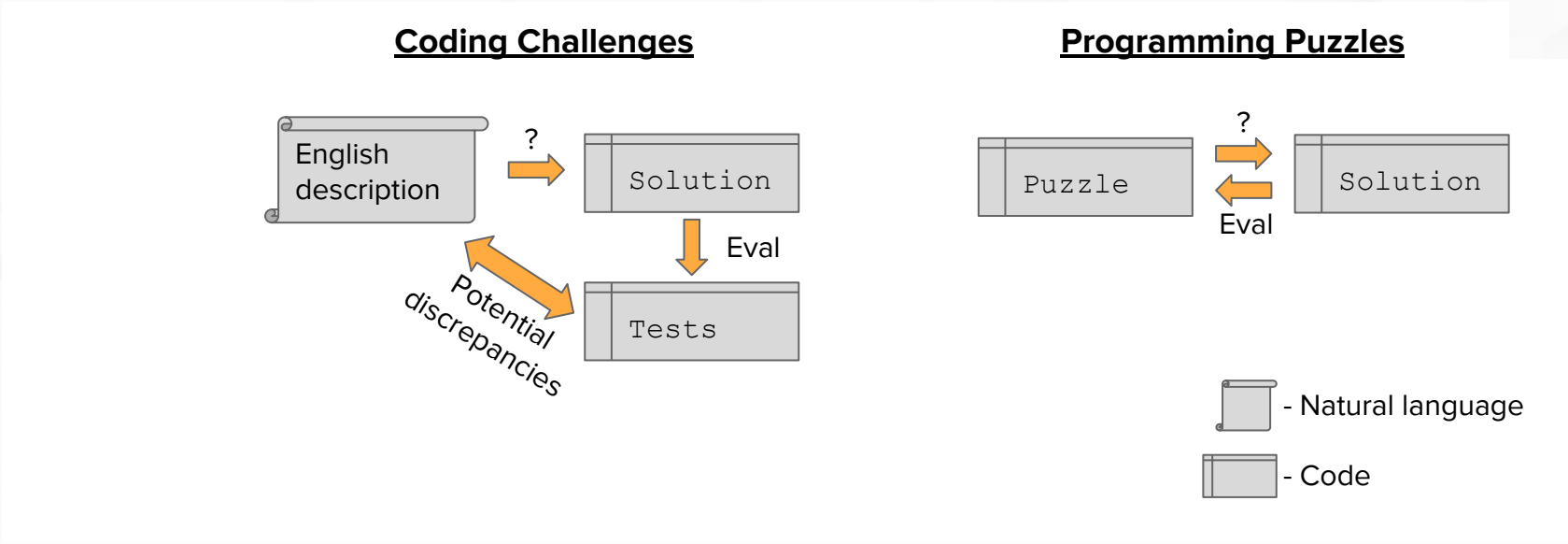
GitHub Copilot

- Need objective evaluation of coding proficiency to measure and spur progress

Pure code evaluation



Pure code evaluation



- Puzzles **focus on the algorithmic challenge**; not mixing with world knowledge or English
- Verification is straightforward and objective; **no hidden test cases**

Python Programming Puzzles (P3) dataset

- Large (and growing) collection of puzzles

  v0.1: 200 puzzles / Current size: **397 puzzles**

- **Comprehensive** in

  Domain

  Algorithmic tools

  Difficulty

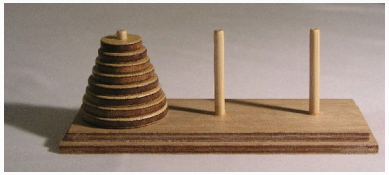
- Baselines

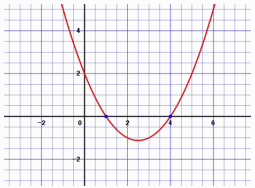
  Enumerative solvers and Language Models

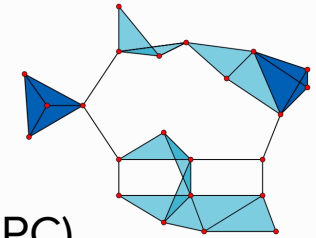
  Human programmers

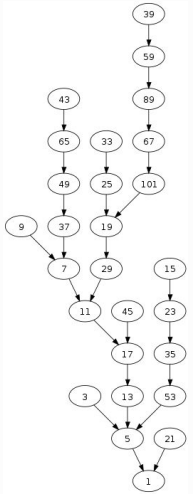
Comprehensive in domain

- Classic puzzles
- Games
- Basic Python programming
- Probability
- Algebra
- Number Theory
- Graphs
- Programming challenges
- International Collegiate Programming Contest (ICPC)
- International Mathematical Olympiad (IMO)
- ...









acm International Collegiate Programming Contest

Comprehensive in algorithmic tools

- Learning PL specific operations (e.g. string manipulation)
- Recursion
- Linear programming
- Dynamic programming
- Convex optimization
- Sorting
- Graph search
- ...

```python
# Find a string that when reversed and concatenated with "world" gives "Hello world"
def f1(y: str):
    return y[::-1] + "world" == "Hello world"

# Tower of Hanoi, often teaches recursion. Move [i, j] means move top disk on tower i to j, with 1 ≤ i,j ≤ 3
def f2(moves: List[List[int]], num_disks=8):
    state = [1] * num_disks  # All disks start at tower 1.
    for [i, j] in moves:
        assert state.index(i) <= (state + [1, 2, 3]).index(j), "bigger disk on top"
        state[state.index(i)] = j  # Move smallest disk from tower i to tower j.
    return state == [3] * num_disks  # All disks must end on tower 3.
```

Comprehensive in difficulty

- From trivial coding operations to major open algorithms and math problems
- Beating humans would mean scientific breakthroughs

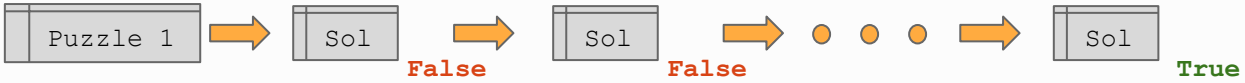
```python
def f1(s: str):   # find a string with 1000 o's but no consecutive o's.
    return s.count("o") == 1000 and s.count("oo") == 0

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 1 < d < n and n % d == 0
```

Collatz conjecture (unsolved)

```python
def sat(n: int):
    """
    Consider the following process. Start with an integer `n` and repeatedly applying the operation:
    * if n is even, divide n by 2,
    * if n is odd, multiply n by 3 and add 1
    Find n > 4 which is part of a cycle of this process
    """
    m = n
    while n > 4:
        n = 3 * n + 1 if n % 2 else n // 2
        if n == m:
            return True
```

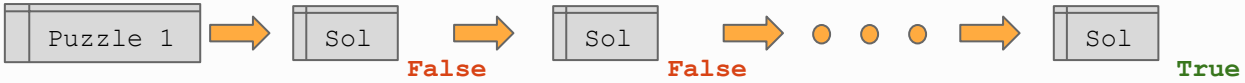
Evaluation (zero-shot)

- Number of required tries for finding a correct solution



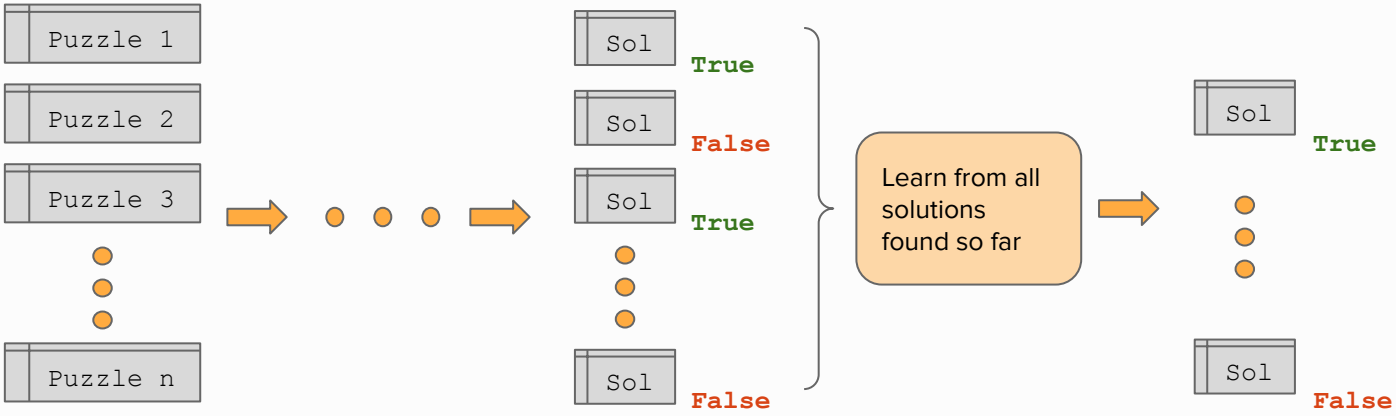
Evaluation

- Number of required tries for finding a correct solution



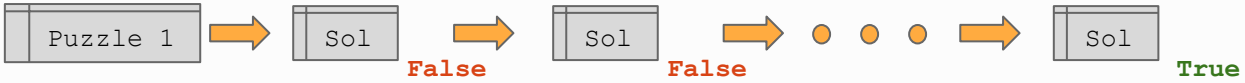
- **Test time bootstrapping**

  At step m, learn from the puzzles that were solved in $<m$ steps



Evaluation

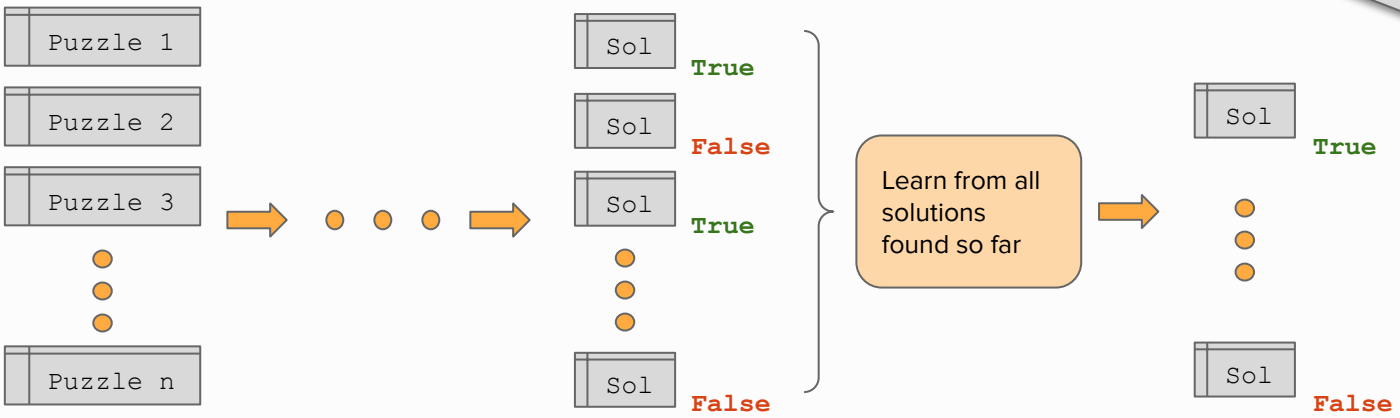
- Number of required tries for finding a correct solution



- **Test time bootstrapping**

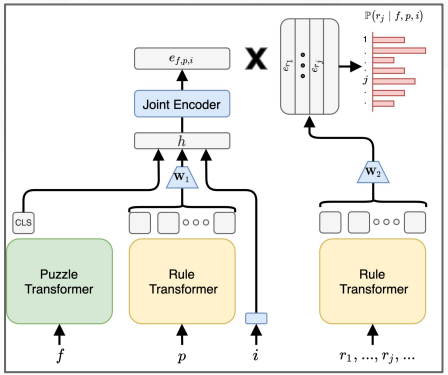
  At step m, learn from the puzzles that were solved in $<m$ steps

How many puzzles were solved with up to k tries



Solvers

- Enumerative
  - Uniform
  - Ranform forest
  - Transformer



Enumerative Transformer

- Language Models (GPT-3/ Codex)
  - Short (zero-shot)
  - Medium (five-shot)
  - Long (five-shot + EN description)

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

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

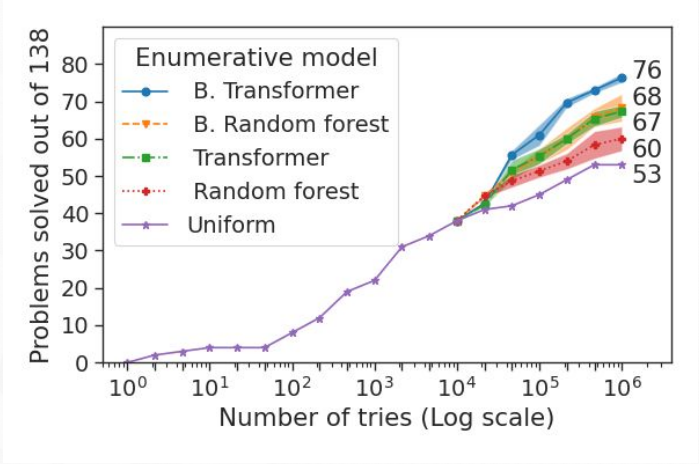
LM short prompt (zero-shot)

- Bootstrapping setting
  - Retrain model with new solutions (enumerative)/ add examples to prompt (LM)

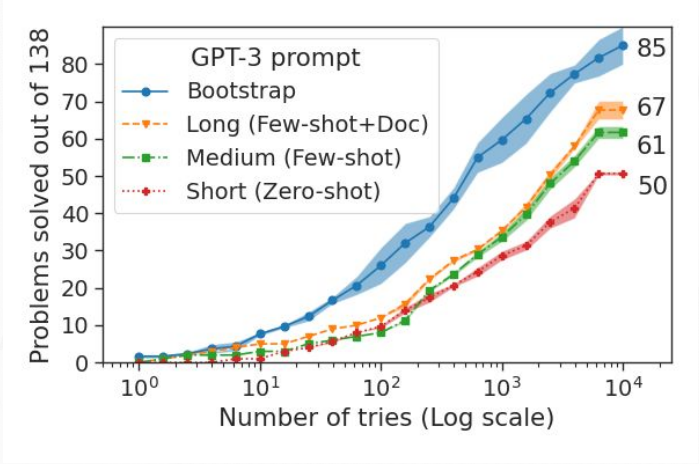
* See paper for more details

Results

Enumerative

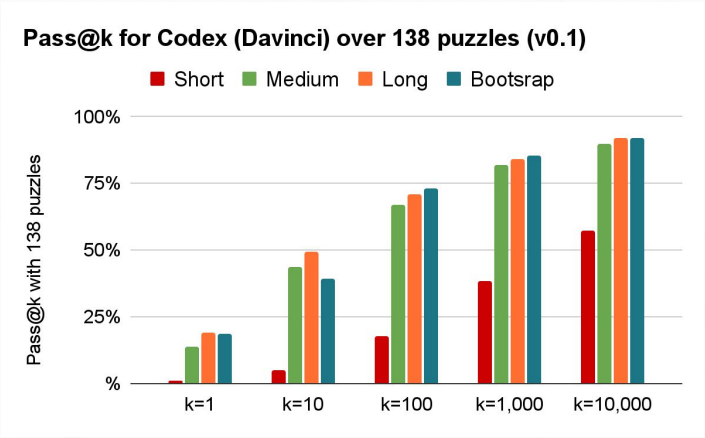


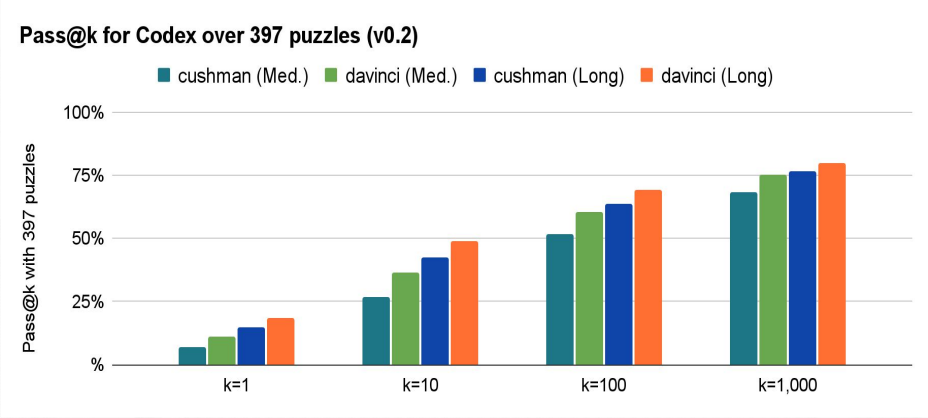
GPT-3



- More complex models perform better
- Large LMs outperform our enumerative baselines
- Learning from past solutions (bootstrapping) helps

Results - Codex model





- Larger version (Davinci) performs better
- Solves most of the puzzles (with enough tries)
- **Future challenges**:

  improve Pass@*k* for small *k*

  Add harder puzzles

Human study

- 21 participants with varying experience in Python
- Interface based on Jupyter notebook
- Up to 6 minutes per puzzle
- 30 puzzles

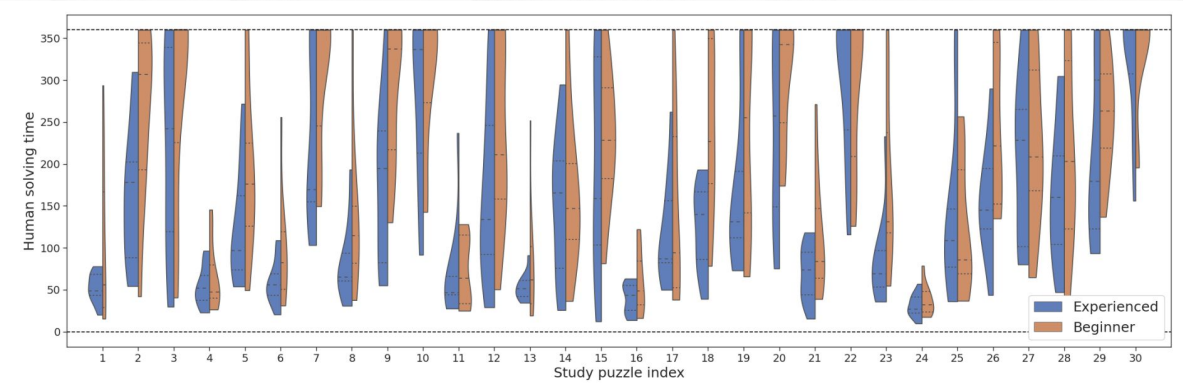
- Very positive feedback

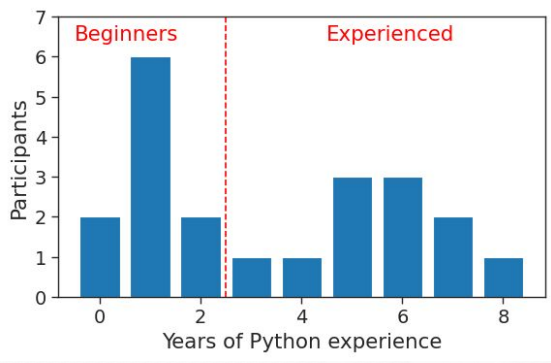

```
In [4]: # The first 3 puzzles are warmup. Begin the warmup part by running this cell.

        next_puzzle()

              Time:

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

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

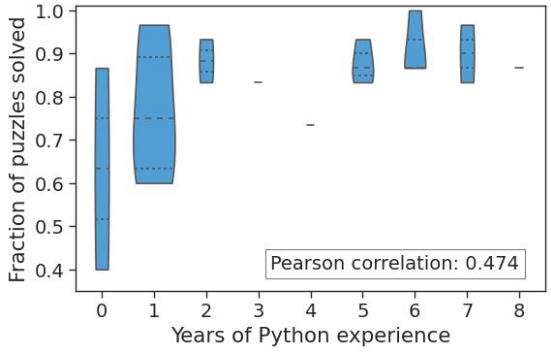
```
In [5]: # Solve the first puzzle by running this cell.

        puzzle('world')

        CORRECT in 00:39 sec.

Out[5]: True
```

Performance correlates with experience

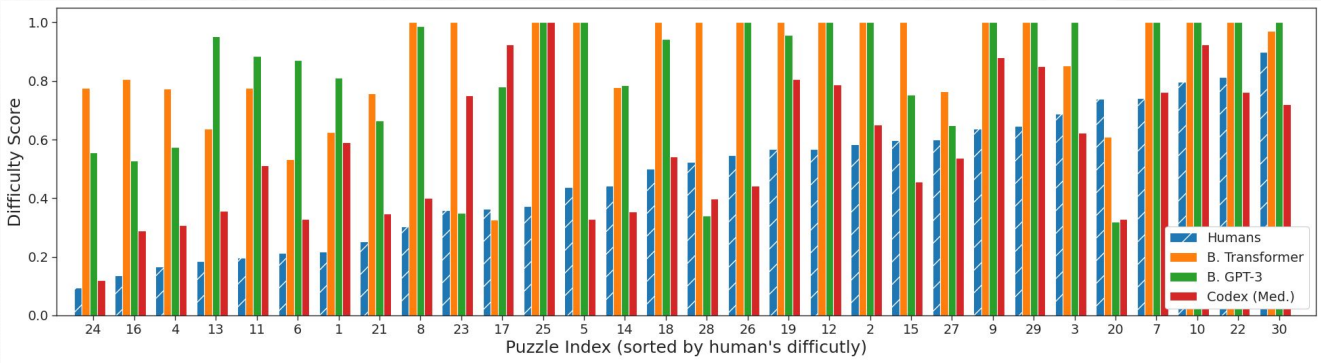
- Experienced coders solved more puzzles, faster







Human/ AI perceived difficulty

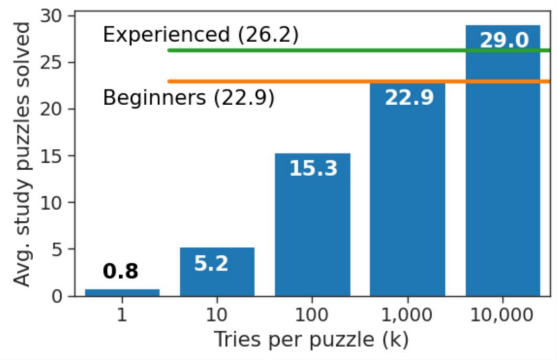


Pearson rank correlation (Human/ AI):

Model	All humans	Beginners	Experienced
Transformer	0.443	0.493	0.433
GPT-3	0.512	0.541	0.470
Codex	**0.563**	0.562	0.544

Human/ AI perceived difficulty

- Codex-davinci vs. human coders:



- Codex required up to 1K tries per puzzle to match the performance of beginner coders with up to 6 minutes per puzzle

Example

Puzzle:

- **Study_10** (1 instance)

```
def sat(s: str):
    """Find a palindrome of length greater than 11 in the decimal representation of 8^1818."""
    return s in str(8 ** 1818) and s == s[::-1] and len(s) > 11
```

Human-written solution:

▼ 1 hand-written solution

```
def sol():
    s = str(8 ** 1818)
    return next(s[i: i + le]
                for le in range(12, len(s) + 1)
                for i in range(len(s) - le + 1)
                if s[i: i + le] == s[i: i + le][::-1]
                )
```

Codex solutions:

▼ 16 AI solutions from codex (shortest and longest ones below)

```
def sol():
    x = str(8 ** 1818)
    return [x[i:(i+13)] for i in range(len(x)-11) if x[i:(i+13)] == x[i:(i+13)][::-1]][0]
```

```
def sol():
    s = str(8 ** 1818)
    n = len(s)

    pal = set()
    for i in range(11, 55):
        for j in range(n-i+1):
            pal.add(s[j:j+i])

    for p in pal:
        if p == p[::-1] and len(p) > 11:
            return p
```

Example 2: Recursion

Puzzle:

```
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):
```

Codex (Med.) solution:

```
    # 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
```

Conclusion

- P3: Large (and growing) collection of Programming Puzzles
  - Evaluating advancements of AI systems in programming proficiency
  - Can lead to scientific breakthroughs
- High correlation between AI and human perceived difficulty

- Dataset, solvers and online demos are available:

`Github.com/Microsoft/PythonProgrammingPuzzles`

- **Contributions are welcomed!**