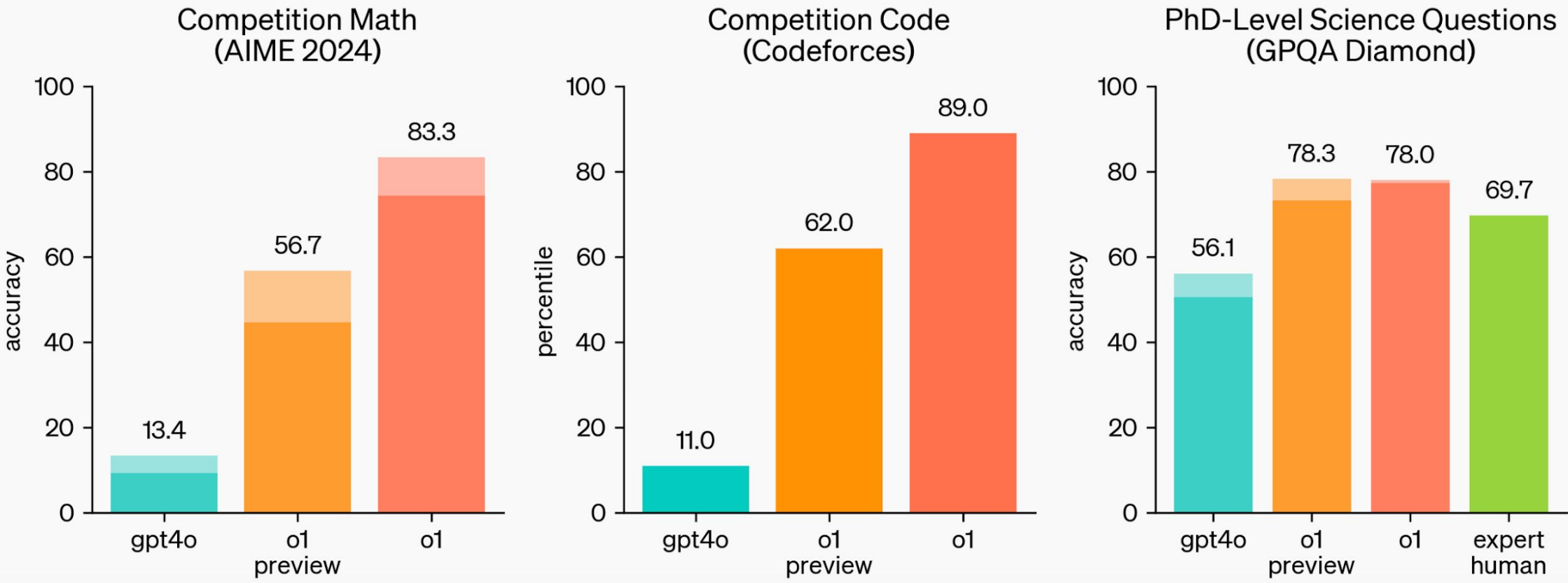


# Inference-Time Techniques for LLM Reasoning

Xinyun Chen

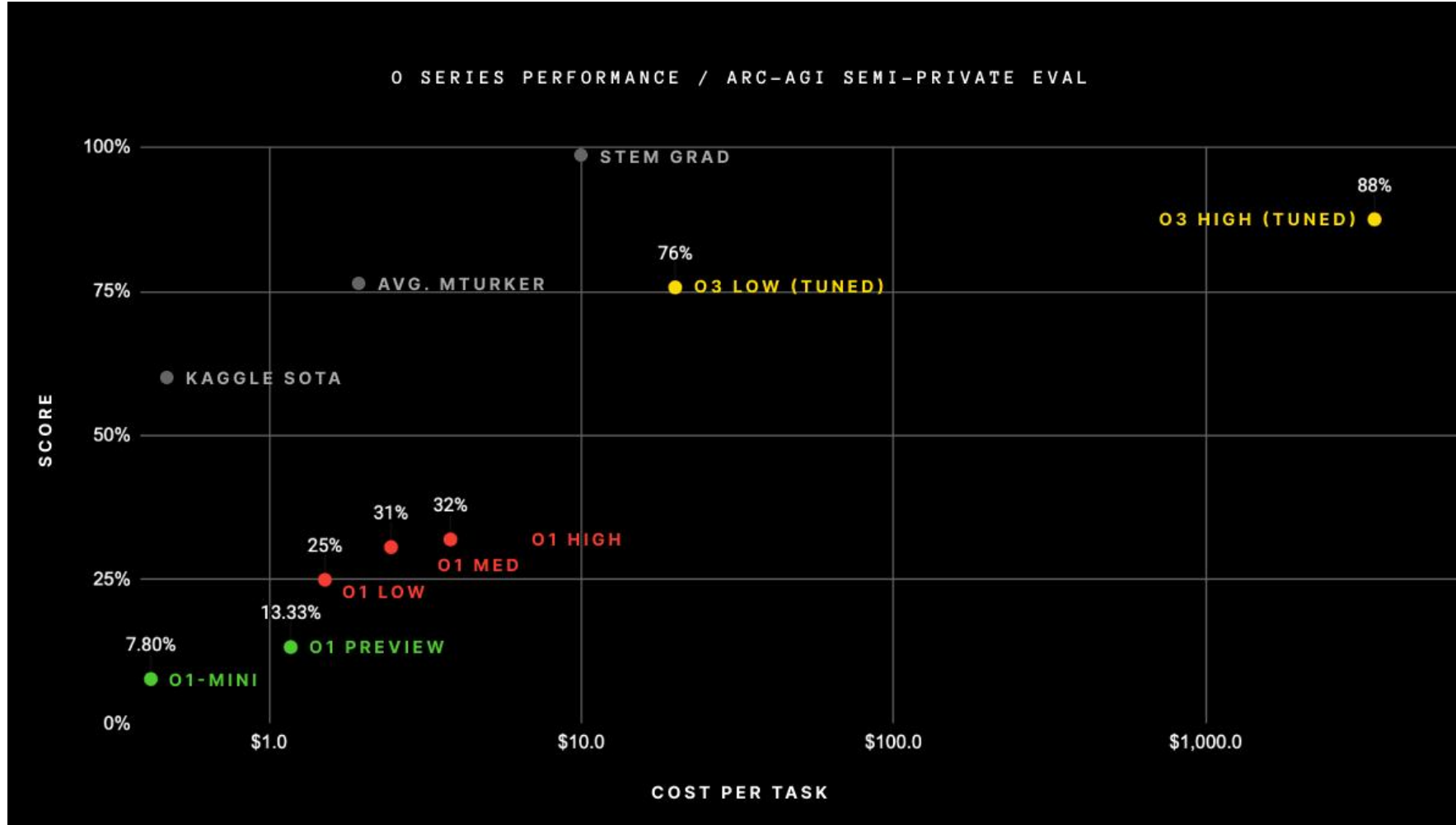
 DeepMind

# Highlight of LLMs in 2024: the advancement of reasoning models



OpenAI o1 started to achieve impressive performance across various challenging reasoning tasks in math, coding, STEM, etc.

# Performance improves with more inference-time compute



O3 achieved 87.5% accuracy on ARC-AGI, with >\$1k test-time cost to solve each task

- Other existing LLMs achieved <25% accuracy if no special inference-time techniques were used

# OpenAI o1 demo: hidden thought before generating the solution

User  
query

You are visiting San Francisco for the day and want to meet as many friends as possible. Solve the problem by considering various different schedules and picking the best one to optimize your goals.

Travel distances (in minutes):  
Bayview to Alamo Square: 16.  
Bayview to Union Square: 17.  
Bayview to Golden Gate Park: 22.  
Alamo Square to Bayview: 16.  
Alamo Square to Union Square: 14.  
Alamo Square to Golden Gate Park: 9.  
Union Square to Bayview: 15.  
Union Square to Alamo Square: 15.  
Union Square to Golden Gate Park: 22.  
Golden Gate Park to Bayview: 23.  
Golden Gate Park to Alamo Square: 10.  
Golden Gate Park to Union Square: 22.

CONSTRAINTS: You arrive at Bayview at 9:00AM. Ronald will be at Alamo Square from 8:30AM to 7:45PM. You'd like to meet Ronald for a minimum of 90 minutes. Richard will be at Union Square from 2:30PM to 9:45PM. You'd like to meet Richard for a minimum of 30 minutes. Kenneth will be at Golden Gate Park from 10:00AM to 3:15PM. You'd like to meet Kenneth for a minimum of 60 minutes.



Thought about optimizing meeting schedules for 1m 14s

Below is one clear way to fit in all three visits without wasting time, together with the travel times and the required minimum meeting windows. The key insights are:

Model-generated response

Details ×

*Crafting meeting logistics*

I'm collaborating on a schedule to meet Ronald, Richard, and Kenneth, ensuring each meeting fits within their specified time slots, factoring in travel time and optimal routing.

*Mapping travel logistics*

Evaluating the best route to Golden Gate Park first, considering travel time and waiting for Kenneth. Balancing early arrival against travel efficiency.

*Crafting the itinerary*

Piecing together a travel plan, starting with Kenneth, then Ronald, and ending with Richard. Assessing the optimal sequence and meeting durations.

**Mapping travel routes**

OK, let me see. Traveling from Golden Gate Park to Alamo Square, I'm thinking about extending Ronald's meeting to 1.5 hours then moving to Union Square to meet Richard.

**Evaluating options**

I'm weighing the benefits of a longer Ronald meeting or a later one to avoid waiting for Richard. The goal is to strike a good balance.

**Balancing schedules**

I'm organizing a plan to visit Alamo Square first, then head to Golden Gate Park to meet Kenneth. This sequence maximizes time management and minimizes travel between meetings.

**Streamlining the timeline**

I'm thinking through the optimal approach: meeting Ronald for an uninterrupted 90 minutes to avoid waiting for Richard and making efficient use of the available time before 2:30.

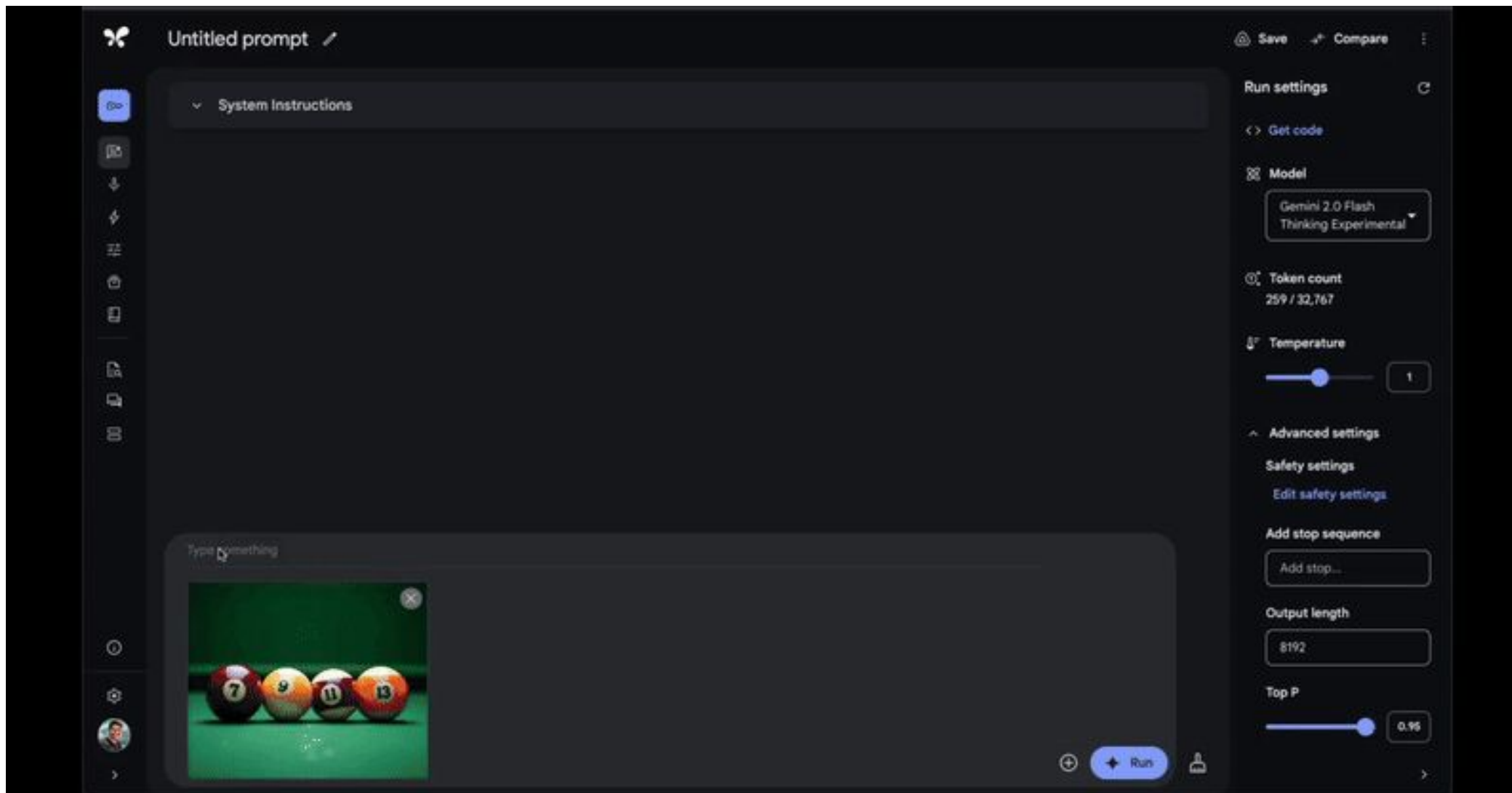
**Tweaking the schedule**

Meeting Ronald for 90 minutes as scheduled, then quickly traveling to Union Square. This reduces waiting time significantly.

**Mapping the timeline**

Summarized  
hidden  
thought

# Gemini 2.0 Flash Thinking demo: generating long thought before the solution



# Core idea: trigger the LLM to generate **long** chain-of-thought (CoT)

## Chain-of-Thought Prompting

### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

### Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had  $23 - 20 = 3$ . They bought 6 more apples, so they have  $3 + 6 = 9$ . The answer is 9. ✓

**Thought**

## Approaches to trigger CoT generation

- Few-shot CoT prompting
- Instruction prompting
- Instruction tuning
- Reinforcement learning

## This lecture: inference-time techniques for scaling token budget

# Outline

- Part 1: Introduction to basic prompting techniques
  - Use **more token budget** to generate **a single solution**
- Part 2: Search and selection from multiple candidates
  - Increase the **width** to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the **depth** to reach the final solution

# Outline

- Part 1: Introduction to basic prompting techniques
  - Use **more token budget** to generate **a single solution**
- Part 2: Search and selection from multiple candidates
  - Increase the width to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the depth to reach the final solution



# Background: standard prompting

## Standard Prompting

### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

Exemplar

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

### Model Output

A: The answer is 27. ❌

- Before the advancement in post-training techniques, standard prompting performance is poor on reasoning benchmarks.
- Issue: standard few-shot exemplars only provide information on the final solution format, but not the **rationale** to derive the solution.

# Chain-of-thought prompting: providing thoughts in the exemplars

## Standard Prompting

### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Exemplar

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

### Model Output

A: The answer is 27. ❌

## Chain-of-Thought Prompting

### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? **Thought**

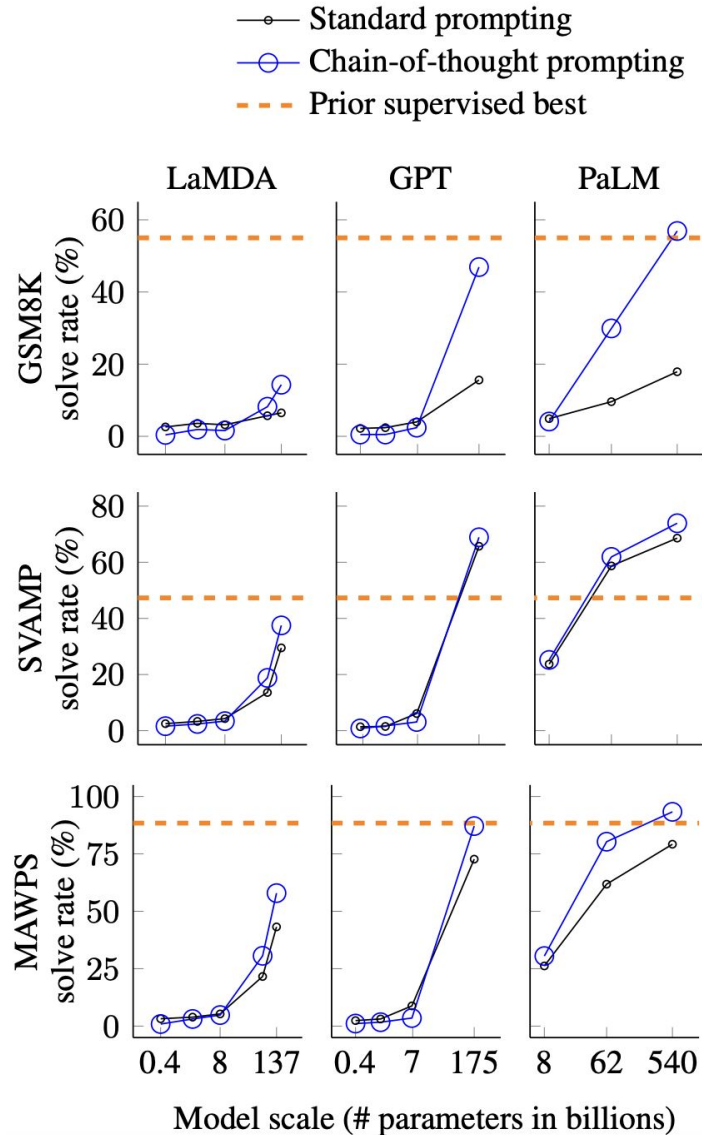
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

### Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had  $23 - 20 = 3$ . They bought 6 more apples, so they have  $3 + 6 = 9$ . The answer is 9. ✅

# CoT performance scales with the model size



- CoT performance improves more significantly with the increase of the model size.
- Better models benefit more with CoT generation
  - A drastic improvement on reasoning performance when the model reaches a certain scale.
- Note: these experiments used pretrained-only LLMs
  - Recent post-trained LLMs might have different scaling curves, but the main conclusions still hold.

# Zero-shot CoT: elicit CoT generation with an instruction

(a) Few-shot

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) The answer is 8. **X**

(b) Few-shot-CoT

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are  $16 / 2 = 8$  golf balls. Half of the golf balls are blue. So there are  $8 / 2 = 4$  blue golf balls. The answer is 4. **✓**

(c) Zero-shot

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: The answer (arabic numerals) is

(Output) 8 **X**

(d) Zero-shot-CoT (Ours)

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: **Let's think step by step.**

(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. **✓**

“Let’s think step by step” triggers CoT generation w/o exemplars.

# Zero-shot CoT significantly outperforms zero-shot performance

	Arithmetic					
	SingleEq	AddSub	MultiArith	GSM8K	AQUA	SVAMP
zero-shot	74.6/ <b>78.7</b>	<b>72.2/77.0</b>	17.7/22.7	10.4/12.5	22.4/22.4	58.8/58.7
zero-shot-cot	<b>78.0/78.7</b>	69.6/74.7	<b>78.7/79.3</b>	<b>40.7/40.5</b>	<b>33.5/31.9</b>	<b>62.1/63.7</b>
	Common Sense		Other Reasoning Tasks		Symbolic Reasoning	
	Common SenseQA	Strategy QA	Date Understand	Shuffled Objects	Last Letter (4 words)	Coin Flip (4 times)
zero-shot	<b>68.8/72.6</b>	12.7/ <b>54.3</b>	49.3/33.6	31.3/29.7	0.2/-	12.8/53.8
zero-shot-cot	64.6/64.0	<b>54.8/52.3</b>	<b>67.5/61.8</b>	<b>52.4/52.9</b>	<b>57.6/-</b>	<b>91.4/87.8</b>

# Issue: zero-shot CoT performance is still worse than few-shot CoT

	MultiArith	GSM8K
<b>Zero-Shot</b>	<b>17.7</b>	<b>10.4</b>
Few-Shot (2 samples)	33.7	15.6
Few-Shot (8 samples)	33.8	15.6
<b>Zero-Shot-CoT</b>	<b>78.7</b>	<b>40.7</b>
Few-Shot-CoT (2 samples)	84.8	41.3
Few-Shot-CoT (4 samples : First) (*1)	89.2	-
Few-Shot-CoT (4 samples : Second) (*1)	90.5	-
Few-Shot-CoT (8 samples)	93.0	48.7
<b>Zero-Plus-Few-Shot-CoT (8 samples) (*2)</b>	<b>92.8</b>	<b>51.5</b>

How to improve CoT performance w/o manually labeling exemplars?

# Analogical prompting: instruct the LLM to generate exemplars

Prompt the LLM to **first recall relevant exemplars**, before solving the test problem.

## Model Input

**Q:** What is the area of the square with the four vertices at (-2, 2), (2, -2), (-2, -6), and (-6, -2)?

**# Instruction:**

**## Recall relevant exemplars:**

**## Solve the initial problem:**

## Model Output

**# Relevant exemplars:**

**Q:** What is the area of the square with a side length of 5?

**A:** The area of a square is found by squaring the length of its side.

So, the area of this square is  $5^2 = 25$ . ...

**# Solve the initial problem:**

To find the area of the square, we need to find the side length.

The length is ...  $\sqrt{(2 - (-2))^2 + (-2 - 2)^2} = \sqrt{32}$ . So, the area of the square is  $(\sqrt{32})^2 = 32$ .

# Analogical prompting: instruct the LLM to generate exemplars

## Benefits

- Exemplars are **self-generated by LLMs**, no manual labeling.
- Exemplars are **tailored** to individual problems.

### Model Input

**Q:** What is the area of the square with the four vertices at (-2, 2), (2, -2), (-2, -6), and (-6, -2)?

**# Instruction:**

**## Recall relevant exemplars:**

**## Solve the initial problem:**

### Model Output

**# Relevant exemplars:**

**Q:** What is the area of the square with a side length of 5?

**A:** The area of a square is found by squaring the length of its side.

So, the area of this square is  $5^2 = 25$ . ...

**# Solve the initial problem:**

To find the area of the square, we need to find the side length.

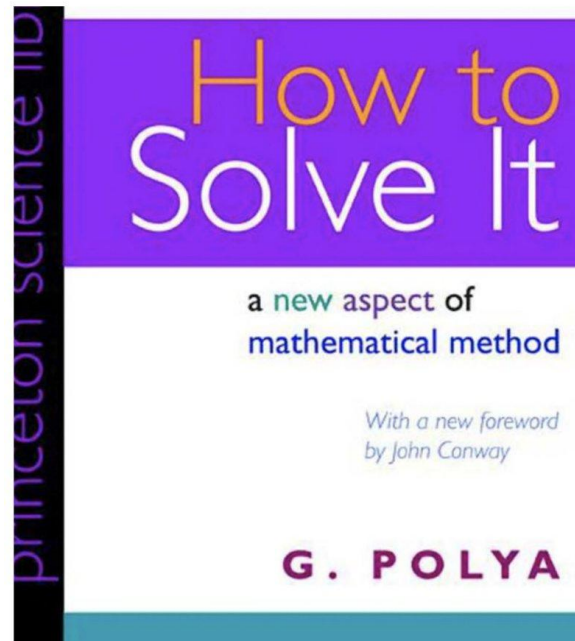
The length is ...  $\sqrt{(2 - (-2))^2 + (-2 - 2)^2} = \sqrt{32}$ . So, the

area of the square is  $(\sqrt{32})^2 = 32$ .



# Motivation: human analogical reasoning

- Humans are not explicitly given demonstrations every time for a new task
- Instead, humans intrinsically recall from past relevant experience



**Do you know a related problem?** We can scarcely imagine a problem absolutely new, unlike and unrelated to any formerly solved problem; but, if such a problem could exist, it would be insoluble. In fact, when solving a problem, we always profit from previously solved problems, using their result, or their method, or the experience we acquired solving them. And, of course, the problems from which we profit must be in some way related to our present problem. Hence the question: *Do you know a related problem?*

There is usually no difficulty at all in recalling formerly solved problems which are more or less related to our present one. On the contrary, we may find too many such problems and there may be difficulty in choosing a useful one. We have to look around for closely related problems; we look at the unknown, or we look for a formerly solved problem which is linked to our present one by GENERALIZATION, SPECIALIZATION, OR ANALOGY.

# Example: Codeforces (prompt)

## Our Prompt: Self-generate Exemplars + Knowledge

Your goal is to write Python3 code to solve competitive programming problems. Given a problem, explain the core concepts in it and provide other relevant problems. Then solve the original problem.

**# Problem:**  
You are given a sequence  $a_1, a_2, \dots, a_n$ . Each element of  $a$  is 1 or 2. Find out if an integer  $k$  exists so that the following conditions are met.  $1 \leq k \leq n-1$ , and  $a_1 \cdot a_2 \dots a_k = a_{k+1} \dots a_n$ . If there exist multiple  $k$  that satisfy the given condition, print the smallest.

**# Instruction:**

**## Algorithms:**

Identify the core concepts or algorithms used to solve the problem.

**## Tutorial:**

Write a tutorial about these algorithms.

**## Example Problems:**

Provide three examples of relevant competitive programming problems that involve these algorithms. For each problem, describe the problem, explain the solution in detail, and then write the correct Python3 code.

**## Python3 code to solve the original problem:**

- Explanation of the solution:
- Python3 code to solve the problem:

- Besides exemplars, the LLM can also self-generate high-level knowledge
- The generated knowledge complements the problems with broader insights

# Example: Codeforces (self-generated knowledge + exemplars)

Output by GPT3.5-turbo

## Self-generated Knowledge

**## Algorithms:** The core algorithm for solving this problem is the prefix product algorithm.

**## Tutorial:** The prefix product algorithm is used to calculate the product of all the elements in a prefix of an array. The algorithm works by initializing a variable "prefix" to 1, and then iterating through the array from left to right. At each iteration, the current element is multiplied with the prefix, and the result is stored in the prefix variable. This way, the prefix variable keeps track of the product of all the elements in the prefix of the array. The time complexity of this algorithm is  $O(n)$ , where  $n$  is the size of the input array. ....

## Self-generated Exemplars

**## Example Problems:**

1. Problem: Given an array of integers, find the product of all the elements in each subarray of size  $k$ .

Solution: We can use the prefix product algorithm to solve this problem. We initialize a prefix variable to 1, and then iterate through the array from left to right. At each iteration, we multiply the current element with the prefix, .....

```
def product_of_subarrays(arr, k):
    prefix = 1
    result = []
    for i in range(len(arr)):
        prefix = prefix * arr[i]
        if i >= k-1:
            result.append(prefix)
            prefix = prefix // arr[i-k+1]
    return result
```

2. Problem: Given an array of integers, find the longest subarray with a product less than or equal to a given target.

Solution: .....

3. Problem: Given an array of integers, find the smallest  $k$  such that the product of all the elements in the prefix of the array is equal to the product of all the elements in the suffix of the array.

Solution: .....

# Example: Codeforces (solving the initial problem)

**## Python3 code to solve the original problem:**

Explanation of the solution:

- Initialize prefix and suffix variables to 1, and k to -1.
- Calculate the product of all the elements in the sequence using the prefix product algorithm.
- Iterate through the sequence from right to left.
  - Update the prefix and suffix variables.
  - If the prefix is equal to the suffix, update k to the current index.
- Print the value of k.

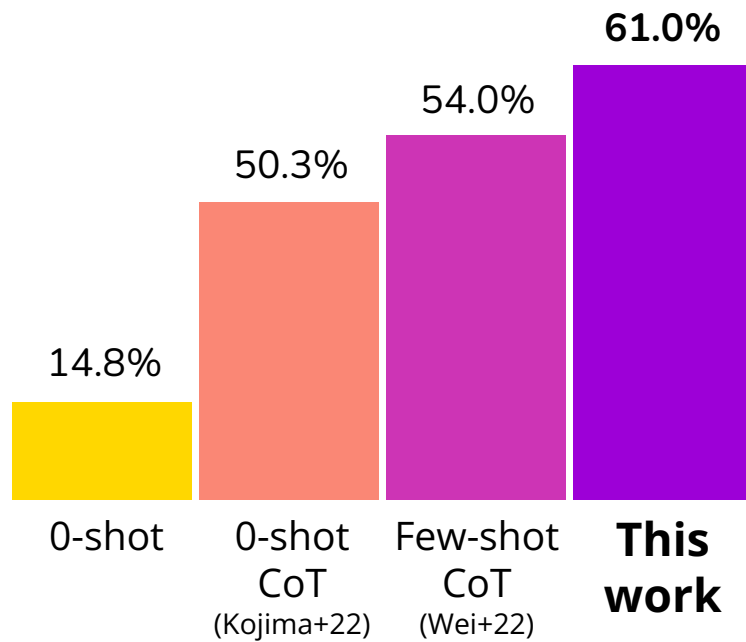
```
t = int(input())
for _ in range(t):
    n = int(input())
    arr = list(map(int, input().split()))
    prefix, suffix = 1, 1
    k = -1
    for i in range(n):
        prefix = prefix * arr[i]
    for i in range(n-1, 0, -1):
        prefix = prefix // arr[i]
        suffix = suffix * arr[i]
        if prefix == suffix:
            k = i
    print(k)
```

# Result overview

- Analogical prompting outperforms 0-shot CoT and manual few-shot CoT

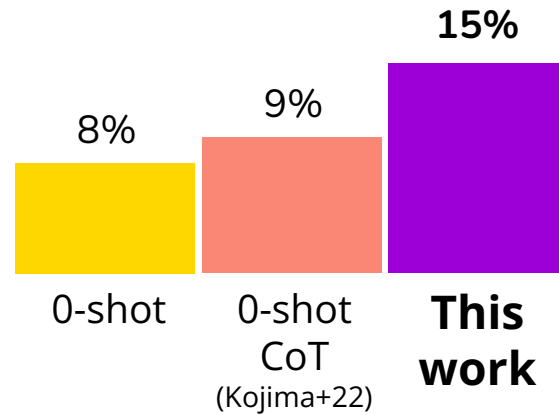
## Math problems

(GSM8K, text-davinci-003)



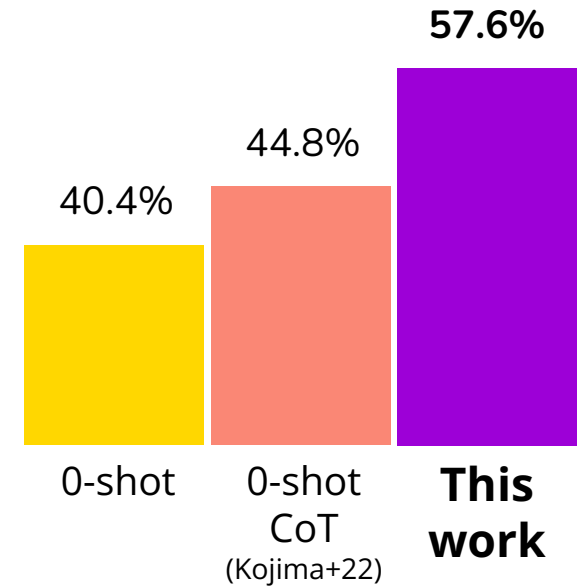
## Code generation

(Codeforces, GPT3.5 turbo)



## Temporal reasoning

(BIG-Bench, GPT3.5 turbo)



# Stronger LLMs are better analogical reasoners

Prompting Method	( $\leftarrow$ scale down)		(scale up $\rightarrow$ )		
	text-curie-001	text-davinci-001	text-davinci-002	text-davinci-003	
0-shot	2%	6%	13%	14%	
0-shot CoT	2%	6%	22%	50%	
5-shot (fixed) CoT	2%	10%	43%	54%	
5-shot retrieved CoT	3%	11%	47%	57%	
<b>Ours: Self-generated Exemplars</b>	2%	9%	<b>48%</b>	<b>61%</b>	

GSM8K for math reasoning

- Weaker LLMs benefit less from analogical prompting, though it does not hurt the zero-shot performance
- With stronger LLMs, analogical prompting outperforms CoT with manually-designed or retrieved exemplars
  - The generated CoT is more tailored to the underlying LLM

# Which instructions work for CoT generation?

No.	Category	Template	Accuracy
1	instructive	Let's think step by step.	<b>78.7</b>
2		First, <input type="text"/>	77.3
3		Let's think about this logically.	74.5
4		Let's solve this problem by splitting it into steps. <input type="text"/>	72.2
5		Let's be realistic and think step by step.	70.8
6		Let's think like a detective step by step.	70.3
7		Let's think	57.5
8		Before we dive into the answer,	55.7
9		The answer is after the proof.	45.7
10	misleading	Don't think. Just feel.	18.8
11		Let's think step by step but reach an incorrect answer.	18.7
12		Let's count the number of "a" in the question.	16.7
13		By using the fact that the earth is round,	9.3
14	irrelevant	By the way, I found a good restaurant nearby.	17.5
15		AbraKadabra!	15.5
16		It's a beautiful day.	13.1
-		(Zero-shot)	17.7

Zero-shot CoT on MultiArith with text-davinci-002

- Current LLMs are sensitive to prompt design
- There is no clear principle of how to write optimal prompts
- How to reduce the manual work for writing prompts?

# Large language models for prompt engineering

✓ Keep the high score candidates    
 ✗ Discard the low score candidates    
 ★ Final selected prompt with highest score

**LLMs as Inference Models**

Professor Smith was given the following instructions: <INSERT>

Here are the Professor's responses:

# Demostration Start  
**Input:** prove    **Output:** disprove  
**Input:** on       **Output:** off  
 ...  
 # Demostration End

[Optional]

**LLMs as Resampling Models**

Generate a variation of the following instruction while keeping the semantic meaning.

**Input:** write the antonym of the word.  
**Output:** <COMPLETE>

**LLMs as Scoring Models**

**Instruction:** write the antonym of the word.    <LIKELIHOOD>

**Input:** direct    **Output:** indirect

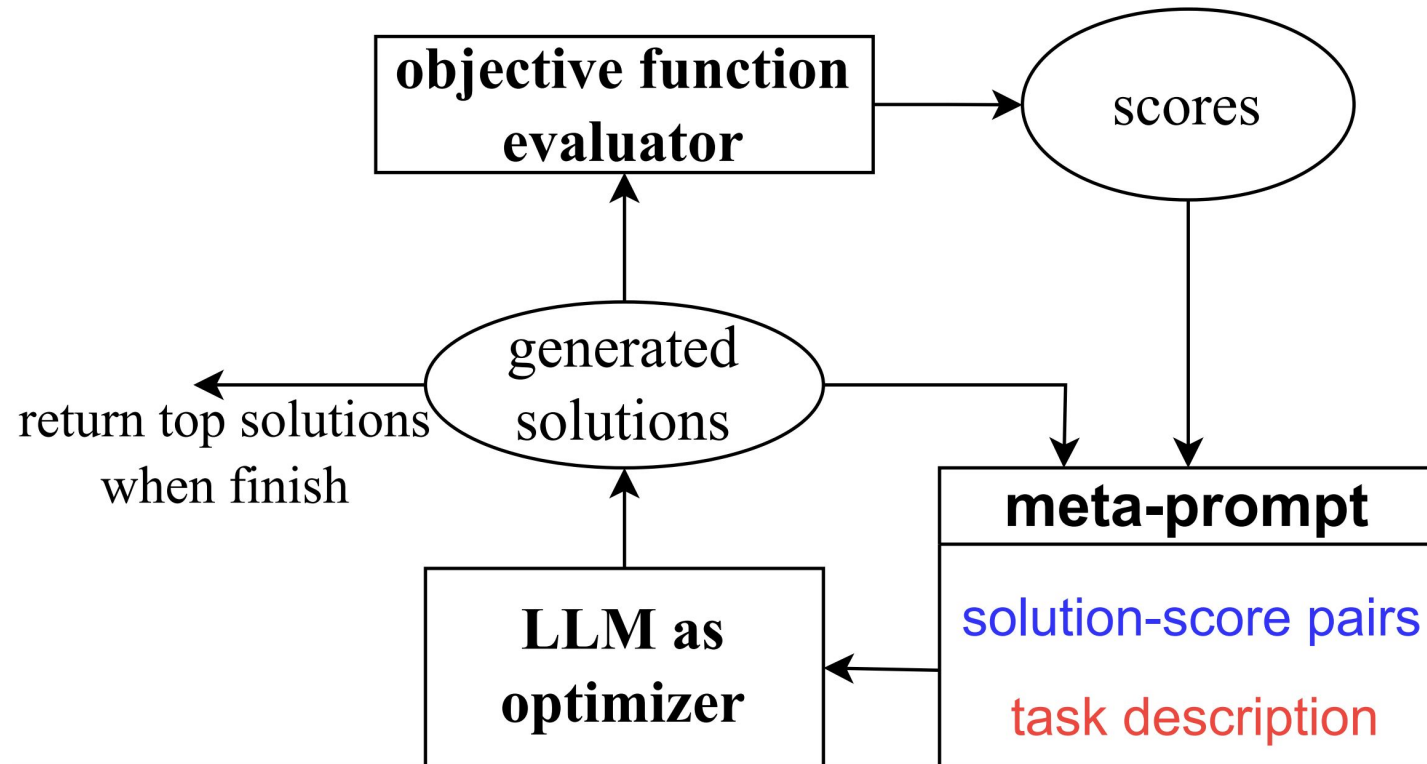
① Proposal			
	➔		
		② Scoring ↑	
			③ Log Probability ↓
④ High Score Candidates	←		
⑤ Similar Candidates	➔		

- Proposal generation: leverage the LLM to generate initial instructions.
- Scoring each instruction based on the prediction correctness on a small set of problems.



# Going further: LLM as the optimizer to iteratively improve the prompt

- Core idea: instruct the LLM to **leverage the past optimization trajectory**, represented as sorted (solution, score) pairs
- **Optimizer**: the LLM to propose a new instruction given old ones and task exemplars
- **Evaluator**: the LLM to evaluate the accuracy of an instruction



# Meta-prompt for prompt optimization

## Example meta-prompt for GSM8K:

I have some texts along with their corresponding scores. The texts are arranged in ascending order based on their scores, where higher scores indicate better quality.

```
text:  
Let's figure it out!  
score:  
61  
text:  
Let's solve the problem.  
score:  
63  
(. . . more instructions and scores . . . )
```

(top) past instructions and accuracies

The following exemplars show how to apply your text: you replace <INS> in each input with your text, then read the input and give an output. We say your output is wrong if your output is different from the given output, and we say your output is correct if they are the same.

```
input:  
Q: Alannah, Beatrix, and Queen are preparing for the new school year and have been given books by their parents. Alannah has 20 more books than Beatrix. Queen has 1/5 times more books than Alannah. If Beatrix has 30 books, how many books do the three have together?  
A: <INS>  
output:  
140  
(. . . more exemplars . . . )
```

exemplars

Write your new text that is different from the old ones and has a score as high as possible. Write the text in square brackets.

# Results on GSM8K

Table 1: Top instructions with the highest GSM8K zero-shot test accuracies from prompt optimization with different optimizer LLMs. All results use the pre-trained PaLM 2-L as the scorer.

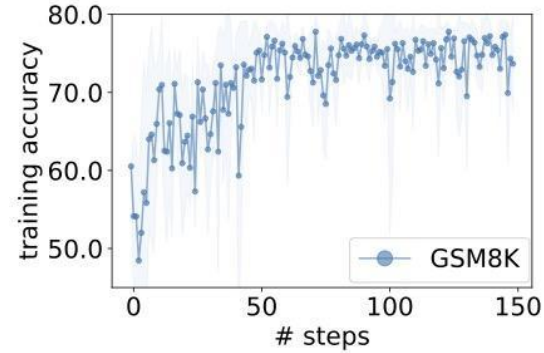
Source	Instruction	Acc
<i>Baselines</i>		
(Kojima et al., 2022)	Let’s think step by step.	71.8
(Zhou et al., 2022b)	Let’s work this out in a step by step way to be sure we have the right answer.	58.8
	(empty string)	34.0
<i>Ours</i>		
PaLM 2-L-IT	Take a deep breath and work on this problem step-by-step.	<b>80.2</b>
PaLM 2-L	Break this down.	79.9
gpt-3.5-turbo	A little bit of arithmetic and a logical approach will help us quickly arrive at the solution to this problem.	78.5
gpt-4	Let’s combine our numerical command and clear thinking to quickly and accurately decipher the answer.	74.5

- Initial instruction for prompt optimization: “Let’s solve the problem.” with acc = 60.8%.
- Our best LLM-generated prompt **outperforms “Let’s think step by step” by ~8%**, matching the few-shot CoT accuracy (80.7%) in PaLM-2 technical report.

# Optimization graphs

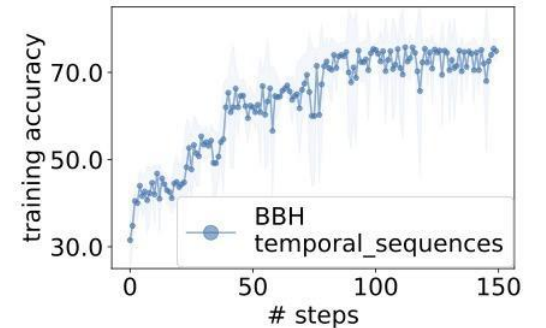
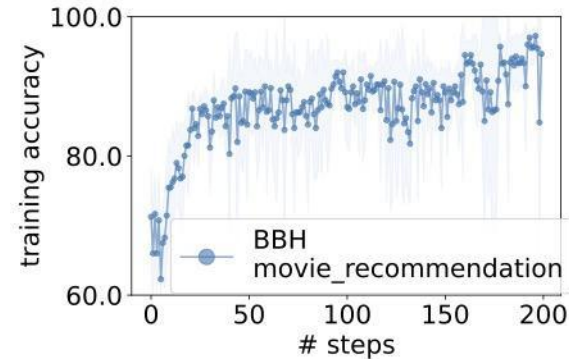
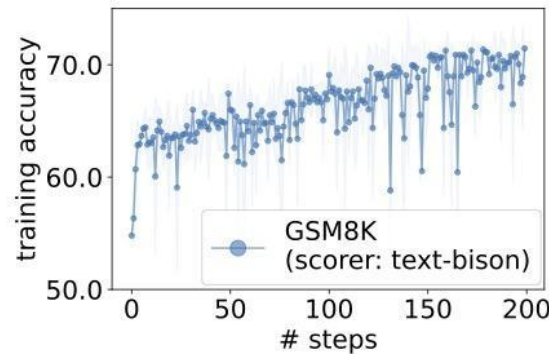
scorer: PaLM 2-L

optimizer: PaLM 2-L-IT



scorer: text-bison

optimizer: PaLM 2-L-IT

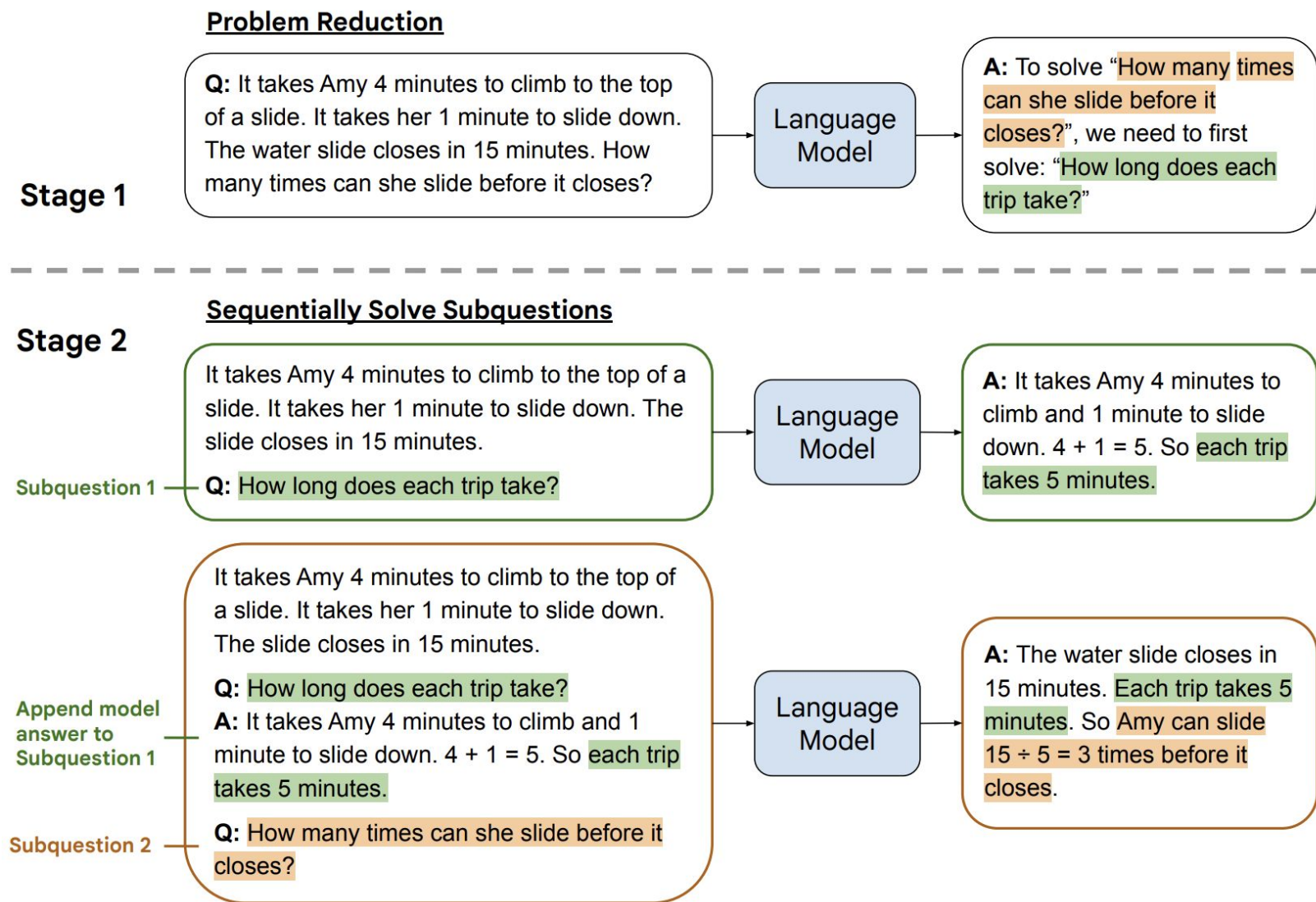


The accuracy increases with more optimization steps, then plateaus.

# Going back to CoT: what CoT brings into LLM reasoning

- Chain-of-thought prompting: **variable computation** of the thought process adapting to tasks of different difficulty levels
  - More complex questions -> more reasoning steps in the chain-of-thought
- Reasoning strategies enabled by CoT
  - Decomposition
  - Planning
  - ...
- We can explicitly instruct the LLM with the desired reasoning strategies for problem solving

# Least-to-most prompting: easy-to-hard generalization via decomposition



## Example: solving the SCAN compositional generalization benchmark

Command	Action Sequence
“look thrice after jump”	JUMP LOOK LOOK LOOK
“run left and walk”	TURN_LEFT RUN WALK
“look opposite right”	TURN_RIGHT TURN_RIGHT LOOK

Method	Standard prompting	Chain-of-Thought	Least-to-Most
code-davinci-002	16.7	16.2	<b>99.7</b>
text-davinci-002	6.0	0.0	<b>76.0</b>
code-davinci-001	0.4	0.0	<b>60.7</b>

### SCAN length split

- SCAN: translating synthetic natural language commands into actions sequences
- Length split: action sequences in the test set are longer than training samples
- Least-to-most prompting can achieve nearly perfect test accuracy with 0.1% training samples as exemplars

# Example: solving text-to-code compositional generalization benchmarks

## Examples in CFQ (Compositional Freebase Questions) benchmark

---

Question: Did M1 star M2 , star M3 , and star a art director and editor of M0?

SPARQL: SELECT count(\*) WHERE

{ ?x0 edited M0 . ?x0 art directed M0 . M1 starred ?x0 . M1 starred M2 . M1 starred M3 }

Question: What was produced by a art director that M1 and M2 employed?

SPARQL: SELECT DISTINCT WHERE

{ ?x0 produced by ?x1 . ?x1 a art director . M0 employed ?x1 . M1 employed ?x1 }

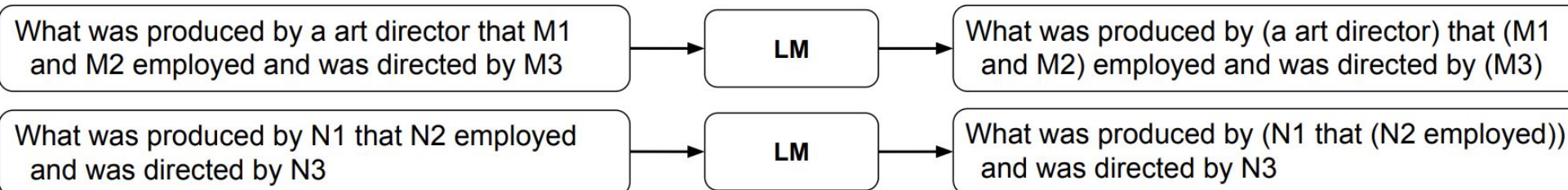
---

- Challenge: more complicated grammar with a larger vocabulary
  - A single prompt might not be enough to cover all grammar rules
- Decomposition enables customized prompt for each subproblem

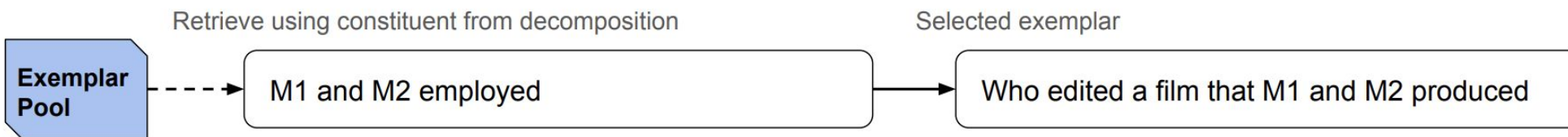


# Our approach: dynamic least-to-most prompting

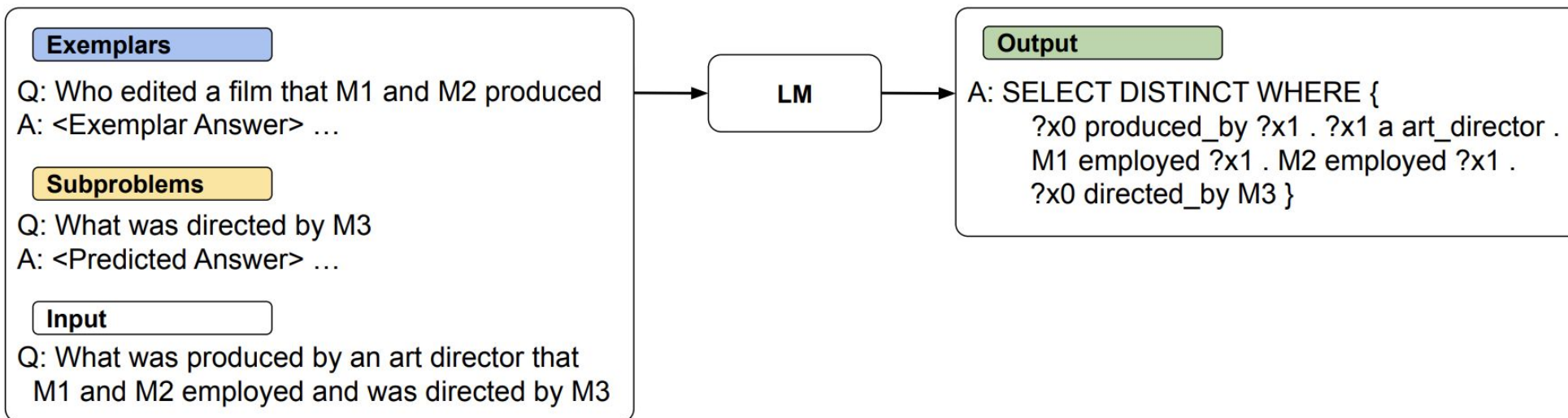
## Problem Reduction (Syntactic Parsing)



## Dynamically Select Exemplars for Each Subproblem

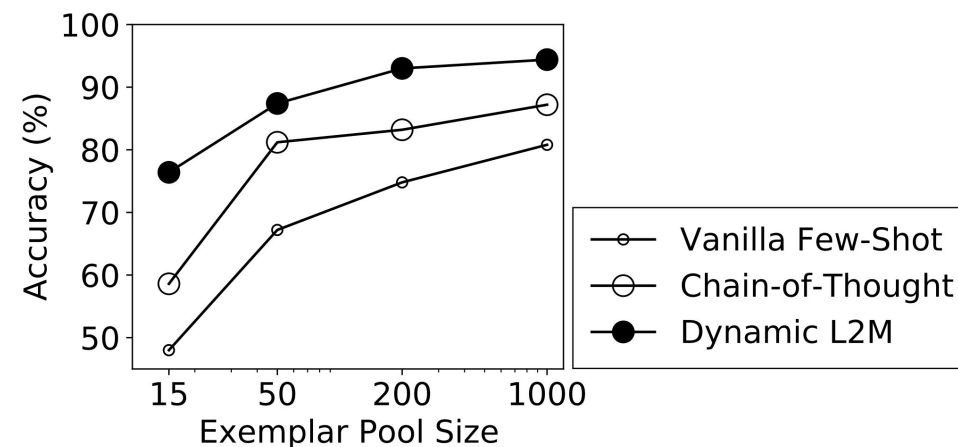


## Sequentially Solve Subproblems

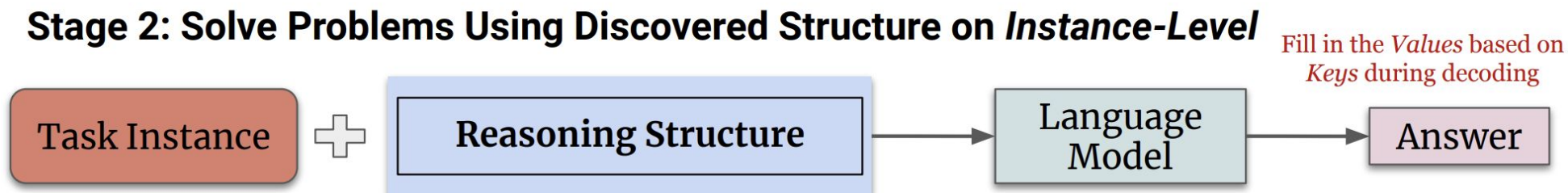
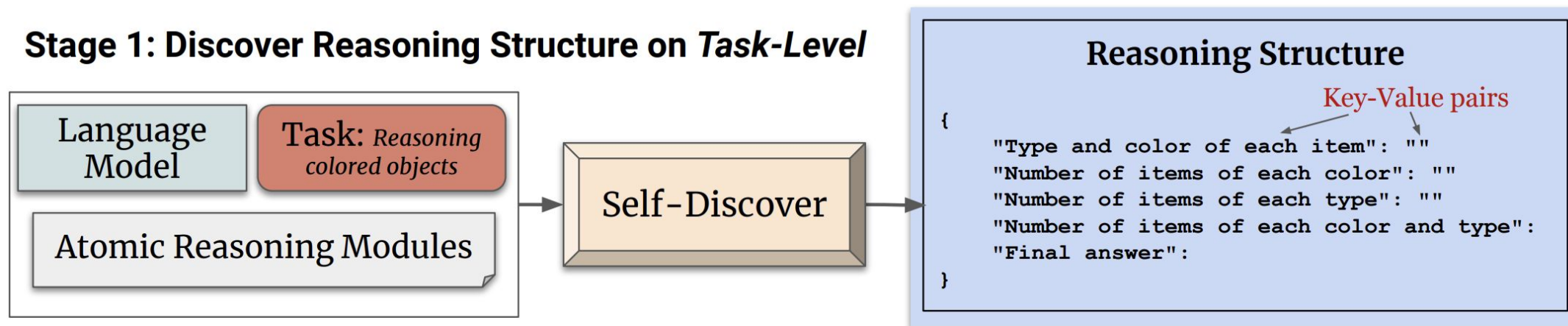


# Evaluation on CFQ

	MCD1	MCD2	MCD3	Ave.
<b>Fully Supervised</b>				
T5-base (Herzig et al., 2021)	58.5	27.0	18.4	34.6
T5-large (Herzig et al., 2021)	65.1	32.3	25.4	40.9
T5-3B (Herzig et al., 2021)	65.0	41.0	42.6	49.5
HPD (Guo et al., 2020)	79.6	59.6	67.8	69.0
T5-base + IR (Herzig et al., 2021)	85.8	64.0	53.6	67.8
T5-large + IR (Herzig et al., 2021)	88.6	79.2	72.7	80.2
T5-3B + IR (Herzig et al., 2021)	88.4	85.3	77.9	83.9
LeAR (Liu et al., 2021)	91.7	89.2	91.7	90.9
<b>Prompting</b>				
(Ours) Dynamic Least-to-Most	<b>94.3</b>	<b>95.3</b>	<b>95.5</b>	<b>95.0</b>

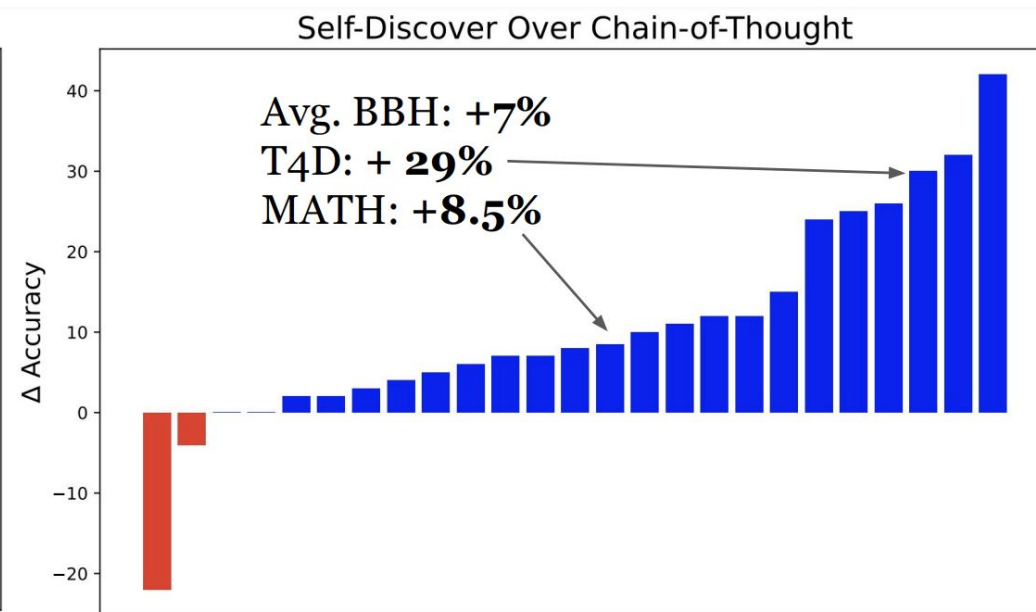
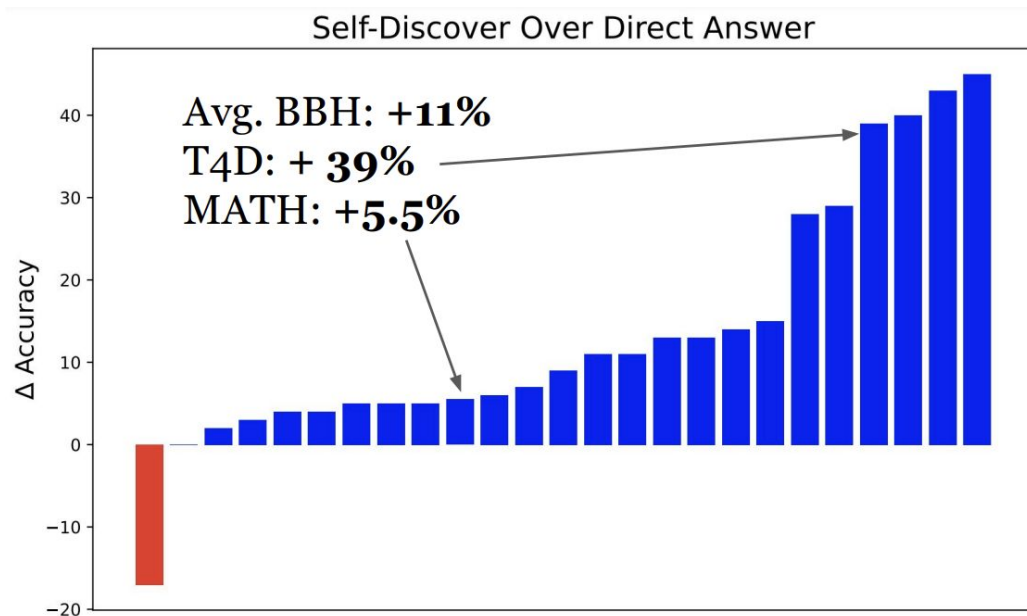


# Self-Discover: instruct the LLM to compose reasoning structures for each task



- Different reasoning tasks require different reasoning structures, i.e., different ways to decompose the task and plan for each stage.
- Self-Discover composes task-specific reasoning structures without manually-written demonstrations.

# Self-Discover: instruct the LLM to compose reasoning structures for each task



# Summary

- Chain-of-thought generation: **variable computation** of the thought process adapting to tasks of different difficulty levels
- How to improve the CoT performance at inference time
  - Few-shot prompting with labeling of thoughts
  - Instruction prompting to trigger CoT generation
  - Instruct the LLM to automate the prompt design
- Note: the best practice to interact with LLMs evolves over time
  - The principles of how to discover good prompting strategies for reasoning hold true
    - Encourage longer CoT for complex tasks
    - Support reasoning strategies required for the task

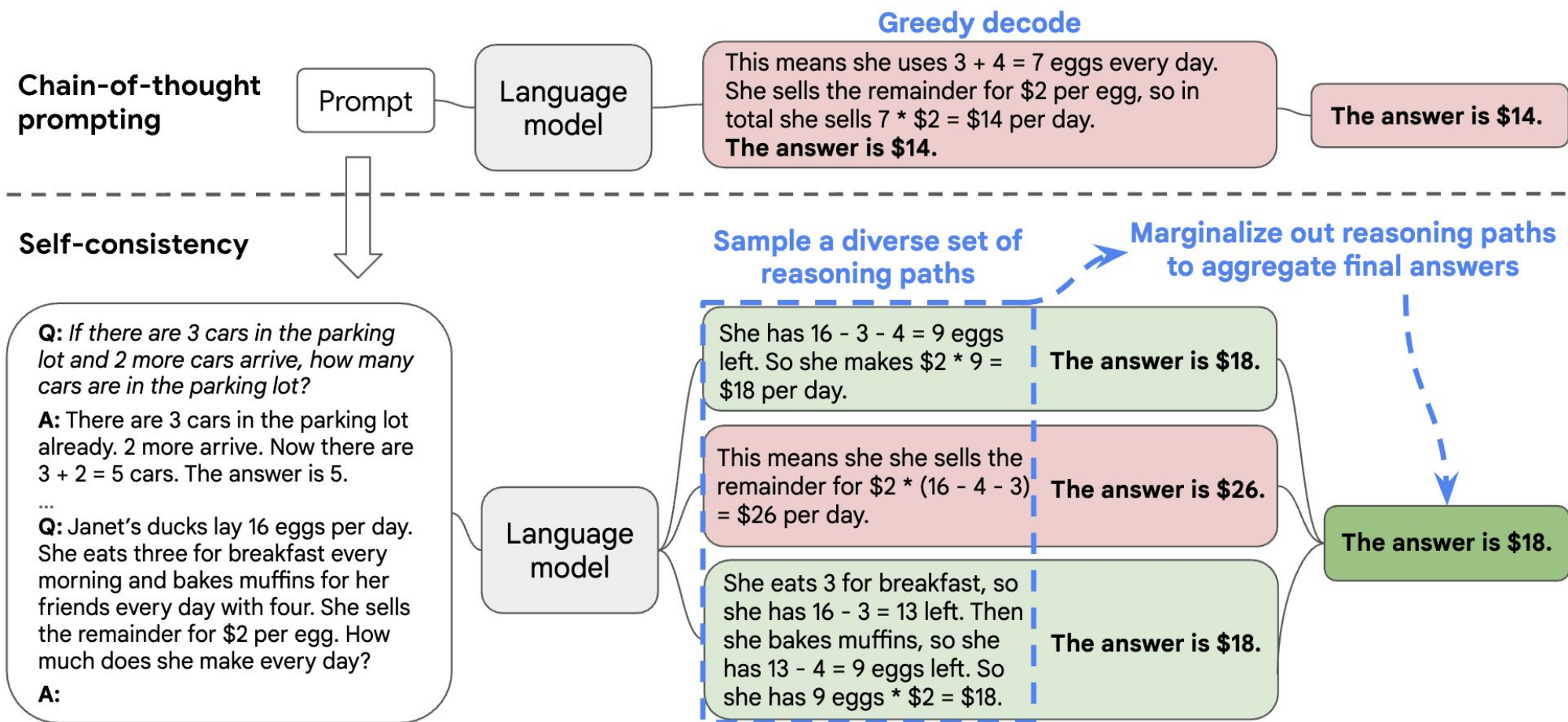
# Outline

- Part 1: Introduction to basic prompting techniques
  - Use more token budget to generate a single solution
- Part 2: Search and selection from multiple candidates
  - Increase the **width** to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the depth to reach the final solution

# What is missing so far?

- We should not limit the LLM to generate only one solution per problem!
- Exploring multiple branches allows the LLM to recover from mistakes in a single generation
  - Generate multiple candidate solutions per problem
  - Generate multiple potential next reasoning steps given the current (partial) thought
- Challenge: how to select the best response from multiple candidates?
  - In most cases, we do not have an oracle scorer at inference time

# Self-Consistency: select the response with the most consistent final answer



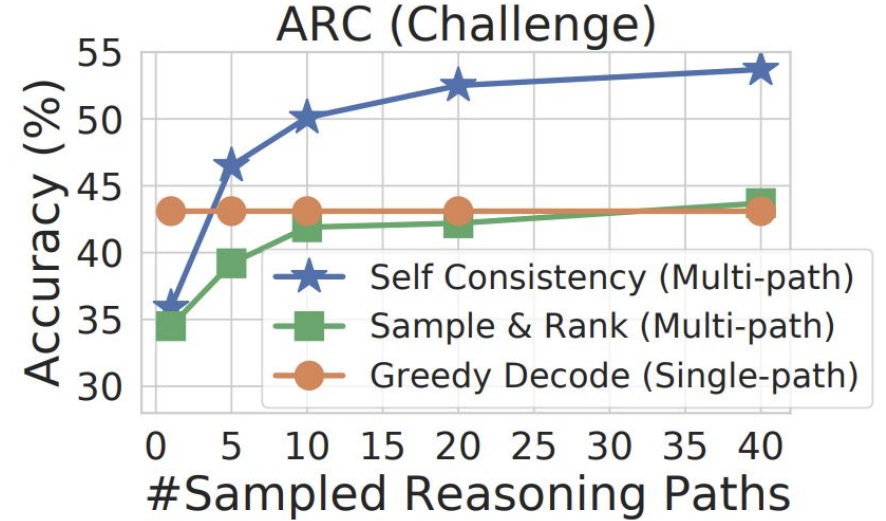
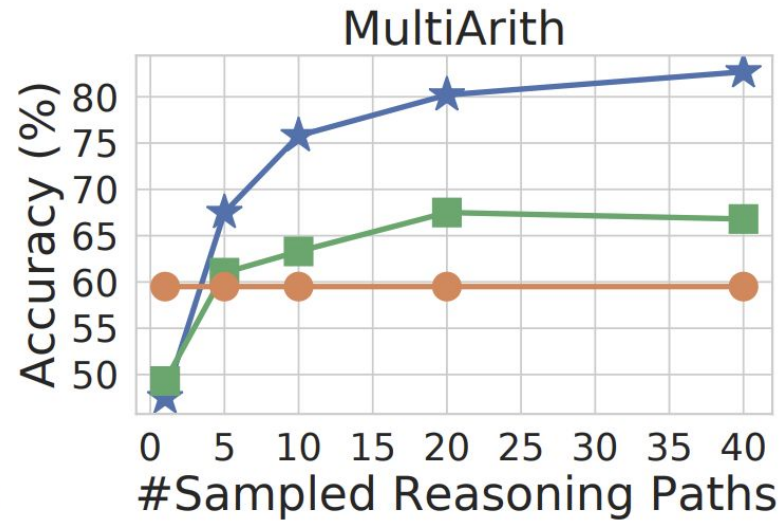
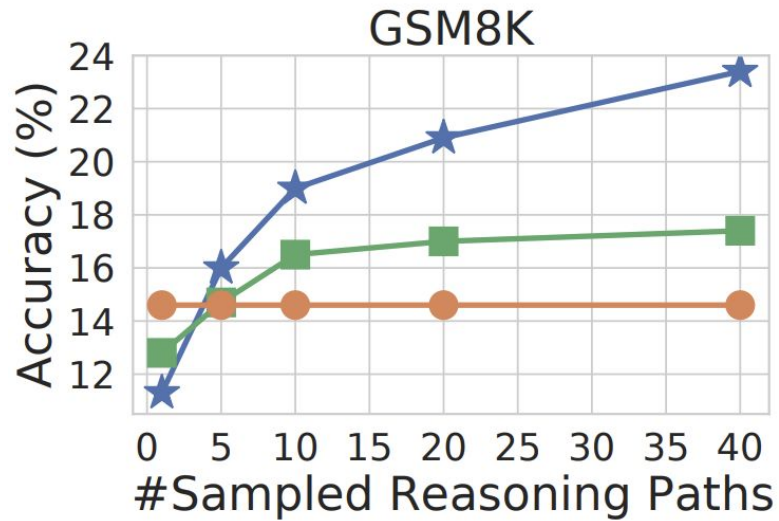
Note: the selection is only based on the final answer, the reasoning paths do not need to be the same across different sampled responses.



# Self-Consistency boosts the performance across models and benchmarks

	Method	AddSub	MultiArith	ASDiv	AQuA	SVAMP	GSM8K
	Previous SoTA	<b>94.9<sup>a</sup></b>	60.5 <sup>a</sup>	75.3 <sup>b</sup>	37.9 <sup>c</sup>	57.4 <sup>d</sup>	35 <sup>e</sup> / 55 <sup>g</sup>
UL2-20B	CoT-prompting	18.2	10.7	16.9	23.6	12.6	4.1
	Self-consistency	24.8 (+6.6)	15.0 (+4.3)	21.5 (+4.6)	26.9 (+3.3)	19.4 (+6.8)	7.3 (+3.2)
LaMDA-137B	CoT-prompting	52.9	51.8	49.0	17.7	38.9	17.1
	Self-consistency	63.5 (+10.6)	75.7 (+23.9)	58.2 (+9.2)	26.8 (+9.1)	53.3 (+14.4)	27.7 (+10.6)
PaLM-540B	CoT-prompting	91.9	94.7	74.0	35.8	79.0	56.5
	Self-consistency	93.7 (+1.8)	99.3 (+4.6)	81.9 (+7.9)	48.3 (+12.5)	86.6 (+7.6)	74.4 (+17.9)
GPT-3 Code-davinci-001	CoT-prompting	57.2	59.5	52.7	18.9	39.8	14.6
	Self-consistency	67.8 (+10.6)	82.7 (+23.2)	61.9 (+9.2)	25.6 (+6.7)	54.5 (+14.7)	23.4 (+8.8)
GPT-3 Code-davinci-002	CoT-prompting	89.4	96.2	80.1	39.8	75.8	60.1
	Self-consistency	91.6 (+2.2)	<b>100.0</b> (+3.8)	<b>87.8</b> (+7.6)	<b>52.0</b> (+12.2)	<b>86.8</b> (+11.0)	<b>78.0</b> (+17.9)

# Self-Consistency performance scales with more samples



- Sample-and-Rank baseline: select the response with the highest log probability
- Self-Consistency performance scales much better than probability-based ranking
  - Unless the model is trained to be a good verifier (will cover later)

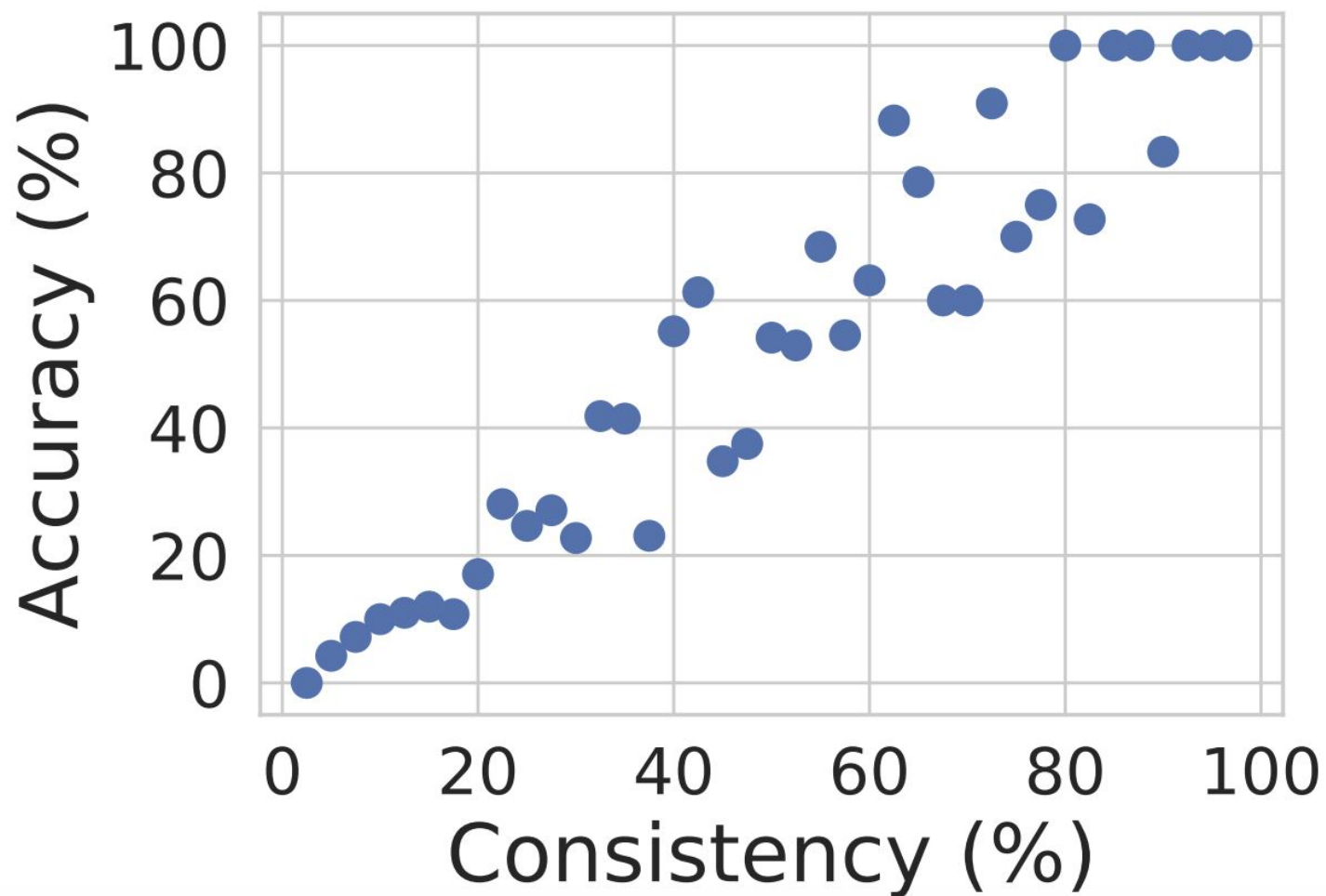
# Sampling **diverse** responses is crucial to self-consistency

	GSM8K	MultiArith	SVAMP	ARC-e	ARC-c
CoT (Wei et al., 2022)	17.1	51.8	38.9	75.3	55.1
Ensemble (3 sets of prompts)	18.6 $\pm$ 0.5	57.1 $\pm$ 0.7	42.1 $\pm$ 0.6	76.6 $\pm$ 0.1	57.0 $\pm$ 0.2
Ensemble (40 prompt permutations)	19.2 $\pm$ 0.1	60.9 $\pm$ 0.2	42.7 $\pm$ 0.1	76.9 $\pm$ 0.1	57.0 $\pm$ 0.1
Self-Consistency (40 sampled paths)	<b>27.7 <math>\pm</math> 0.2</b>	<b>75.7 <math>\pm</math> 0.3</b>	<b>53.3 <math>\pm</math> 0.2</b>	<b>79.3 <math>\pm</math> 0.3</b>	<b>59.8 <math>\pm</math> 0.2</b>

	Beam size / Self-consistency paths	1	5	10	20	40
AQuA	Beam search decoding (top beam)	23.6	19.3	16.1	15.0	10.2
	Self-consistency using beam search	23.6	19.8 $\pm$ 0.3	21.2 $\pm$ 0.7	24.6 $\pm$ 0.4	24.2 $\pm$ 0.5
	Self-consistency using sampling	19.7 $\pm$ 2.5	<b>24.9 <math>\pm</math> 2.6</b>	<b>25.3 <math>\pm</math> 1.8</b>	<b>26.7 <math>\pm</math> 1.0</b>	<b>26.9 <math>\pm</math> 0.5</b>
MultiArith	Beam search decoding (top beam)	10.7	12.0	11.3	11.0	10.5
	Self-consistency using beam search	10.7	11.8 $\pm$ 0.0	11.4 $\pm$ 0.1	12.3 $\pm$ 0.1	10.8 $\pm$ 0.1
	Self-consistency using sampling	9.5 $\pm$ 1.2	11.3 $\pm$ 1.2	<b>12.3 <math>\pm</math> 0.8</b>	<b>13.7 <math>\pm</math> 0.9</b>	<b>14.7 <math>\pm</math> 0.3</b>

- Beam search: keep top k paths with the highest probabilities in the decoding process
- Ensemble baselines: apply greedy decoding for all prompt variants of a problem
- Self-Consistency using sampling scales with more samples
  - The sampling method needs to ensure the response diversity, e.g., using a high temperature, nucleus sampling, etc.

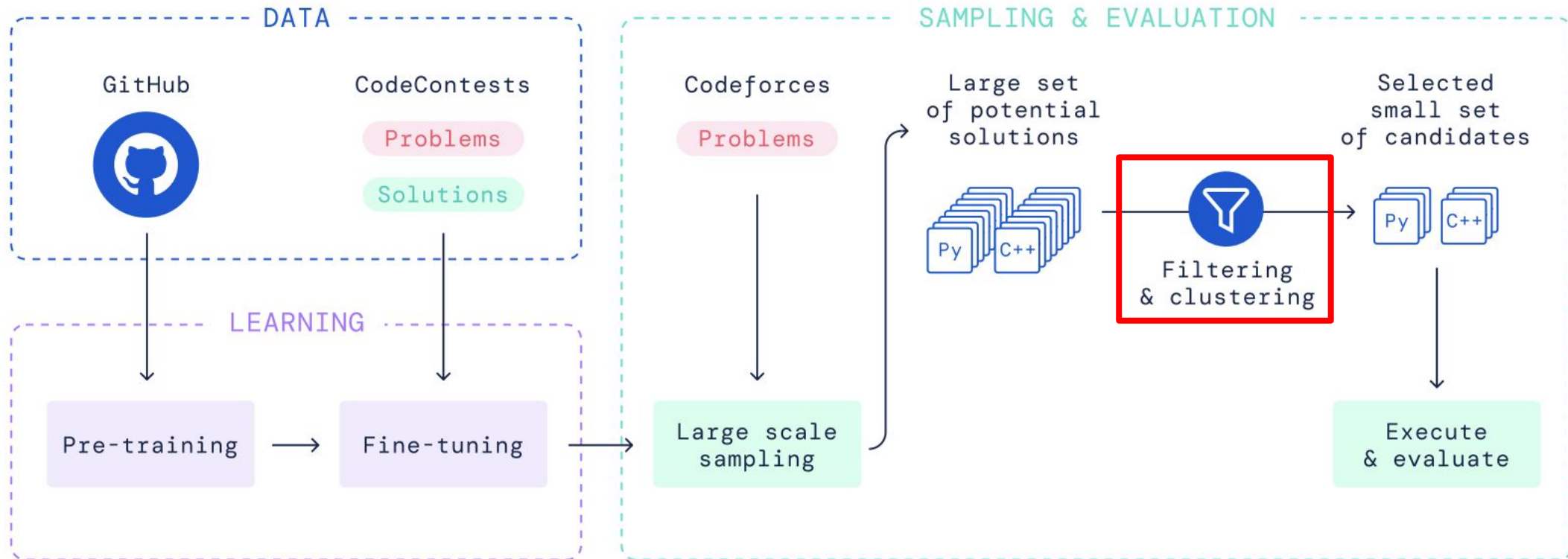
## Consistency is highly correlated with the accuracy



If more sampled responses lead to the same final answer:

- The LLM is more certain with its predicted conclusion
- The aggregated solution is more likely to be correct

# Consistency-based code selection in AlphaCode



Clustering predicted code based on the consistency on execution results

# Problem setting: competitive programming

## D.Backspace

You are given two strings  $s$  and  $t$ , both consisting of lowercase English letters. You are going to type the string  $s$  character by character, from the first character to the last one.

When typing a character, instead of pressing the button corresponding to it, you can press the "Backspace" button. It deletes the last character you have typed among those that aren't deleted yet (or does nothing if there are no characters in the current string). For example, if  $s$  is "abcba" and you press Backspace instead of typing the first and the fourth characters, you will get the string "ba" (the first press of Backspace deletes no character, and the second press deletes the character 'c'). Another example, if  $s$  is "abcaa" and you press Backspace instead of the last two letters, then the resulting text is "a".

Your task is to determine whether you can obtain the string  $t$ , if you type the string  $s$  and press "Backspace" instead of typing several (maybe zero) characters of  $s$ .

### Input

The first line contains a single integer  $q$  ( $1 \leq q \leq 10^5$ ) — the number of test cases.

The first line of each test case contains the string  $s$  ( $1 \leq |s| \leq 10^5$ ). Each character of  $s$  is a lowercase English letter.

The second line of each test case contains the string  $t$  ( $1 \leq |t| \leq 10^5$ ). Each character of  $t$  is a lowercase English letter.

It is guaranteed that the total number of characters in the strings over all test cases does not exceed  $2 \cdot 10^5$ .

### Output

For each test case, print "YES" if you can obtain the string  $t$  by typing the string  $s$  and replacing some characters with presses of "Backspace" button, or "NO" if you cannot.

You may print each letter in any case (YES, yes, Yes will all be recognized as positive answer, NO, no and nO will all be recognized as negative answer).

### Input

```
4
ababa
ba
ababa
bb
aaa
aaaa
aababa
ababa
```

### Output

```
YES
NO
NO
YES
```

### Note

Consider the example test from the statement.

In order to obtain "ba" from "ababa", you may press Backspace instead of typing the first and the fourth characters.

There's no way to obtain "bb" while typing "ababa".

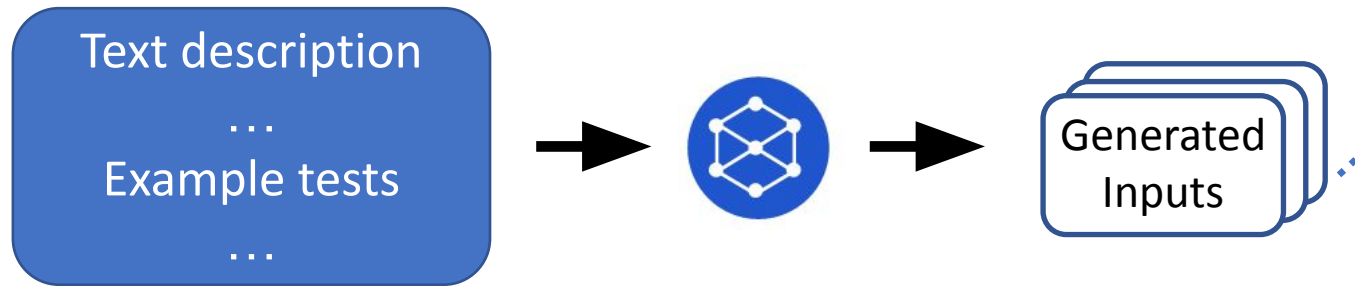
There's no way to obtain "aaaa" while typing "aaa".

In order to obtain "ababa" while typing "aababa", you have to press Backspace instead of typing the first character, then type all the remaining characters.

- The problem includes: (1) long and complicated text description; and (2) a few input-output pairs as test cases
- The code needs to pass both given and held-out test cases

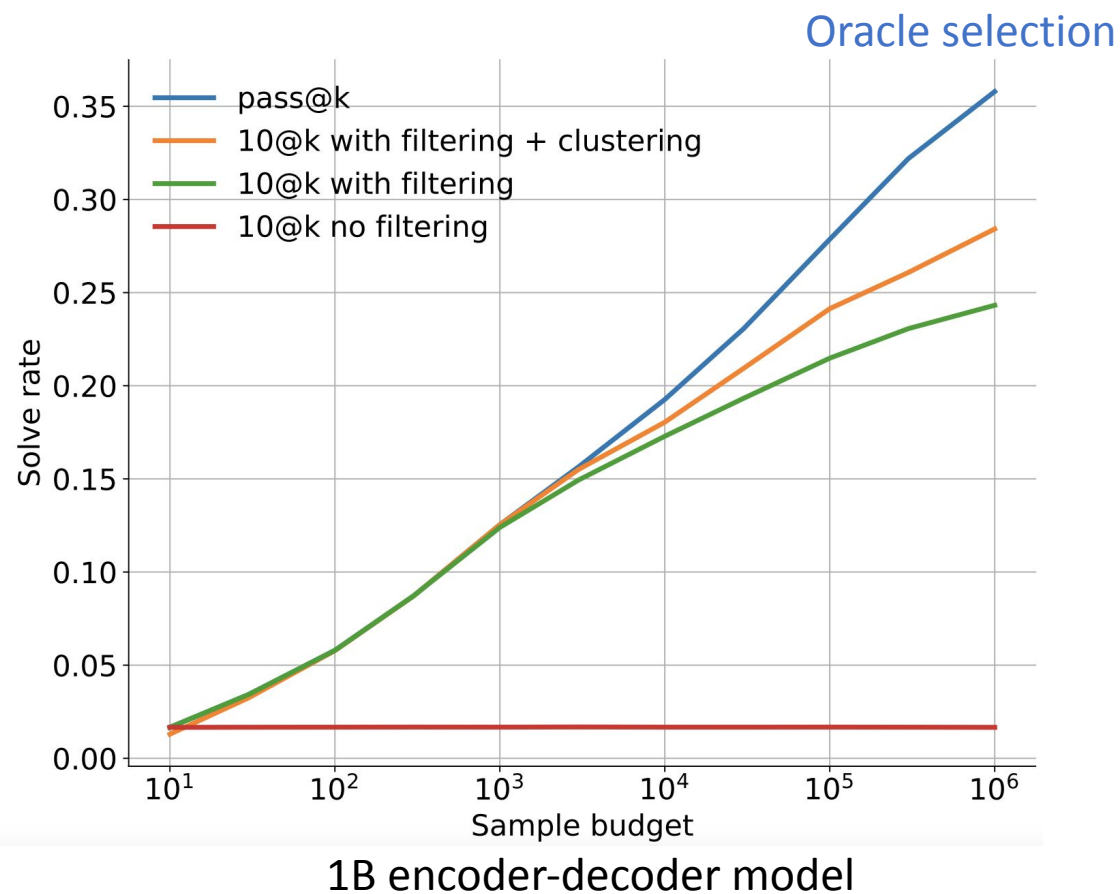
# Clustering by **execution** on generated inputs

- We can filter programs that fail the given test cases, but the remaining programs might still fail on the held-out test cases
- Train a model to generate new test inputs



- **Execute** sampled programs on all **generated inputs**
- Cluster all programs with the same outputs together
  - Assumption: all programs in the same cluster are semantically equivalent if the generated inputs are diverse and of high quality
- Sample 1 program from each of the 10 largest clusters

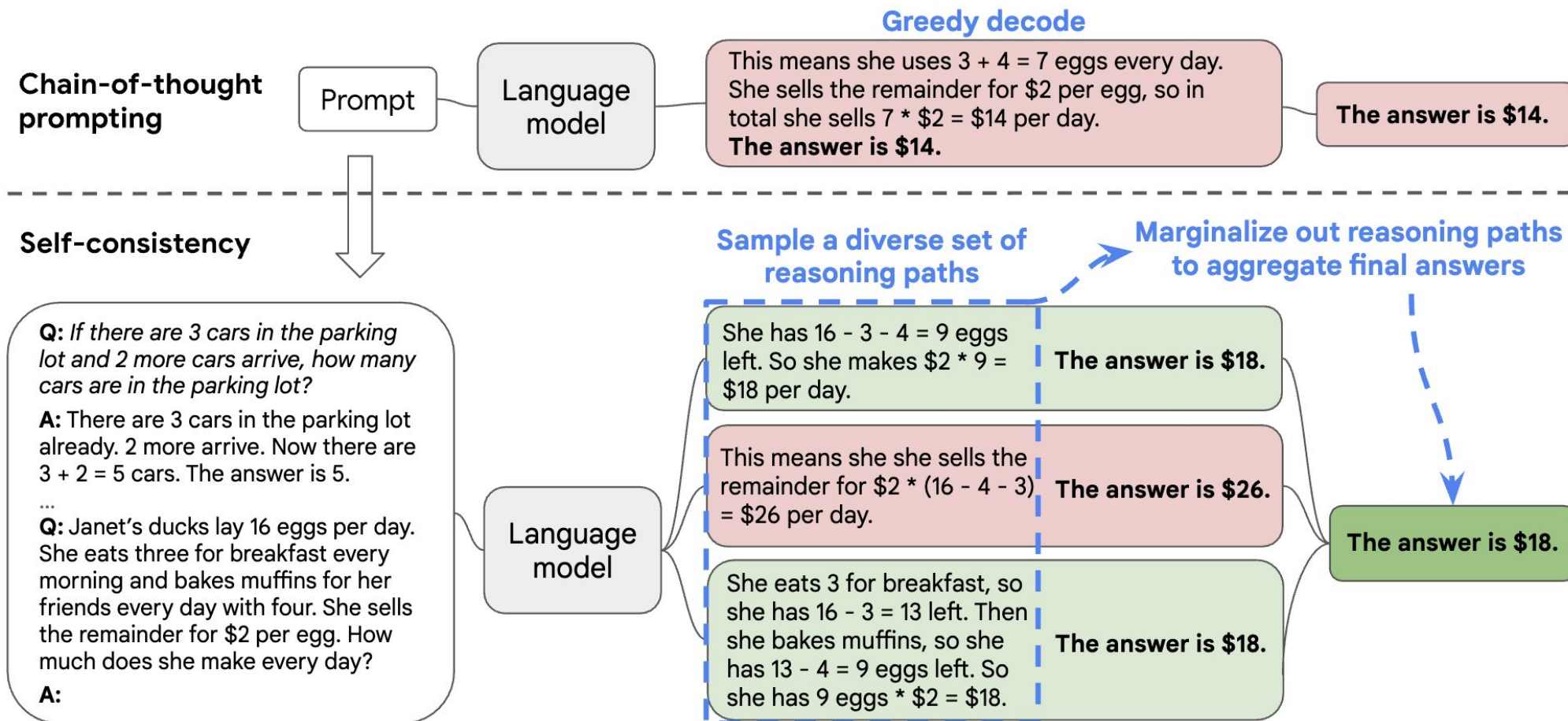
# Results on Codeforces



- Clustering provides additional performance gain over filtering only
- Still a gap from the oracle selection

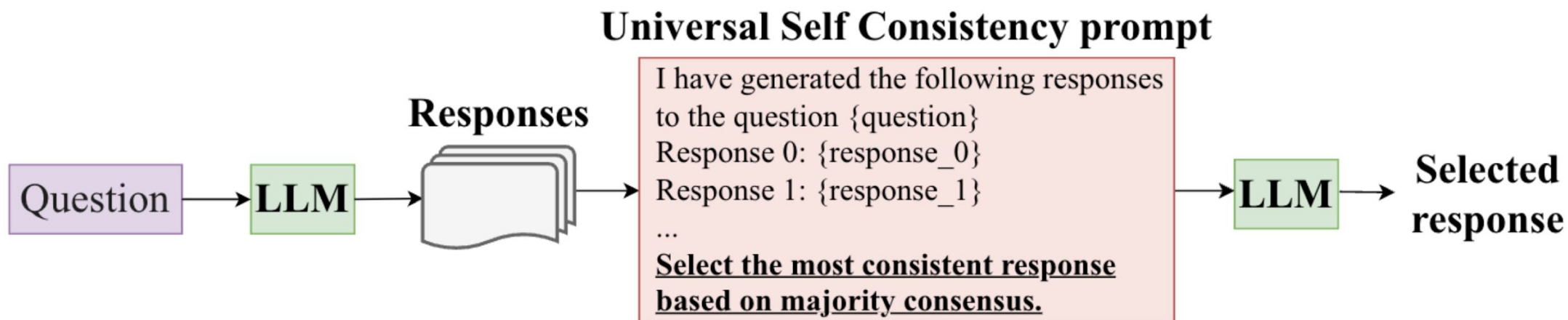


# Limitation of self-consistency decoding: require an answer extraction process

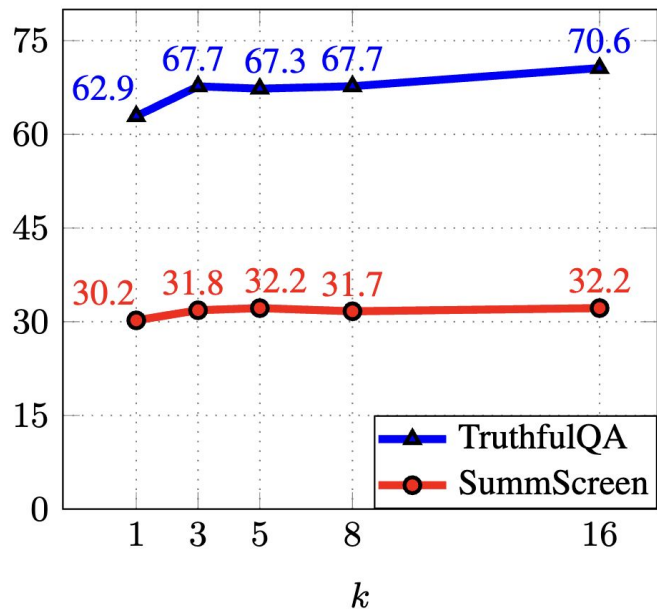


Can we enable consistency-based decoding for **free-form generation**?

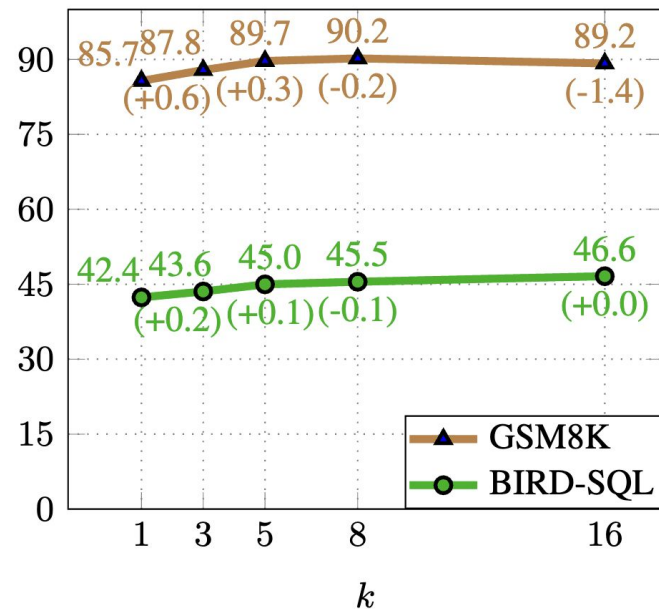
# Universal self-consistency: ask the LLM to perform consistency-based selection



# Results



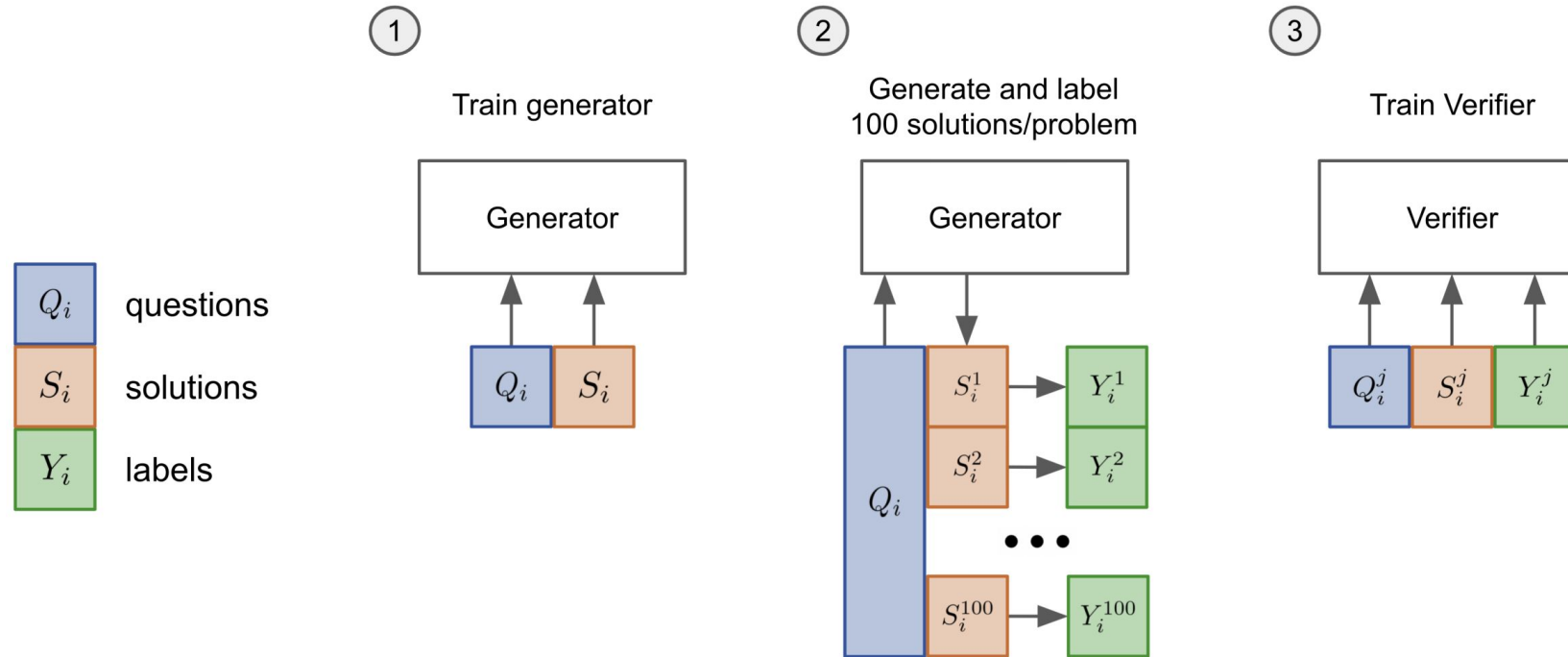
(a) Results on open-ended generation.



(b) Results on GSM8K and BIRD-SQL. The top numbers are USC accuracies, and the bottom numbers are the differences to SC accuracies.

- Universal Self-Consistency (USC) improves the performance on open-ended generation (summarization, QA), where the original self-consistency is inapplicable.
- USC matches self-consistency performance on math reasoning and coding.
  - USC does not require answer extraction and code execution.
- USC performance is bounded by the long-context capability.

# Improve further over consistency-based selection: training LLM rankers

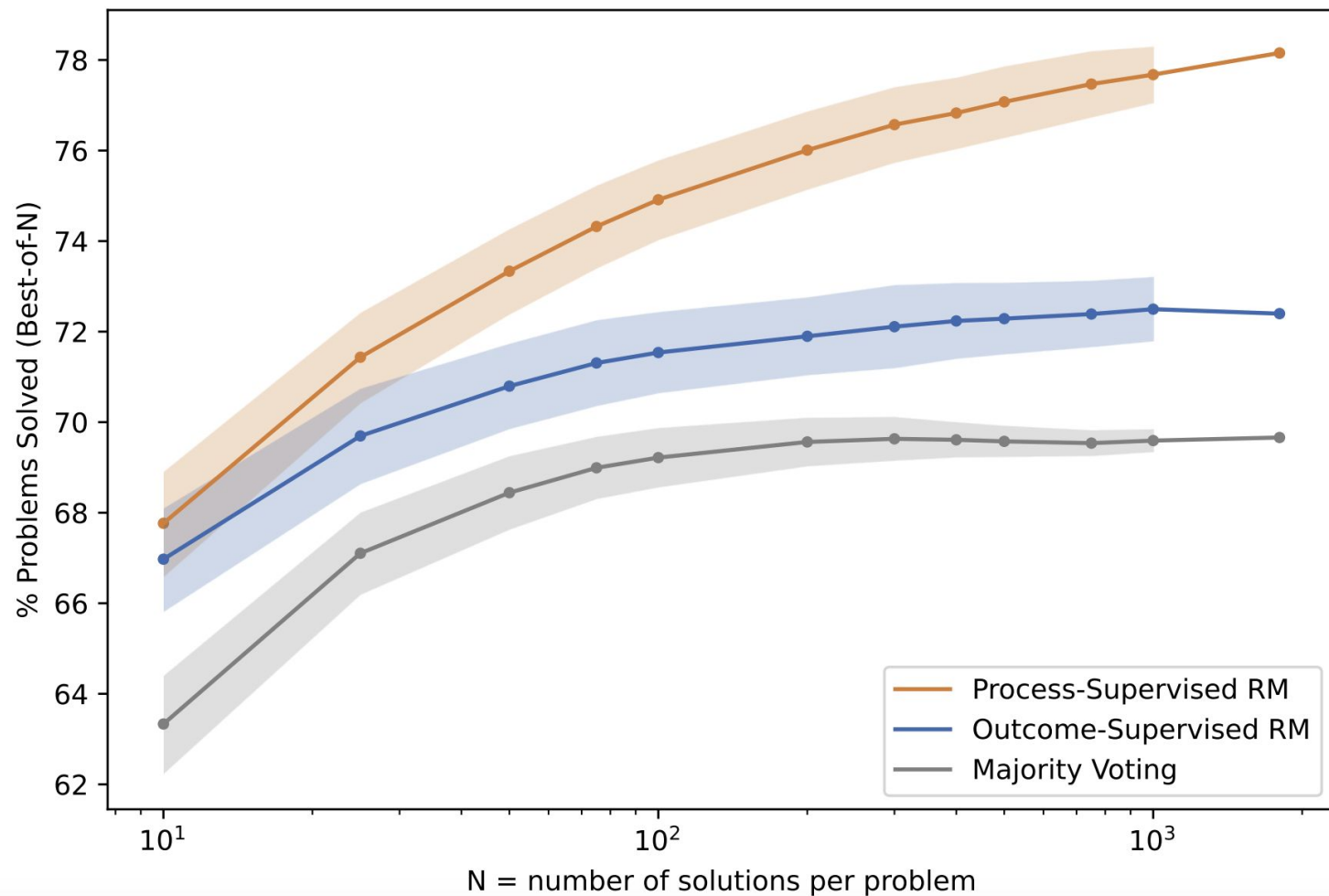


Two types of LLM-based verifiers/reward models

- Outcome-supervised Reward Model (ORM): verify at the **solution** level
- Process-supervised Reward Model (PRM): verify at the **step** level for each solution

# (Strong) LLM-based verifiers outperform consistency-based selection

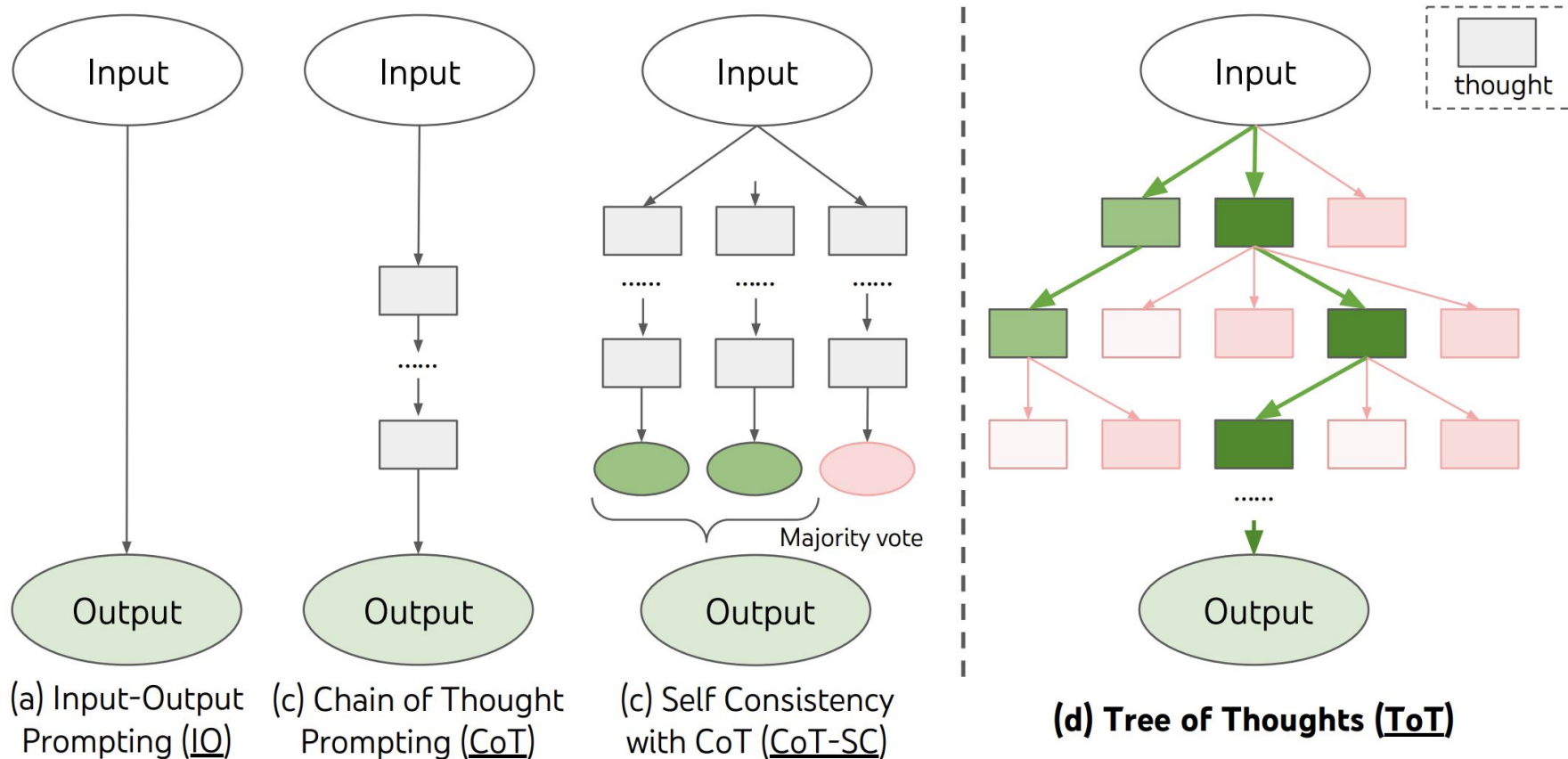
	ORM	PRM	Majority Voting
% Solved (Best-of-1860)	72.4	<b>78.2</b>	69.6



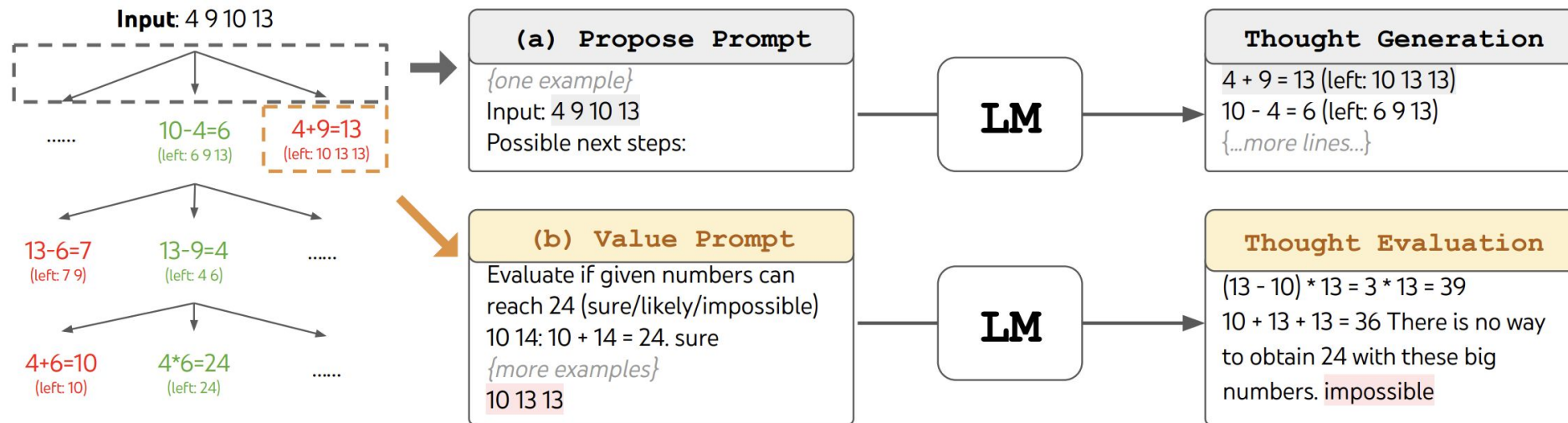
- Process-supervised reward model (PRM) scales better with more samples
- Note: the performance is highly dependent on the verifier quality
  - The same verifier might not generalize across tasks

# So far: response selection only after the full responses are generated

- This does not fully utilize a step-wise scorer!
- LLM + tree search: prioritize the exploration of more promising partial solutions



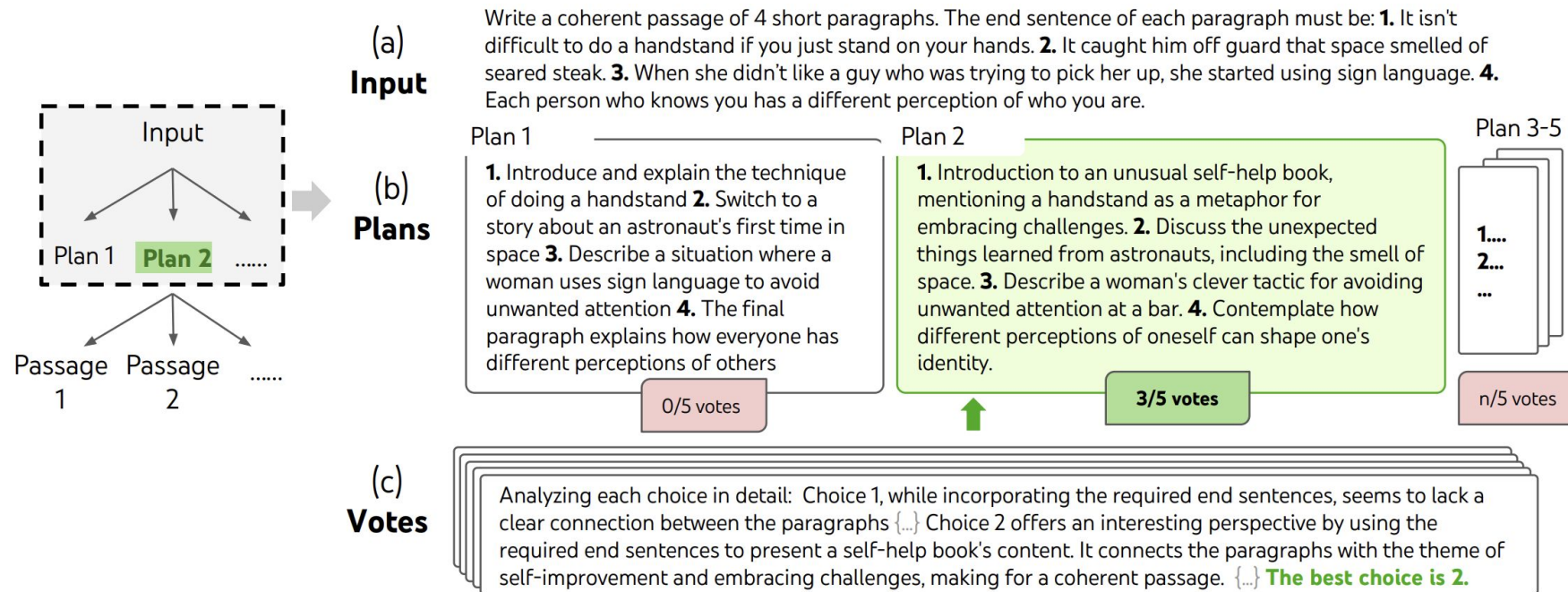
# Tree-of-thought example: game of 24



At each step:

- Thought generation: prompt the LLM to propose possible next thinking steps
- Thought evaluation: prompt the LLM to evaluate how promising the current state is

# Voting-based state evaluation



LLM selects the best state among the candidates

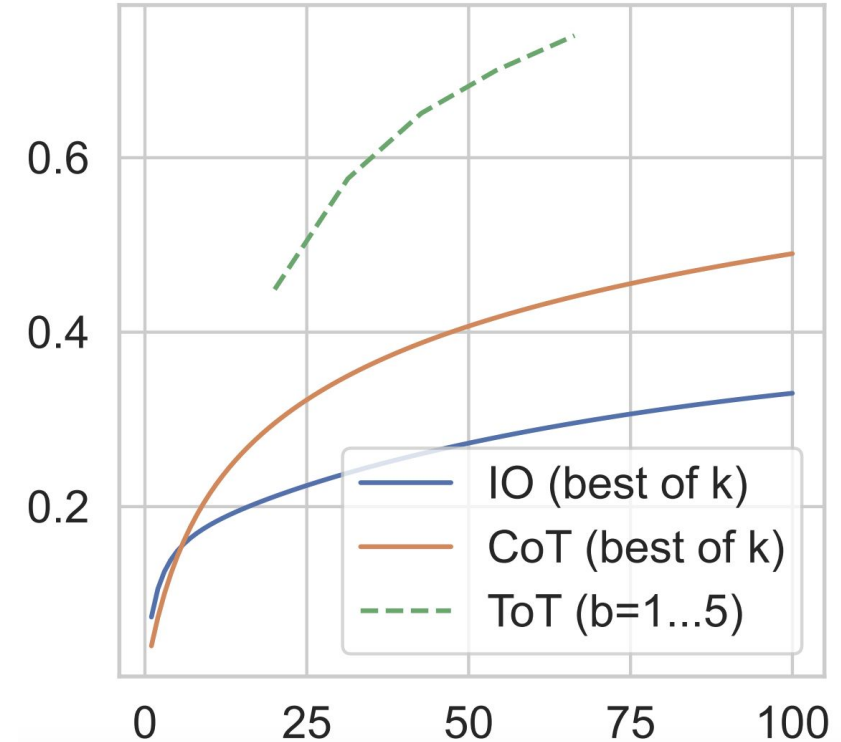
- LLM votes multiple times, then selects the majority vote as the final choice



## Tree-of-thought results: game of 24

Method	Success
IO prompt	7.3%
CoT prompt	4.0%
CoT-SC (k=100)	9.0%
ToT (ours) (b=1)	45%
ToT (ours) (b=5)	<b>74%</b>

(a) Success rate with nodes visited



- ToT with breadth-first search (BFS) scales better than standard prompting and CoT w.r.t. token budget
- Going further: we can integrate more advanced search algorithms, e.g., Monte-Carlo Tree Search (MCTS)
  - Need a good LLM + prompt design for self-evaluation

# Summary

- We can further scale the inference-time compute by sampling multiple branches in the solution space
- Consistency-based selection: a simple, effective and general principle
  - Self-Consistency: marginalize out reasoning paths and select based on the final answer
  - Code generation: reranking based on execution consistency
- When LLM self-evaluation works well: search in the partial solution space can help

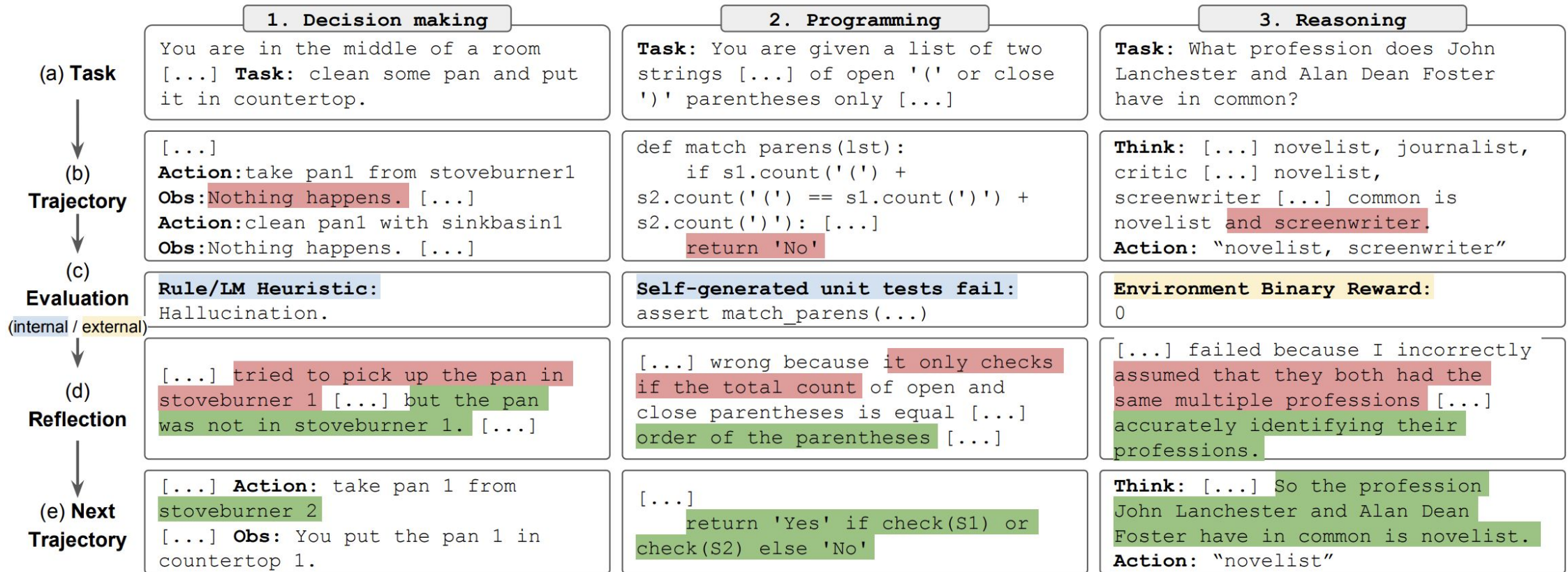
# Outline

- Part 1: Introduction to basic prompting techniques
  - Use more token budget to generate a single solution
- Part 2: Search and selection from multiple candidates
  - Increase the width to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the **depth** to reach the final solution

## Even the best LLMs still make (sometimes obvious) mistakes...

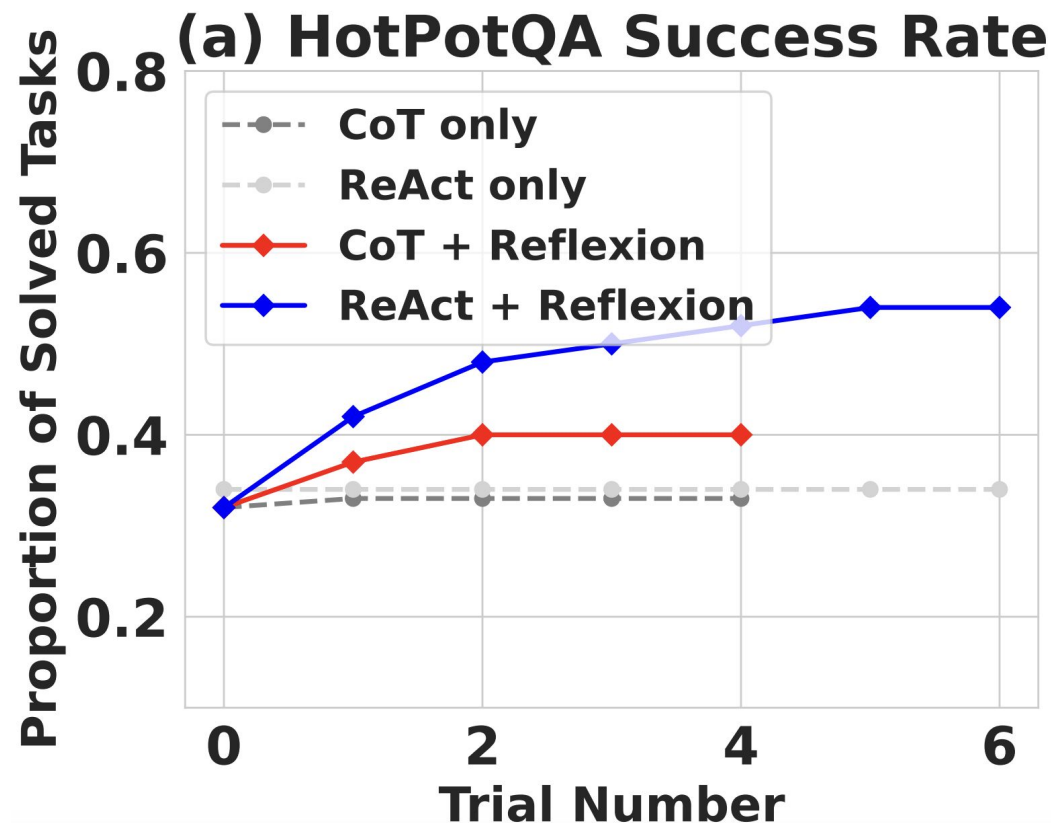
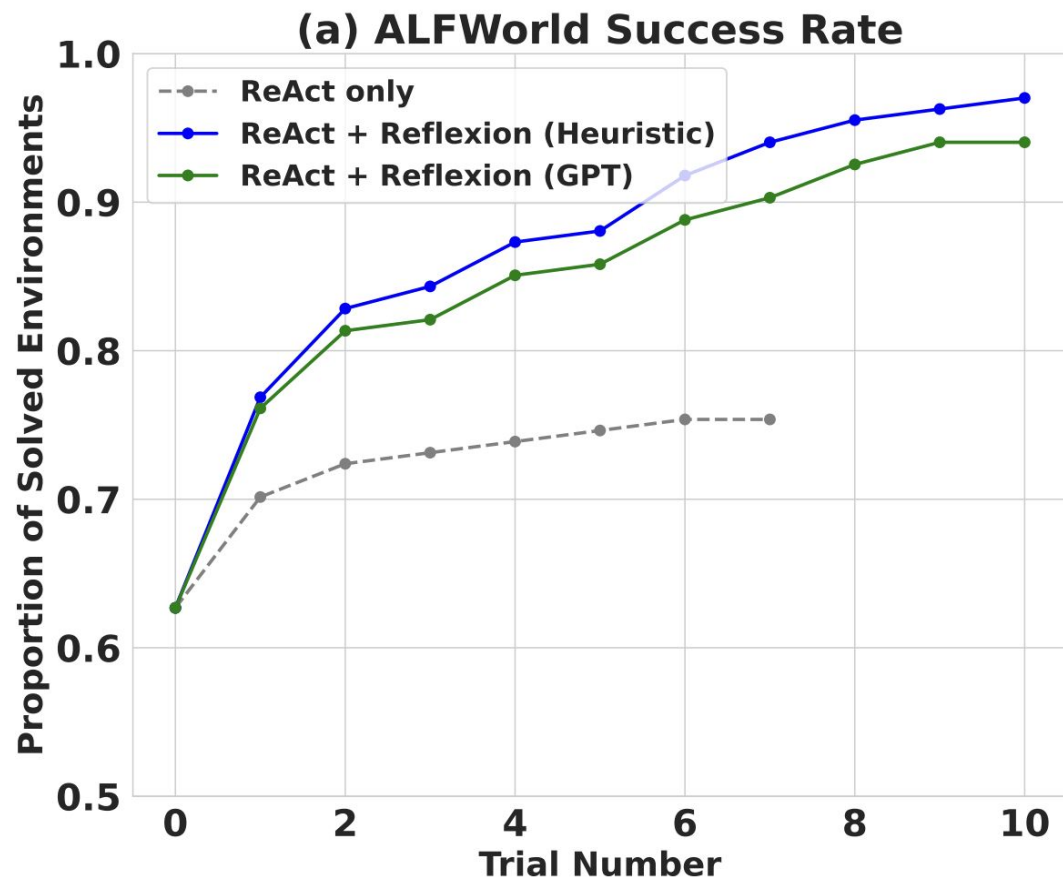
- But humans also tend to make (sometimes trivial) mistakes at first thought!
- Sampling multiple solutions can reduce mistakes from a single prediction, but it is still suboptimal
  - No feedback loop to correct the mistakes after a complete solution is generated
- Inference-time self-improvement: LLM iteratively improves its own response for the given task

# Reflexion and Self-Refine



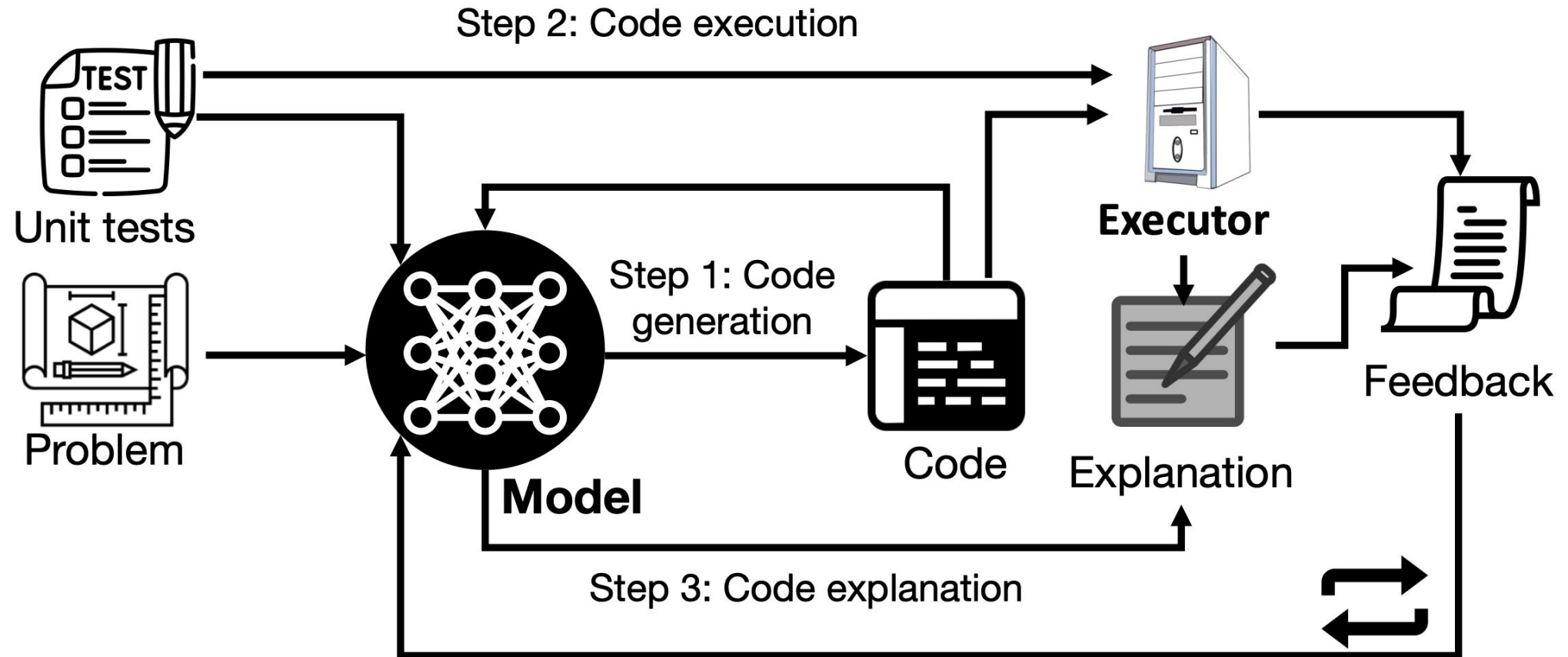
- LLM generates feedback on its output. Use external evaluation when available.
- LLM self-refines its output given both internal feedback and external evaluation.

# Self-reflection and self-refinement work with good (external) evaluation



- Reflexion improves on tasks with effective evaluation heuristics, e.g., ALFWorld.
- On HotPotQA: the external evaluation gives the answer correctness at each reflection step.

# Self-debugging is a natural workflow for code generation



Code execution provides natural external feedback: humans often debug better within an IDE

# Self-debugging with different feedback formats

**[C++] =**  

```
int count_trailing_zeroes_factorial_number ( int n ) {  
    int count = 0;  
    for ( int i = 5; n / i >= 1; i *= 5 )  
        count += n / i;  
    return count;  
}
```

**[Original Python] =**  

```
def count_trailing_zeroes_factorial_number(n):  
    cnt = 0  
    i = 5  
    while ((n / i) >= 1):  
        cnt = (cnt + (n / i))  
        i *= 5  
    return cnt
```

**[Simple Feedback] =**  
The above Python translation does not do the same thing as the C++ code. Correct the Python translation.

**[UT Feedback] =**  
The Python translation does not do the same thing as the C++ code. These are the results of failed unit tests that test whether the Python translation's outputs match the C++ program's outputs:

Failed: assert  
count\_trailing\_zeroes\_factorial\_number(9) == 1  
Actual Result: 1.8

Correct the translation.

**[Revised Python #n] =**  

```
def count_trailing_zeroes_factorial_number(n):  
    cnt = 0  
    i = 5  
    while ((n / i) >= 1):  
        cnt = (cnt + (n // i))  
        i *= 5  
    return cnt
```

**[C++ Explanation] =**  
The code is an implementation of counting the number of trailing zeroes in the factorial of a given number.  

```
int count = 0;
```

  
The variable count is used to record the number of trailing zeroes.  

```
for ( int i = 5; n / i >= 1; i *= 5 )
```

  
The loop iterates until the quotient of the given number and i (the divisor) is greater than or equal to 1.  

```
count += n / i;
```

  
When the loop is executed, the quotient of the given number and i will be added to the variable count.  

```
return count;
```

  
Finally, the function returns the number of trailing zeroes.

**[Python Explanation] =**  
The code is an implementation of counting the number of trailing zeroes in the factorial of a given number.  

```
cnt = 0;
```

  
The variable cnt is used to record the number of trailing zeroes.  

```
i = 5
```

  
The variable i is used as the divisor.  

```
while ((n / i) >= 1):
```

  
The loop iterates until the quotient of the given number and i (the divisor) is greater than or equal to 1.  

```
cnt = (cnt + (n / i))
```

  
When the loop is executed, the quotient of the given number and i will be added to the variable cnt.  

```
i *= 5
```

  
The variable i will be multiplied by 5.  

```
return cnt
```

  
Finally, the function returns the number of trailing zeroes.

**[Trace] =**  
1. Initialize `n` as 9, `cnt` as 0, `i` as 5.  
2. Enter the while loop since `(9 / 5)` is greater or equal to 1.  
3. `cnt` is assigned `cnt + (9 / 5)` which evaluates to 1.8.  
4. `i` is multiplied by 5, becomes 25.  
5. Check the condition again `(9 / 25)` is not greater or equal to 1. Exit the while loop.  
6. Return `cnt` which is 1.8.

The incorrect translation fails because the division `n / i` in Python returns a float value, whereas in C++ it returns an integer value. This issue affects the value assigned to `cnt` and results in an incorrect output.

Simple: a short universal feedback for all wrong code

Unit test feedback: include the execution results

Code explanation: line-by-line explanation of the implementation

Trace: line-by-line simulation of the execution trace



# Self-debugging results

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	<b>72.2</b>	73.4	<b>64.9</b>
+Expl.	<b>84.1</b>	<b>72.2</b>	<b>73.6</b>	<b>64.9</b>

(b) Results on TransCoder.

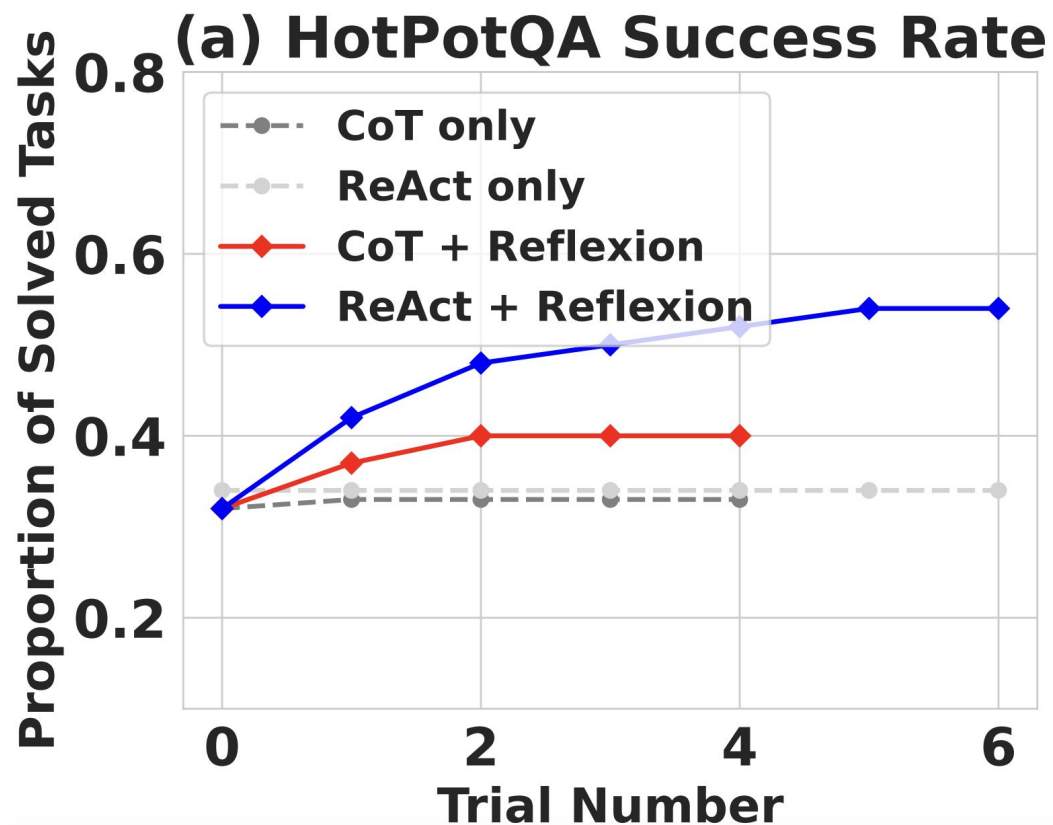
TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	<b>92.7</b>	88.8	76.4
+ Expl.	<b>92.5</b>	<b>92.7</b>	<b>90.4</b>	<b>76.6</b>
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	<b>80.6</b>	52.2
+ Expl.	69.8	<b>74.2</b>	80.4	52.2
+ Trace.	<b>70.8</b>	72.8	80.2	<b>53.2</b>

- Self-debugging consistently boosts the performance across different LLMs
- More informative feedback further improves the debugging performance

# How does self-correction work for QA-style reasoning tasks?



	GSM8K	MultiArith	AddSub	SVAMP	SingleEq
Zero-Shot	78.35	96.06	85.83	78.35	91.34
Zero-Shot + RCI	85.43	97.64	89.76	84.65	<b>94.49</b>
Zero-Shot CoT	82.28	96.85	83.86	79.92	89.37
Zero-Shot CoT + RCI	<b>86.22</b>	97.24	89.88	85.83	90.94
Few-Shot CoT	80.31	98.82	89.37	83.46	91.73
Few-Shot CoT + RCI	84.25	<b>99.21</b>	<b>90.55</b>	<b>87.40</b>	93.70

- Some prior work show improvement with self-correction, but using an oracle verifier.
- Oracle verifier is not available in most use cases. How do LLMs perform without such external feedback?

# Self-correction without oracle feedback hurts the reasoning performance

		GSM8K	CommonSenseQA	HotpotQA
GPT-3.5	Standard Prompting	75.9	75.8	26.0
	Self-Correct (Oracle)	84.3	89.7	29.0
GPT-4	Standard Prompting	95.5	82.0	49.0
	Self-Correct (Oracle)	97.5	85.5	59.0

		# calls	GSM8K	CommonSenseQA	HotpotQA
GPT-3.5	Standard Prompting	1	<b>75.9</b>	<b>75.8</b>	<b>26.0</b>
	Self-Correct (round 1)	3	75.1	38.1	25.0
	Self-Correct (round 2)	5	74.7	41.8	25.0
GPT-4	Standard Prompting	1	<b>95.5</b>	<b>82.0</b>	<b>49.0</b>
	Self-Correct (round 1)	3	91.5	79.5	<b>49.0</b>
	Self-Correct (round 2)	5	89.0	80.0	43.0

- Oracle: utilize the ground truth answer for correction
- Without oracle feedback, LLMs need to judge the response correctness themselves
- LLMs can wrongly judge the correctness of its predictions, leading to worse performance after self-correction

# General-purpose feedback prompt variants do not improve the performance

	# calls	GSM8K	CommonSenseQA
Standard Prompting	1	91.5	84.0
<i>Feedback Prompt:</i> Assume that this answer could be either correct or incorrect. Review the answer carefully and report any serious problems you find.			
Self-Correct (round 1)	3	88.0	81.5
Self-Correct (round 2)	5	90.0	83.0
<i>Feedback Prompt:</i> Review your previous answer and determine whether it's correct. If wrong, find the problems with your answer.			
Self-Correct (round 1)	3	90.0	74.5
Self-Correct (round 2)	5	90.0	81.0
<i>Feedback Prompt:</i> Verify whether your answer is correct, and provide an explanation.			
Self-Correct (round 1)	3	91.0	81.5
Self-Correct (round 2)	5	91.0	83.5

Edit the feedback prompt affects the self-correction behavior (tendency to keep the initial response), but none of them significantly improves over the initial performance.

# Multi-agent debate does not improve over self-consistency

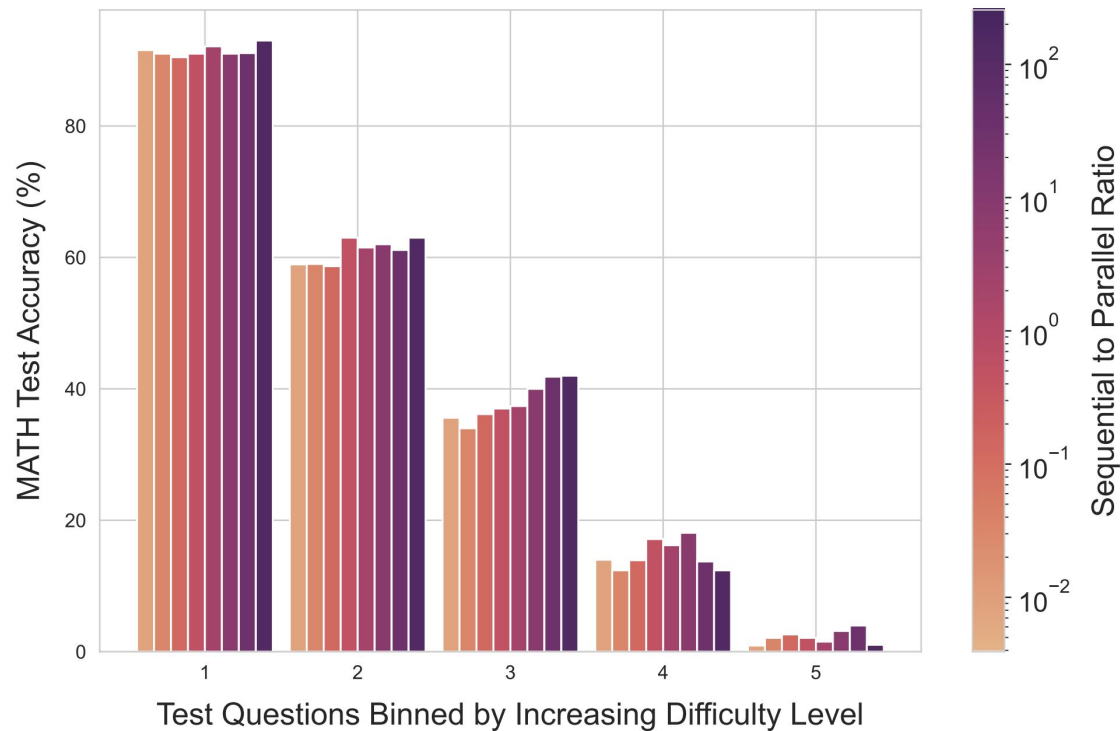
	# responses	GSM8K
Standard Prompting	1	76.7
Self-Consistency	3	82.5
Multi-Agent Debate (round 1)	6	83.2
Self-Consistency	6	85.3
Multi-Agent Debate (round 2)	9	83.0
Self-Consistency	9	<b>88.2</b>

- Multi-agent debate: prompt the LLM to review multiple responses and give an updated one.
- Recall: self-consistency selects the response with the most common final answer.
- Without a good evaluator, multi-agent debate does not effectively utilize the token budget.

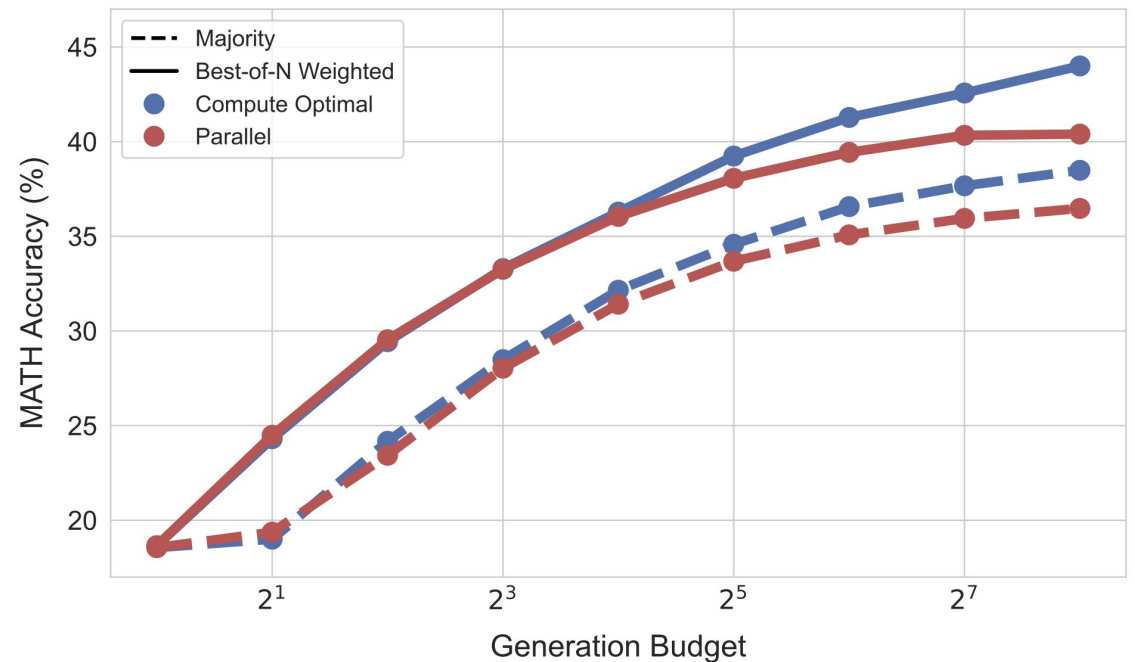
# Putting everything together: how to best utilize the token budget

- How to balance the inference budget for generating multiple samples
  - In parallel or sequentially?
- This is mostly a model-specific and task-specific empirical question, depending on the model's self-reflection and correction abilities.

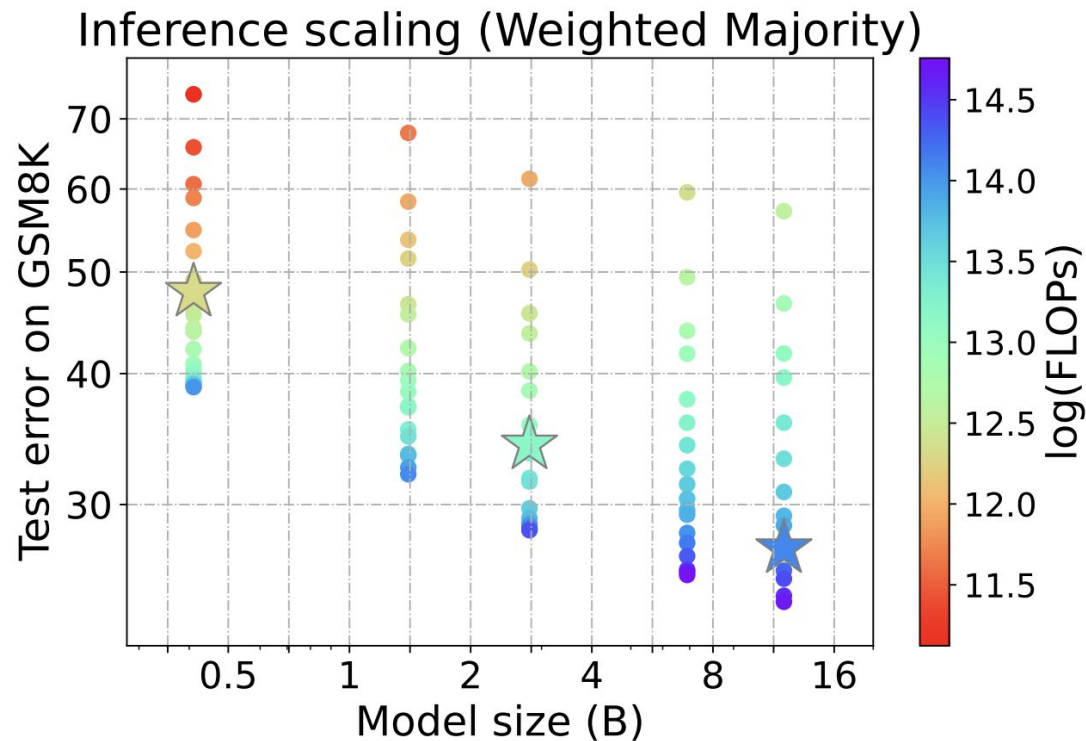
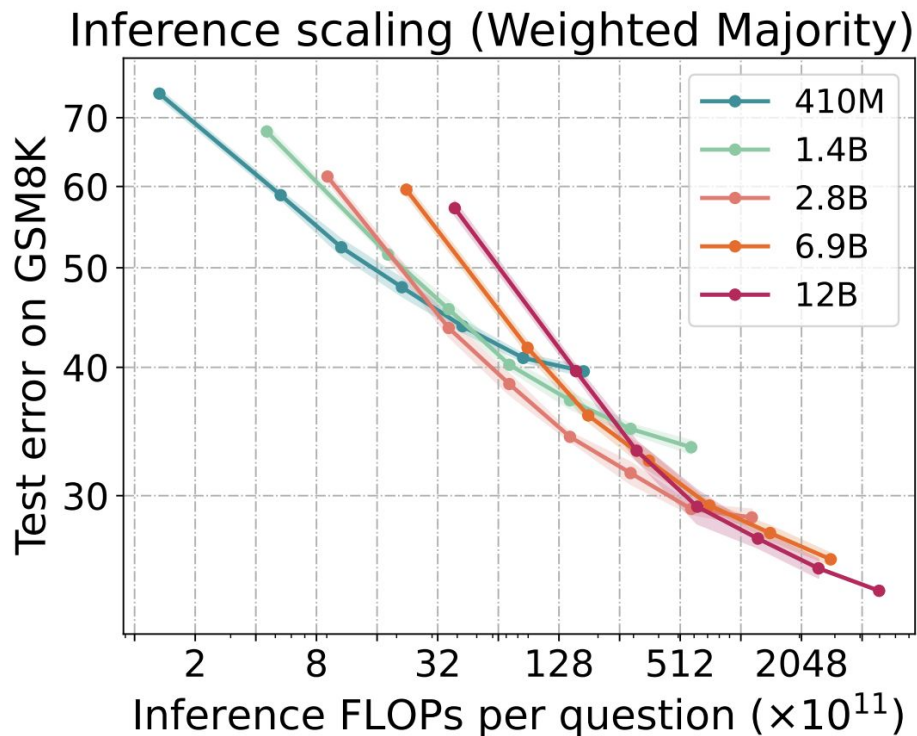
Revisions@128, Varying the Sequential to Parallel Ratio



Compute Optimal Revisions



# Another factor for optimizing inference cost: model size



- With the same FLOPs budget, we can sample more solutions from a lighter model.
- The optimal model with different inference budget can be different

# What we covered in this lecture

- Part 1: Introduction to basic prompting techniques
  - Use **more token budget** to generate **a single solution**
- Part 2: Search and selection from multiple candidates
  - Increase the **width** to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the **depth** to reach the final solution

**The best practice to interact with an LLM should be adapted according to its capabilities.**



# General principle of how to design effective reasoning techniques

- [The Bitter Lesson](#) from Richard Sutton (again) is an important guideline for designing reasoning techniques, including both inference-time and training-time algorithms.
- “One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that **continue to scale with increased computation** even as the available computation becomes very great.”
- “We want AI agents that can **discover like we can, not which contain what we have discovered**. Building in our discoveries only makes it harder to see how the discovering process can be done.”

# Thanks!

- Part 1: Introduction to basic prompting techniques
  - Use **more token budget** to generate **a single solution**
- Part 2: Search and selection from multiple candidates
  - Increase the **width** to explore the solution space
- Part 3: Iterative self-improvement
  - Increase the **depth** to reach the final solution

**The best practice to interact with an LLM should be adapted according to its capabilities.**