

## 1 A Algorithmic Split of Datasets

2 We show the split of the BBEH datasets into algorithmic and non-algorithmic datasets in Table A.1.  
3 This split of datasets into the two groups is used in analyzing the results in the main body of the  
4 paper.

Table A.1: Percent of algorithmic problems in each dataset. We call the datasets with a majority of algorithmic problems the *algorithmic* datasets and the other datasets the *non-algorithmic* datasets.

Dataset	% Algorithmic
<b>Non-algorithmic Datasets</b>	
Leaf	0.000
Disambiguation qa	0.000
Sarc triples	0.000
Nycc	0.000
Movie recommendation	0.000
Hyperbaton	0.015
Geometric shapes	0.040
Causal understanding	0.040
Clevr	0.155
Linguini	0.175
Omnimath-4	0.305
Sportqa	0.345
Omnimath-3	0.410
<b>Algorithmic Datasets</b>	
Clutrr	0.750
Spatial reasoning	0.810
Omnimath-2	0.835
Buggy tables	0.940
Web of lies	0.940
Boardgame qa	0.955
Object properties	0.960
Boolean expressions	0.960
Time arithmetic	0.970
Word sorting	0.980
Dyck languages	0.980
Temporal sequence	0.990
Omnimath-1	0.995
Object counting	0.995
Zebra puzzles	0.995
Multistep arithmetic	1.000
Shuffled objects	1.000

## 5 B PIPS Programs for Figure 4

6 The two corresponding solutions with PIPS to the problems shown in Figure 4a and Figure 4b are  
7 included in Figure B.1 and Figure B.2 respectively.

## 8 C Full Results

9 The full results for Gemini-2.0-Flash, GPT 4.1-mini, and o4-mini and included in Table C.3, Table C.4,  
10 and Table C.2 respectively.

```

1 symbols = {'objects': [{'color': 'green', 'material': 'rubber', 'shape': 'cube',
↪ 'size': 'large', 'x': 0.1, 'y': 0.4},
2   {'color': 'blue', 'material': 'metal', 'shape': 'sphere', 'size': 'large',
↪ 'x': 0.4, 'y': 0.3},
3   {'color': 'yellow', 'material': 'metal', 'shape': 'cube', 'size': 'large',
↪ 'x': 0.6, 'y': 0.3},
4   {'color': 'cyan', 'material': 'rubber', 'shape': 'cylinder', 'size': 'small',
↪ 'x': 0.7, 'y': 0.3},
5   {'color': 'brown', 'material': 'rubber', 'shape': 'sphere', 'size': 'large',
↪ 'x': 0.5, 'y': 0.6}]}
6
7 def solve(symbols):
8     """
9     Calculates the number of large objects in front of the cyan object and behind
↪ the brown matte ball.
10
11     Args:
12         symbols: A dictionary containing information about the objects in the image.
13
14     Returns:
15         The number of large objects in front of the cyan object and behind the brown
↪ matte ball.
16     """
17
18     cyan_x = next((obj["x"] for obj in symbols["objects"] if obj["color"] ==
↪ "cyan"), None)
19     brown_x = next((obj["x"] for obj in symbols["objects"] if obj["color"] ==
↪ "brown"), None)
20
21     count = 0
22     for obj in symbols["objects"]:
23         if obj["size"] == "large":
24             if obj["x"] < cyan_x and obj["x"] > brown_x:
25                 count += 1
26
27     return count

```

Figure B.1: Final symbols and generated program from PIPS for the instance shown in Figure 4a. Executing the code from PIPS results in the correct answer of 1.

## 11 D Non-Trivial Program Synthesis on Non-Algorithmic Problems

12 We find that encouraging non-trivial programs for non-algorithmic problems leads to reduced per-  
13 formance. For instance, CoT with Gemini-2.0-Flash results in a harmonic mean accuracy over  
14 all non-algorithmic datasets (as listed in Table A.1) of 0.199 while PoT results in a lower value  
15 of 0.166 and our method without switching results in 0.151. Producing non-trivial programs for  
16 non-algorithmic problems which shouldn't be solved via code in the first place, harms performance.  
17 Therefore, a high performing general reasoning system needs to avoid program synthesis in such  
18 cases.

## 19 E Program Evaluation Criteria

20 The eight criteria we use to evaluate code within PIPS are included below. The input dependence  
21 criteria is meant to catch when the program is trivial, the proper output criteria catches cases where the  
22 program does not output the answer in the correct format, and we also include the symbol extraction  
23 issues criteria to find issues during the first step of symbol extraction.

- 24 • Input dependence: Does the code use the input symbols to compute the answer?
- 25 • Valid return: Does the code avoid returning None unless it is the correct answer?

```

1 symbols = {'objects': [
2     {'color': 'purple', 'material': 'rubber', 'shape': 'sphere', 'size': 'large',
3     ↪ 'x': 0.0},
4     {'color': 'purple', 'material': 'metal', 'shape': 'sphere', 'size': 'large',
5     ↪ 'x': 1.0},
6     {'color': 'blue', 'material': 'rubber', 'shape': 'cube', 'size': 'small', 'x':
7     ↪ 2.0},
8     {'color': 'gray', 'material': 'rubber', 'shape': 'cylinder', 'size': 'small',
9     ↪ 'x': 2.5},
10    {'color': 'brown', 'material': 'rubber', 'shape': 'cube', 'size': 'large',
11    ↪ 'x': 4.0},
12    {'color': 'yellow', 'material': 'rubber', 'shape': 'sphere', 'size': 'large',
13    ↪ 'x': 1.5},
14    {'color': 'brown', 'material': 'metal', 'shape': 'sphere', 'size': 'small',
15    ↪ 'x': 3.5},
16    {'color': 'red', 'material': 'metal', 'shape': 'cube', 'size': 'small', 'x':
17    ↪ 2.0},
18    {'color': 'red', 'material': 'metal', 'shape': 'cube', 'size': 'large', 'x':
19    ↪ 5.0}]]
20
21 def solve(symbols):
22     """
23     Finds the color of the tiny shiny object behind the big ball to the right of
24     ↪ the big metallic thing behind the big brown cube.
25
26     Args:
27         symbols (dict): A dictionary containing information about the objects in
28         ↪ the image.
29
30     Returns:
31         str: The color of the object.
32     """
33
34     objects = symbols["objects"]
35
36     # 1. Find the big brown cube
37     big_brown_cube = next((obj for obj in objects if obj["shape"] == "cube" and
38     ↪ obj["color"] == "brown" and obj["size"] == "large"), None)
39
40     # 2. Find the big metallic thing (red cube) to the right of the big brown cube
41     big_metallic_thing = next((obj for obj in objects if obj["shape"] == "cube"
42     ↪ and obj["color"] == "red" and obj["size"] == "large" and obj["x"] >
43     ↪ big_brown_cube["x"]), None)
44
45     # 3. Find the closest big ball
46     closest_big_ball = min((obj for obj in objects if obj["shape"] == "sphere" and
47     ↪ obj["size"] == "large"), key=lambda obj: abs(obj["x"] -
48     ↪ big_metallic_thing["x"]))
49
50     # 4. Find the tiny shiny object (gold sphere) behind the big ball
51     tiny_shiny_object = next((obj for obj in objects if obj["shape"] == "sphere"
52     ↪ and obj["material"] == "metal" and obj["size"] == "small" and obj["x"] >
53     ↪ closest_big_ball["x"]), None)
54
55     return tiny_shiny_object["color"]

```

Figure B.2: Final symbols and generated program from PIPS for the instance shown in Figure 4b. The code returns “brown” which is the correct answer.

Table C.2: Accuracy and non-trivial-code percentage for o4-mini. Best accuracy per row is bolded.

Dataset	CoT	PoT	PoT Non-Trivial (%)	PIPS	PIPS Non-Trivial (%)
Buggy tables	0.275	0.512	85.6%	<b>0.594</b>	96.9%
Temporal sequence	0.325	0.306	92.5%	<b>0.519</b>	96.2%
Dyck languages	<b>0.650</b>	0.637	8.8%	0.606	88.8%
Multistep arithmetic	0.281	0.600	72.5%	<b>0.631</b>	87.5%
Time arithmetic	0.875	<b>0.900</b>	76.2%	0.894	87.5%
Shuffled objects	0.081	0.150	43.1%	<b>0.344</b>	71.2%
Web of lies	0.388	0.344	68.1%	<b>0.525</b>	70.6%
Object counting	0.850	0.812	93.1%	<b>0.900</b>	70.0%
Zebra puzzles	0.150	0.100	62.5%	<b>0.231</b>	68.1%
Object properties	0.144	0.219	38.8%	<b>0.344</b>	65.6%
Boolean expressions	<b>0.550</b>	0.469	24.4%	0.419	63.7%
Spatial reasoning	0.463	0.506	60.6%	<b>0.550</b>	55.0%
Word sorting	<b>0.806</b>	0.775	46.2%	<b>0.806</b>	53.1%
Omnimath-2	<b>0.812</b>	0.706	56.9%	0.787	29.4%
Movie recommendation	<b>0.819</b>	0.744	10.6%	0.731	18.8%
Omnimath-1	0.925	0.925	78.1%	<b>0.944</b>	15.6%
Boardgame qa	<b>0.688</b>	0.637	10.6%	0.675	13.8%
Hyperbaton	<b>0.206</b>	0.194	34.4%	0.188	13.8%
Omnimath-3	<b>0.556</b>	0.356	36.9%	0.544	11.9%
Omnimath-4	<b>0.662</b>	0.419	31.2%	0.644	6.9%
Clutrr	0.762	<b>0.800</b>	1.2%	0.762	2.5%
Sportqa	<b>0.287</b>	0.256	0.0%	<b>0.287</b>	1.9%
Linguini	0.138	<b>0.175</b>	6.2%	0.138	1.2%
Clevr	<b>0.769</b>	0.750	6.2%	<b>0.769</b>	0.6%
Causal understanding	<b>0.581</b>	0.550	0.6%	<b>0.581</b>	0.6%
Leaf	0.364	<b>0.455</b>	0.0%	0.364	0.0%
Geometric shapes	0.056	<b>0.119</b>	0.0%	0.087	0.0%
Disambiguation qa	0.562	<b>0.573</b>	0.0%	0.562	0.0%
Sarc triples	<b>0.338</b>	0.300	0.0%	<b>0.338</b>	0.0%
Nycc	<b>0.231</b>	0.150	0.0%	<b>0.231</b>	0.0%
Harmonic Mean	0.304	0.340	0.0%	0.397	0.1%

- Proper output: Does the code return (not print) the correct answer in the expected format?
- No example usage: Does the code omit example calls or usage?
- Simplifiability: Could the solution be implemented in a simpler way?
- Correctness bugs: Are there any bugs affecting correctness?
- Symbol extraction issues: Are there any problems with the extracted input symbols?
- Sanity check: Does the output pass a basic sanity check?

## F Switch Analysis

In this section, we go in-depth on the CoT vs. program synthesis switch design.

### F.1 Switch Criteria

The full prompt for the switch is provided in Appendix G, but we provide the 10 criteria we use within the prompt below. The criteria for determining if an instance should be solved directly via CoT or by program synthesis are the following:

1. Simple formalizability: Likelihood that the solution can be easily expressed as simple, deterministic code.
2. Straightforward executability: Likelihood that a first code attempt runs correctly without debugging.

Table C.3: Accuracy and non-trivial-code percentage for Gemini-2.0-Flash. Best accuracy per row is bolded.

Dataset	CoT	PoT	PoT Non-Trivial (%)	CI	PIPS	PIPS Non-Trivial (%)
Shuffled objects	0.094	0.025	35.6%	<b>0.537</b>	0.188	98.1%
Buggy tables	0.019	0.100	99.4%	0.031	<b>0.188</b>	97.5%
Time arithmetic	0.438	0.331	82.5%	0.312	<b>0.475</b>	97.5%
Temporal sequence	0.006	0.006	42.5%	0.006	<b>0.094</b>	96.2%
Multistep arithmetic	<b>0.144</b>	0.037	87.5%	0.087	0.119	90.6%
Dyck languages	<b>0.119</b>	0.081	1.9%	0.106	0.050	90.0%
Boolean expressions	0.294	0.219	65.6%	0.325	<b>0.456</b>	86.2%
Object counting	0.144	0.119	98.1%	0.181	<b>0.281</b>	85.0%
Omnimath-1	0.850	0.838	57.5%	0.844	<b>0.869</b>	84.4%
Word sorting	0.287	0.525	48.8%	0.338	<b>0.556</b>	73.1%
Object properties	0.006	0.062	26.9%	0.125	<b>0.163</b>	71.9%
Spatial reasoning	0.231	<b>0.237</b>	12.5%	0.219	0.231	70.6%
Clevr	0.637	0.619	26.2%	0.669	<b>0.688</b>	46.2%
Causal understanding	0.537	0.438	1.2%	<b>0.544</b>	0.537	15.6%
Clutrr	0.556	0.588	0.0%	<b>0.662</b>	0.506	13.8%
Linguini	<b>0.144</b>	0.113	1.9%	0.119	0.125	13.8%
Boardgame qa	<b>0.463</b>	0.419	5.0%	0.394	<b>0.463</b>	11.9%
Zebra puzzles	<b>0.300</b>	0.256	73.8%	0.131	0.275	10.0%
Geometric shapes	0.312	0.269	0.6%	<b>0.388</b>	0.300	9.4%
Sportqa	0.200	0.244	1.9%	<b>0.269</b>	0.194	6.9%
Hyperbaton	<b>0.031</b>	0.019	46.2%	<b>0.031</b>	0.025	5.6%
Web of lies	<b>0.219</b>	0.206	3.1%	0.188	<b>0.219</b>	2.5%
Movie recommendation	<b>0.581</b>	0.562	0.0%	0.556	0.569	2.5%
Disambiguation qa	0.448	0.417	0.0%	<b>0.479</b>	0.448	1.0%
Leaf	0.602	<b>0.636</b>	0.0%	0.102	0.602	0.0%
Sarc triples	<b>0.375</b>	0.369	0.0%	0.344	<b>0.375</b>	0.0%
Nycc	0.106	<b>0.131</b>	0.0%	0.113	0.106	0.0%
Omnimath-2	<b>0.544</b>	0.463	46.9%	<b>0.544</b>	<b>0.544</b>	0.0%
Omnimath-3	0.269	0.194	15.6%	<b>0.275</b>	0.269	0.0%
Omnimath-4	0.312	0.244	16.2%	<b>0.319</b>	0.312	0.0%
Harmonic Mean	0.107	0.115	0.0%	0.134	0.201	0.0%

3. Robust systematic search: Likelihood that systematic code (e.g., brute-force, recursion) reliably solves the problem.
  4. Manageable state representation: Likelihood that all necessary variables and concepts can be cleanly represented in code.
  5. Structured knowledge encoding: Likelihood that required background knowledge can be encoded as rules or data.
  6. Hallucination risk reduction: Likelihood that code avoids fabricated steps better than chain-of-thought reasoning.
  7. Arithmetic and data processing advantage: Likelihood that code handles arithmetic or data processing more reliably.
  8. Branching and case handling advantage: Likelihood that code handles special cases or branching logic more systematically.
  9. Algorithmic reliability over heuristics: Likelihood that a deterministic algorithm outperforms intuitive reasoning.
  10. Overall comparative success: Likelihood that code yields a more reliable solution than chain-of-thought reasoning.
- Our full prompt asks the LLM itself to quantify each of these criteria and then we build a simple logistic classifier based on the LLM’s own judgements to determine when to use program synthesis.

Table C.4: Accuracy and non-trivial-code percentage for gpt-4.1-mini-2025-04-14. Best accuracy per row is bolded.

Dataset	CoT	PoT	PoT Non-Trivial (%)	PIPS	PIPS Non-Trivial (%)
Time arithmetic	0.588	<b>0.706</b>	93.1%	0.463	98.8%
Buggy tables	0.075	<b>0.481</b>	100.0%	0.406	98.1%
Multistep arithmetic	0.275	0.394	86.2%	<b>0.494</b>	96.2%
Dyck languages	0.150	0.175	6.9%	<b>0.506</b>	95.0%
Shuffled objects	0.119	0.256	68.8%	<b>0.294</b>	95.0%
Omnimath-1	<b>0.894</b>	0.875	76.2%	0.875	89.4%
Word sorting	<b>0.681</b>	0.600	73.1%	0.656	85.6%
Object counting	0.263	<b>0.331</b>	66.2%	0.287	83.1%
Temporal sequence	0.250	<b>0.356</b>	96.2%	0.275	82.5%
Boolean expressions	0.294	0.356	11.2%	<b>0.469</b>	79.4%
Spatial reasoning	0.181	<b>0.394</b>	48.1%	0.362	68.1%
Object properties	0.025	<b>0.237</b>	67.5%	0.175	60.0%
Omnimath-2	<b>0.569</b>	<b>0.569</b>	63.1%	0.556	59.4%
Web of lies	<b>0.362</b>	0.300	36.2%	0.219	58.8%
Clevr	<b>0.719</b>	0.669	8.1%	0.700	53.8%
Zebra puzzles	<b>0.194</b>	0.150	36.2%	0.188	49.4%
Clutrr	<b>0.662</b>	0.562	0.6%	0.613	21.9%
Omnimath-3	<b>0.338</b>	0.244	35.6%	0.325	16.2%
Geometric shapes	<b>0.344</b>	0.294	16.9%	0.319	15.0%
Boardgame qa	0.512	0.500	81.2%	<b>0.519</b>	11.2%
Omnimath-4	<b>0.463</b>	0.312	28.7%	0.431	10.0%
Sportqa	0.169	<b>0.244</b>	2.5%	0.200	9.4%
Hyperbaton	<b>0.087</b>	0.062	4.4%	0.075	7.5%
Causal understanding	<b>0.562</b>	<b>0.562</b>	1.2%	0.550	5.6%
Linguini	0.094	<b>0.144</b>	1.2%	0.094	3.1%
Movie recommendation	<b>0.606</b>	0.475	8.8%	0.594	1.9%
Leaf	0.341	<b>0.409</b>	0.0%	0.341	0.0%
Disambiguation qa	<b>0.552</b>	0.500	1.0%	<b>0.552</b>	0.0%
Sarc triples	0.287	<b>0.331</b>	10.0%	0.287	0.0%
Nycc	<b>0.188</b>	0.150	15.0%	<b>0.188</b>	0.0%
Harmonic Mean	0.211	0.297	0.3%	0.305	0.1%

Table C.5: Harmonic Mean Accuracy (All Datasets)

Method	gpt-4.1-mini	Gemini-2.0-Flash	o4-mini
CoT	0.211	0.107	0.304
PoT	0.297	0.115	0.340
PIPS	0.305	0.201	0.397
CI	0.000	0.134	0.000

## 60 F.2 Examples of Switch Decisions

61 We show two questions from BBEH Word Sorting in Figure F.3 and the corresponding switch  
62 decisions for the different models. We also show a question from Omnimath-2 in Figure F.4 where  
63 the different CoT/Synthesis switch decisions are made for different models.

## 64 F.3 Question Analysis

65 We include the logistic regression weights for each of the ten questions in Appendix F.1 in Table F.8.  
66 The most important questions for the switch actually vary significantly between models. For Gemini-  
67 2.0-Flash, question 5 and 6 are the most important while for GPT 4.1-mini it is question 8 and 5.  
68 Finally, for o4-mini, questions 2 and 1 are the most important. Interestingly, we see that questions 5,  
69 6, and 8 which are important for the non-reasoning models are related to the ability of the model to

Table C.6: Harmonic Mean Accuracy (Algorithmic Datasets)

Method	gpt-4.1-mini	Gemini-2.0-Flash	o4-mini
CoT	0.187	0.079	0.339
PoT	0.357	0.094	0.389
PIPS	0.369	0.217	0.548
CI	0.000	0.112	0.000

Table C.7: Harmonic Mean Accuracy (Non-Algorithmic Datasets)

Method	gpt-4.1-mini	Gemini-2.0-Flash	o4-mini
CoT	0.254	0.199	0.267
PoT	0.244	0.166	0.291
PIPS	0.248	0.184	0.292
CI	0.000	0.181	0.000

70 produce error-free code and avoid tedious steps using code while the reasoning model relies most on  
71 the criteria which concerns traditional problem algorithmicity rather than model capability.

Table F.8: Logistic regression coefficients for the switch for each of the three models.

Model	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Gemini-2.0-Flash	0.14	0.03	0.12	0.15	0.21	-0.21	0.18	-0.09	0.03	0.10
gpt-4.1-mini	0.40	0.42	0.20	0.28	0.22	0.15	-0.02	0.01	0.21	0.21
o4-mini	0.22	0.04	0.16	0.24	0.27	-0.05	0.12	0.35	0.18	0.24

## 72 G Prompts

73 All prompts used in our evaluation and method are included below.

74 The prompt used to create the LLM-based classifier for question algorithmicity is the following.

### Algorithmic Question Evaluation

You will determine whether a given target question can be definitively solved by  
writing a Python program (algorithmic) or if it necessitates another form  
of reasoning (non-algorithmic). A Python solution may import standard  
libraries, but cannot simply invoke external services, APIs, or LLMs.  
If the input contains images, an algorithmic solution may use information  
manually extracted without needing to interpret the image itself.

Evaluate the target question carefully against the following criteria. Answer  
each sub-question rigorously with a binary response (1 for yes, 0 for no),  
ensuring a high threshold for certainty:

1. Does the problem have explicitly defined inputs and outputs, such that  
identical inputs always yield identical outputs?
2. Are there explicit, clearly stated rules, formulas, algorithms, or known  
deterministic procedures available for solving this problem?
3. Does solving this problem strictly require exact computation (no  
approximations, intuition, or interpretation)?
4. Can this problem be fully formalized in clear mathematical, logical, or  
structured computational terms without ambiguity?
5. Can the solution method be decomposed into a finite, clear, and unambiguous  
sequence of computational steps?
6. Is there a universally recognized and objective standard for verifying the  
correctness of the solution?
7. Does solving the problem inherently involve repetitive or iterative  
computations clearly suitable for automation?
8. Are the inputs structured, quantifiable, and inherently suited to algorithmic  
manipulation?

75

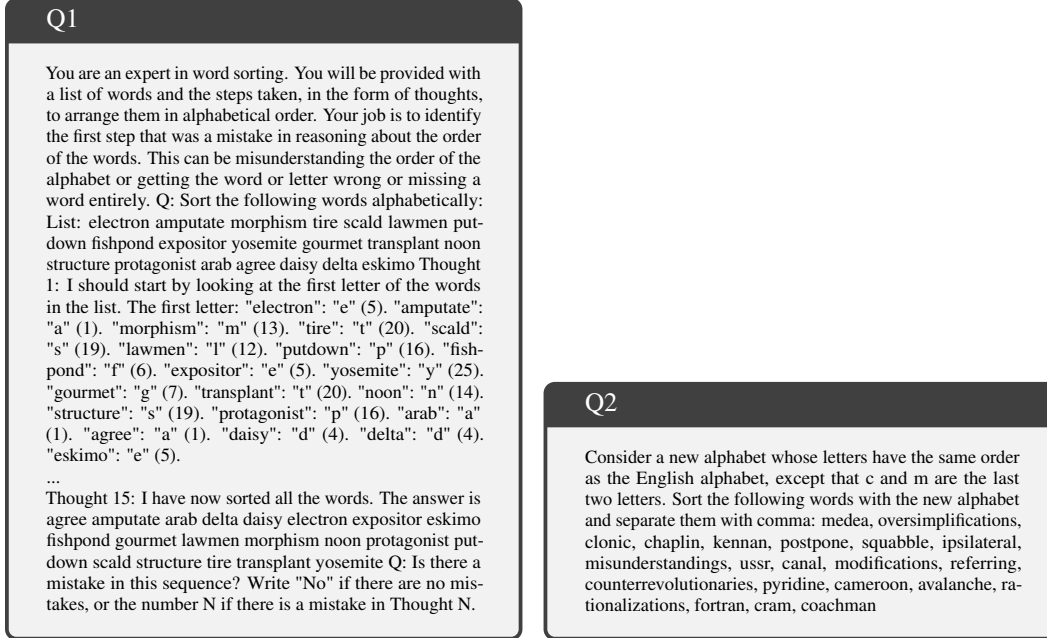


Figure F.3: Two questions from the BBEH Word Sorting task where the first question asks for determining which thought in a model's CoT is incorrect and the second question asks for sorting a list of words. The switch for these problems chooses to answer Q1 with CoT for all models while ansewr Q2 with program synthesis for all models.

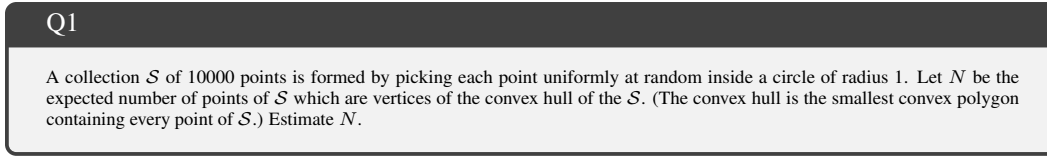
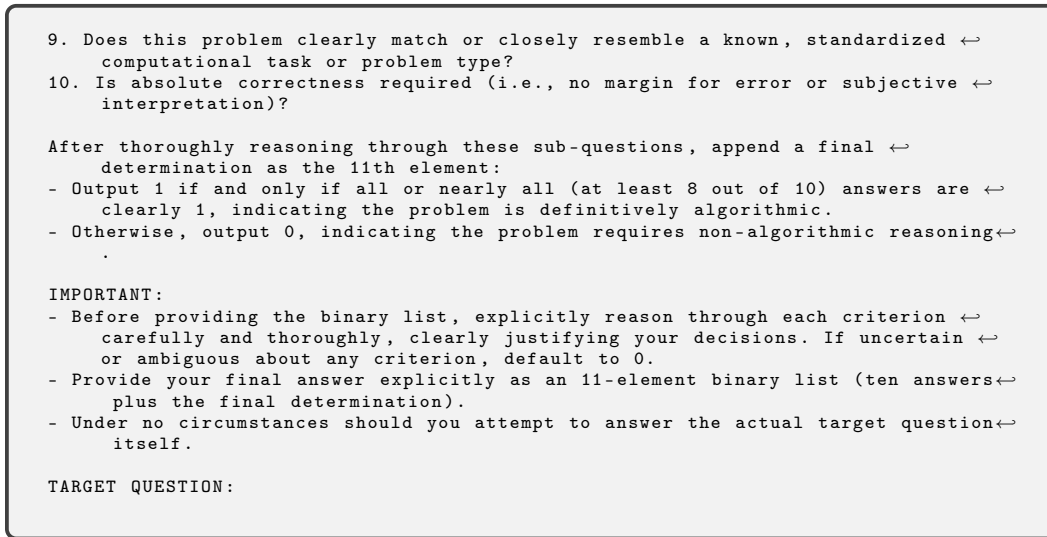


Figure F.4: A question from Omnimath-2 where both Gemini-2.0-Flash and GPT 4.1-mini switch to CoT to answer while o4-mini chooses program synthesis.



76

77 The prompt used to extract the ten criteria for building our instance-level CoT or program synthesis

78 switch is provided next.



## CoT or Synthesis Switch Criteria

You will self-reflect to estimate whether you are more likely to correctly solve a given target question by writing executable Python code or by using chain-of-thought (natural-language) reasoning.

### **\*\*IMPORTANT:\*\***

- This is a hypothetical evaluation.
- **\*\*You must NOT attempt to answer, solve, write code, or reason through the target question yet.\*\***
- Instead, you must reflect carefully and conservatively on your expected ability if you were to attempt solving the question through either method.

### Solution Expectations:

- You may assume standard library modules are allowed for code.
- You may NOT call external services, APIs, databases, or other LLMs.
- The code must be self-contained and executable without internet access.
- Chain-of-thought reasoning must be clear, logically sound, and internally verifiable without external tools.

### **\*\*CRITICAL GUIDANCE:\*\***

- **\*\*Be cautious, not optimistic.\*\*** Overestimating your capabilities will lead to choosing a method you cannot successfully complete.
- **\*\*If you feel any uncertainty, complexity, or ambiguity, lower your probability accordingly.\*\***
- **\*\*Assume that even small mistakes can cause failure\*\*** when writing code or reasoning through complex tasks.
- **\*\*Use conservative estimates.\*\***
- If unsure between two options, **\*\*prefer lower probabilities rather than guessing high\*\***.

Here are the self-reflection sub-questions you must answer hypothetically:

1. **\*\*Simple Formalizability\*\*** - \*What is the probability that the full solution can be easily and directly expressed as simple, deterministic code, without needing complex transformations or deep insight?\*
2. **\*\*Straightforward Executability\*\*** - \*What is the probability that a first attempt at writing code would execute correctly without needing debugging, even if the problem has subtle or complex aspects?\*
3. **\*\*Robust Systematic Search\*\*** - \*What is the probability that coding a systematic method (like brute-force search or recursion) would reliably find the correct answer, without missing hidden constraints or introducing edge-case errors?\*
4. **\*\*Manageable State Representation\*\*** - \*What is the probability that all intermediate concepts, variables, and conditions can be simply and explicitly represented in code, without requiring difficult or error-prone state tracking?\*
5. **\*\*Structured Knowledge Encoding\*\*** - \*What is the probability that all required background knowledge can be neatly encoded in code (e.g., as rules, formulas, or data), rather than needing flexible, intuitive understanding better suited to reasoning?\*
6. **\*\*Hallucination Risk Reduction\*\*** - \*What is the probability that code execution would more reliably avoid fabricated steps or unwarranted assumptions compared to chain-of-thought reasoning?\*
7. **\*\*Arithmetic and Data Processing Advantage\*\*** - \*What is the probability that the problem requires extensive or error-prone arithmetic/data handling that code could perform perfectly, but that chain-of-thought would likely fumble?\*
8. **\*\*Branching and Case Handling Advantage\*\*** - \*What is the probability that the solution involves many branching conditions, special cases, or exceptions that code can handle systematically but chain-of-thought might overlook?\*
9. **\*\*Algorithmic Reliability Over Heuristics\*\*** - \*What is the probability that following a deterministic algorithm in code would reach the correct answer more reliably than relying on intuitive or heuristic chain-of-thought reasoning?\*

```

10. **Overall Comparative Success** - *Considering all factors, what is the ←
    probability that code will ultimately produce a correct solution more ←
    reliably than chain-of-thought reasoning for this question?*

After thoroughly reasoning through each criterion:

- Output a single list of 10 probability scores (each between 0 and 1) as your ←
  FINAL ANSWER, in order:
  - Scores 1-10 correspond to the ten sub-questions above.

**Additional Instructions:**
- Explicitly reason through each criterion carefully before giving a probability ←
  .
- If uncertain or if the problem seems complex, favor lower probabilities to ←
  reflect the difficulty.
- Make sure to put only the list after FINAL ANSWER.
- **Under no circumstances should you write, sketch, pseudocode, or attempt any ←
  part of the solution itself during this reflection phase.**

TARGET QUESTION:

```

80

81 The prompt for generating the first program with PIPS in iteration one is the following.

### PIPS Code Generator (Iteration 0)

```

You will be given a question and you must answer it by extracting relevant ←
symbols in JSON format and then writing a Python program to calculate the ←
final answer.

You MUST always plan extensively before outputting any symbols or code.

You MUST iterate and keep going until the problem is solved.

# Workflow

## Problem Solving Steps
1. First extract relevant information from the input as JSON. Try to represent ←
   the relevant information in as much of a structured format as possible to ←
   help with further reasoning/processing.
2. Using the information extracted, determine a reasonable approach to solving ←
   the problem using code, such that executing the code will return the final ←
   answer.
3. Write a Python program to calculate and return the final answer. Use comments ←
   to explain the structure of the code and do not use a main() function.
The JSON must be enclosed in a markdown code block and the Python function must ←
be in a separate markdown code block and be called `solve` and accept a ←
single input called `symbols` representing the JSON information extracted. ←
Do not include any `if __name__ == "__main__":` statement and you can assume ←
the JSON will be loaded into the variable called `symbols` by the user.
The Python code should not just return the answer or perform all reasoning in ←
comments and instead leverage the code itself to perform the reasoning.
Be careful that the code returns the answer as expected by the question, for ←
instance, if the question is multiple choice, the code must return the ←
choice as described in the question.
Be sure to always output a JSON code block and a Python code block.

```

82

83 The prompt used to generate subsequent code solutions with PIPS by leveraging the evaluator output  
84 is show below.

### PIPS Code Generator (Iteration > 0)

```

Please fix the issues with the code and symbols or output "FINISHED".
The following is the result of evaluating the above code with the extracted ←
symbols.
...
Return value: {output}
Standard output: {stdout}
Exceptions: {err}
...

```

85

```

The following is the summary of issues found with the code or the extracted ↵
symbols by another model:
...
{checker_output}
...

If there are any issues which impact the correctness of the answer, please ↵
output code which does not have the issues. Before outputting any code, ↵
plan how the code will solve the problem and avoid the issues.
If stuck, try outputting different code to solve the problem in a different way.
You may also revise the extracted symbols. To do this, output the revised ↵
symbols in a JSON code block. Only include information in the JSON which is ↵
present in the original input to keep the code grounded in the specific ↵
problem. Some examples of symbol revisions are changing the names of ↵
certain symbols, providing further granularity, and adding information ↵
which was originally missed.
If everything is correct, output the word "FINISHED" and nothing else.

```

86

87 Finally, the prompt for the code evaluator is show below.

### PIPS Evaluator

```

You will be given a question and a code solution and you must judge the quality ↵
of the code for solving the problem.

Look for any of the following issues in the code:
- The code should be input dependent, meaning it should use the input symbols to ↵
  compute the answer. It is OK for the code to be specialized to the input (↵
  i.e. the reasoning itself may be hardcoded, like a decision tree where the ↵
  branches are hardcoded).
- The code should not return None unless "None" is the correct answer.
- The code should return the answer, not just print it. If the question asks for ↵
  a multiple choice answer, the code should return the choice as described ↵
  in the question.
- There should not be any example usage of the code.
- If there is a simpler way to solve the problem, please describe it.
- If there are any clear bugs in the code which impact the correctness of the ↵
  answer, please describe them.
- If there are any issues with the extracted symbols, please describe them as ↵
  well, but separate these issues from the issues with the code.
- If it is possible to sanity check the output of the code, please do so and ↵
  describe if there are any obvious issues with the output and how the code ↵
  could be fixed to avoid these issues.

After analyzing the code in depth, output a concrete and concise summary of the ↵
issues that are present, do not include any code examples. Please order the ↵
issues by impact on answer correctness.
The following are extracted symbols from the question in JSON format followed by ↵
a Python program which takes the JSON as an argument called `symbols` and ↵
computes the answer.
```json
{json_str}
```

```python
{code_str}
```

Code execution result:
```
Return value: {output}
Standard output: {stdout}
Exceptions: {err}
```

Output a concrete and concise summary of only the issues that are present, do ↵
not include any code examples.

```

88

## 89 H Hyperparameters

90 For all models we used a temperature of 0.0. For PIPS, we used a maximum of 30 iterations for all  
 91 models.

## 92 I Compute Resources

93 For most experiments we rely on API model access. All experiments \$300 for Gemini-2.0-Flash, for  
94 \$70 GPT 4.1-mini, and \$700 for o4-mini (medium). Other experiments were run on a server with 96  
95 Intel(R) Xeon(R) Gold 5318Y CPUs @ 2.10GHz with 1TB of system RAM. The server also had 10x  
96 NVIDIA A100 80GB GPUs which were only used for local testing of open-weights models.

## 97 J Broader Impacts

98 PIPS offers significant potential benefits, including enhanced AI reliability, trust, transparency, and  
99 the democratization of advanced problem-solving. However, like most systems built from powerful  
100 foundation models, it also presents important considerations. These include the potential for bias  
101 propagation from underlying models, challenges in ensuring the complexity and robustness of dy-  
102 namically generated programs, and concerns regarding misuse. Continued research and development  
103 must prioritize robust safeguards, fairness, transparency, and responsible deployment practices to  
104 harness the benefits while mitigating these potential negative impacts.

## 105 K Connection to Transductive Learning

106 Instance-wise Program Synthesis can be connected to transductive learning [38] where a function is  
107 learned to map from specific function inputs to specific outputs. The general philosophy is that one  
108 should not try to solve the general problem when the specific case is all one needs.

109 We are given a labeled training set  $D_L = \{(x_i, y_i)\}_{i=1}^m \in \mathcal{R} \times \mathcal{Y}$  where  $\mathcal{R}$  is the space of symbolic  
110 input and  $\mathcal{Y}$  is the output space, and an unlabeled test sample  $x \in \mathcal{R}$ . Our goal is to find a program  
111  $p : \mathcal{R} \rightarrow \mathcal{Y}$  which has a low error on the given sample, so  $p(x) = y$ . In contrast, inductive program  
112 synthesis tries to find a program  $p$  which *generalizes*, so it should achieve a low error on samples from  
113 the same distribution as  $D_L$ . To perform transductive program synthesis, we rely on minimizing an  
114 auxiliary “regularization” term,  $\Omega$ , over the test sample, representing a form of test-time computation:

$$\hat{p} = \arg \min_p \left\{ \frac{1}{m} \sum_{i=1}^m \ell(p(x_i), y_i) + \Omega(p; x_1, \dots, x_m, x) \right\}.$$

115 The regularization term can be implemented using probabilistic priors (e.g. the likelihood under some  
116 language model) or more complex heuristics involving static/dynamic code analysis. With powerful  
117 foundation models that can perform zero-shot inference, we can even perform this type of learning  
118 without any training set.

119 **Neuro-symbolic Instance-level Synthesis** The above problem definition is specifically for prob-  
120 lems formulated for program synthesis, meaning the inputs are expressed in symbolic form. Can we  
121 apply such a method for problems expressed in natural language or even using images? To convert  
122 general problems into a form which is conducive to program execution, we use a Neuro-Symbolic  
123 framework, specifically a Prompt-Symbolic approach.

124 We now assume our samples come from an arbitrary raw input space, such as natural language,  
125 images, and other modalities. Using the traditional neuro-symbolic framework, we first extract  
126 symbols from the raw input and then pass these symbols to a program. Let  $C : \mathcal{X} \rightarrow \mathcal{R}$  be a mapping  
127 from raw data to symbols. Now we adapt the above formalism as the following:

$$\hat{p}, \hat{C} = \arg \min_{p, C} \left\{ \frac{1}{m} \sum_{i=1}^m \ell(p(C(x_i)), y_i) + \Omega(C, p; x_1, \dots, x_m, x) \right\}.$$

128 Intuitively, we want to find a raw data to symbols mapping  $\hat{C}$  and program  $\hat{p}$  which perform well on  
129 a training set (if one is available) and minimize the loss  $\Omega$  over the given test samples. Notably, the  
130 program and raw data to symbol mapping do not need to be general, they should be *specialized* for

131 the test samples. In practice, the raw data to symbol mapping is often a form of semantic parsing,  
132 which can be done through zero-shot prompting of foundation models, so the raw data to symbol  
133 mapping is not modified on an instance level.