



**COS 484**

# Reasoning with LLMs

Spring 2026

*Slide credit: Xinyun Chen, Denny Zhou*

# What is LLM Reasoning?

Intermediate steps / tokens



*Ling et al., Program Induction by Rationale Generation, ACL 2017 — the first paper to use natural language for intermediate reasoning steps.*

## What is the output when concatenating the last letter of each word in "artificial intelligence"?

NO REASONING

The answer is "le".

REASONING

The last letter of "artificial" is "l". The last letter of "intelligence" is "e". Concatenating "l" and "e" leads to "le". So the answer is "le".

## The first paper on natural-language intermediate reasoning

### **Problem 1:**

**Question:** Two trains running in opposite directions cross a man standing on the platform in 27 seconds and 17 seconds respectively and they cross each other in 23 seconds. The ratio of their speeds is:

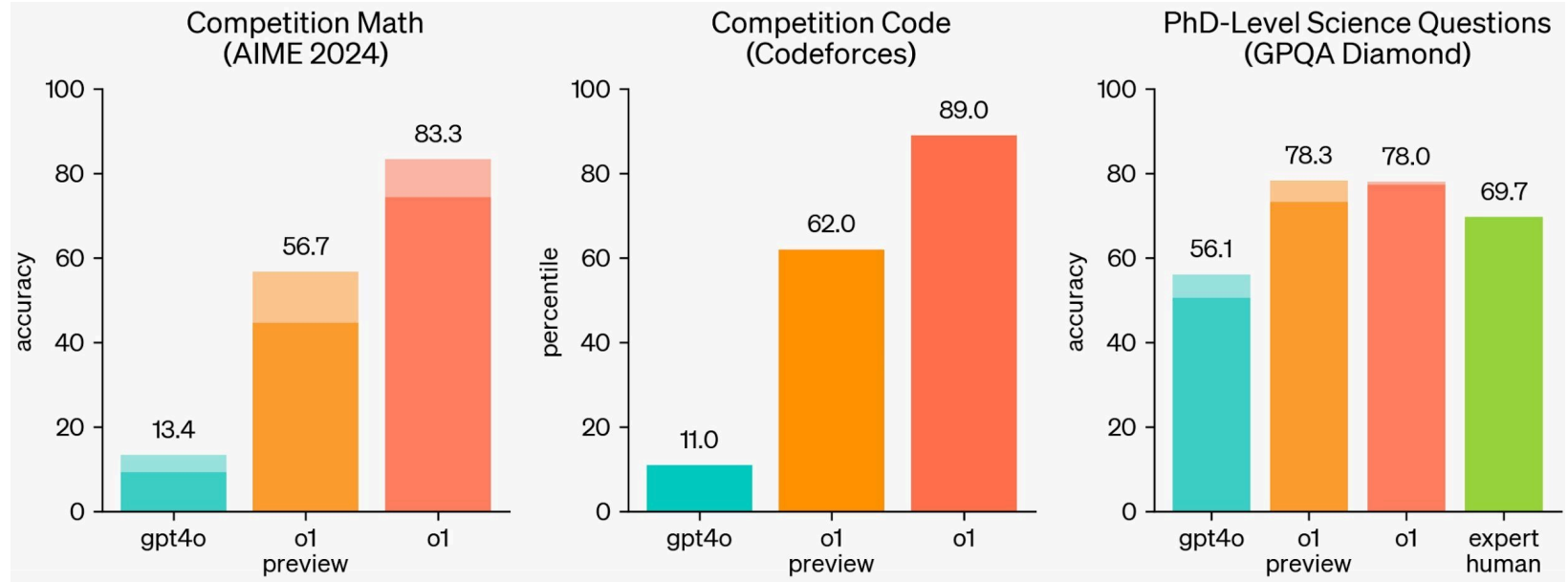
**Options:** A)  $3/7$  B)  $3/2$  C)  $3/88$  D)  $3/8$  E)  $2/2$

**Rationale:** Let the speeds of the two trains be  $x$  m/sec and  $y$  m/sec respectively. Then, length of the first train =  $27x$  meters, and length of the second train =  $17y$  meters.  $(27x + 17y) / (x + y) = 23 \rightarrow 27x + 17y = 23x + 23y \rightarrow 4x = 6y \rightarrow x/y = 3/2$ .

**Correct Option:** B

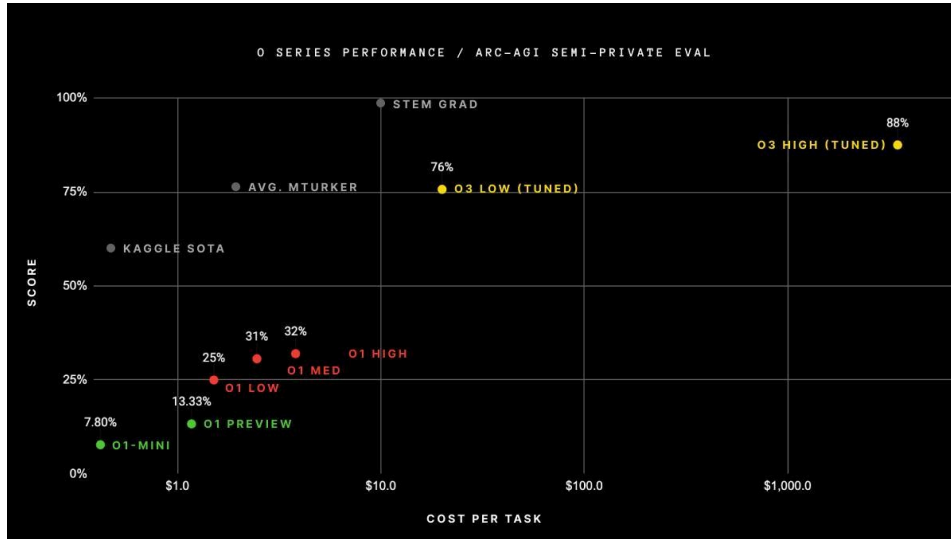
Ling et al., *Program Induction by Rationale Generation*, ACL 2017 — first paper to describe intermediate reasoning steps in natural language.

# Highlight of LLMs in 2024-2026: advancement of reasoning models



OpenAI **o1** achieved impressive performance across challenging reasoning tasks in **math, coding, STEM**.

# Performance improves with more inference-time compute



- o3 achieved **87.5%** accuracy on ARC-AGI, with >\$1k test-time cost per task.
- Other existing LLMs achieved **<25%** accuracy without special inference-time techniques.

# 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. ✓

- 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

# 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

# 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?

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 post-training advances, standard prompting performance was poor on reasoning benchmarks.
- ❑ **Issue:** standard few-shot exemplars only show the *final-answer format*, 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.

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?

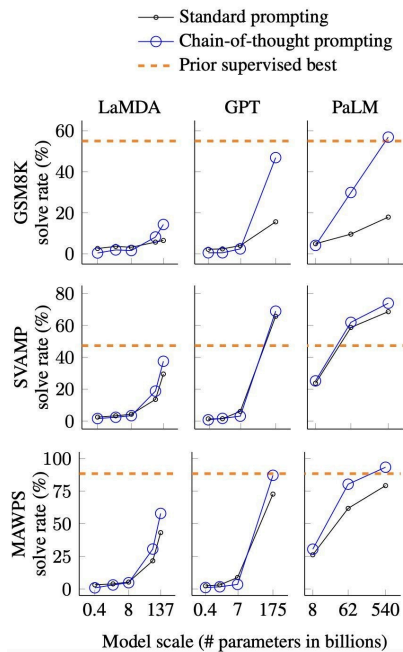
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

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 model size.
- Better models benefit more from CoT generation.
- Drastic improvement when the model reaches a certain scale.

# 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. ✗

(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 ✗

(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. ✓

The phrase **Let's think step by step** triggers CoT generation without any 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 **without manually labeling exemplars?**

## Analogical prompting: instruct the LLM to generate exemplars

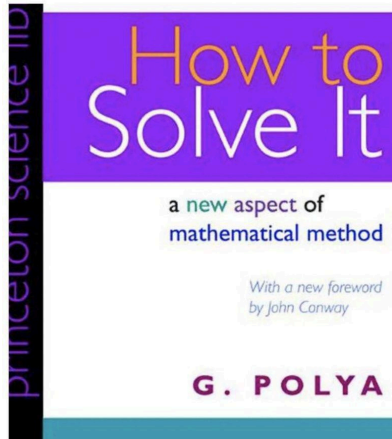
- Prompt the LLM to first recall relevant exemplars, then solve the test problem.
- Instruction: `## Recall relevant exemplars:` `## Solve the initial problem:`
- Example input: *“What is the area of the square with vertices (-2,2), (2,-2), (-2,-6), (-6,-2)?”*
- Model output: self-generated exemplar (square with side length 5, area =  $5^2 = 25$ ) → solution (area = 32).

## Analogical prompting — benefits

- **Exemplars are self-generated** by LLMs — no manual labeling.
- **Exemplars are tailored** to each individual problem.
- Works across math, code, temporal reasoning — out-of-the-box.

# Motivation: human analogical reasoning

- Humans are not explicitly given demonstrations for each new task.
- Instead, humans intrinsically **recall from past relevant experience**.



Contents		xiii
Condition		72
Contradictory†		73
Corollary		73
Could you derive something useful from the data?		73
Could you restate the problem?†		75
Decomposing and recombining		75
Definition	<b>Least-to-most prompting</b>	85
Descartes		92
Determination, hope, success		93
Diagnosis		94
Did you use all the data?		95
Do you know a related problem?		98
Draw a figure†		99
Examine your guess	<b>LLMs as analogical reasoners</b>	99
Figures		103
Generalization		108
Have you seen it before?		110
Here is a problem related to yours and solved before		110
Heuristic		112

**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:

- Beyond exemplars, the LLM can also self-generate **high-level knowledge**.
- Generated knowledge complements the problem with broader algorithmic 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: .....

## 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)
```

## Example: Codeforces (solving the initial problem)

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

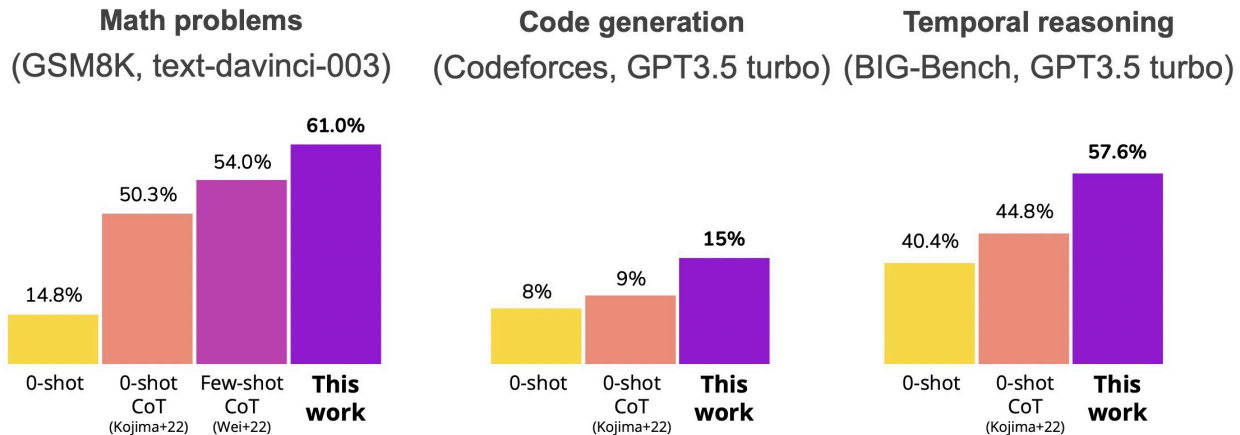
Explanation of the solution:

- 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**



## Stronger LLMs are better analogical reasoners

Prompting Method	(← scale down)			(scale up →)
	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>

- Weaker LLMs benefit less from analogical prompting — **but it does not hurt** zero-shot performance.
- With stronger LLMs, analogical prompting outperforms CoT with manually-designed or retrieved exemplars.

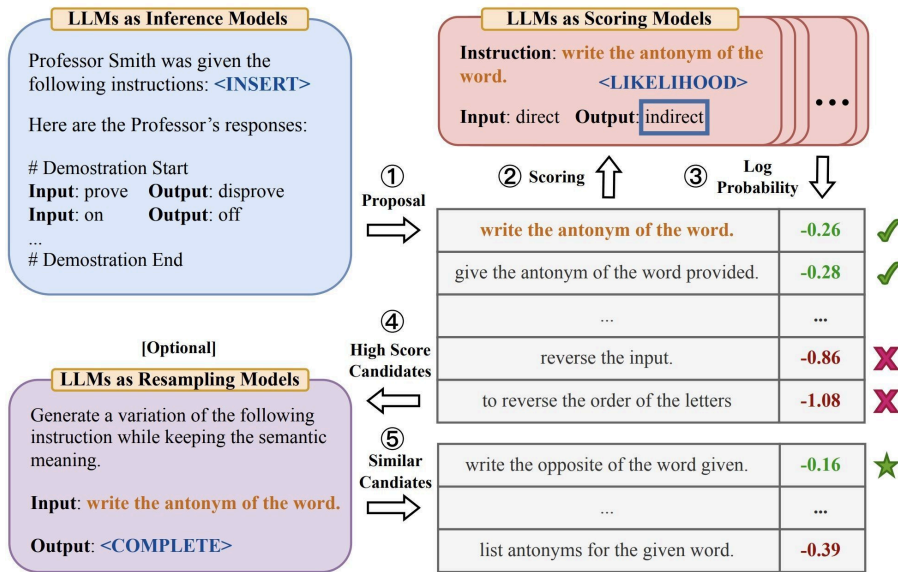
## Which instructions work for CoT generation?

No.	Category	Template	Accuracy
1	instructive	Let's think step by step.	<b>78.7</b>
2		First, [REDACTED]	77.3
3		Let's think about this logically.	74.5
4		Let's solve this problem by splitting it into steps. [REDACTED]	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

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

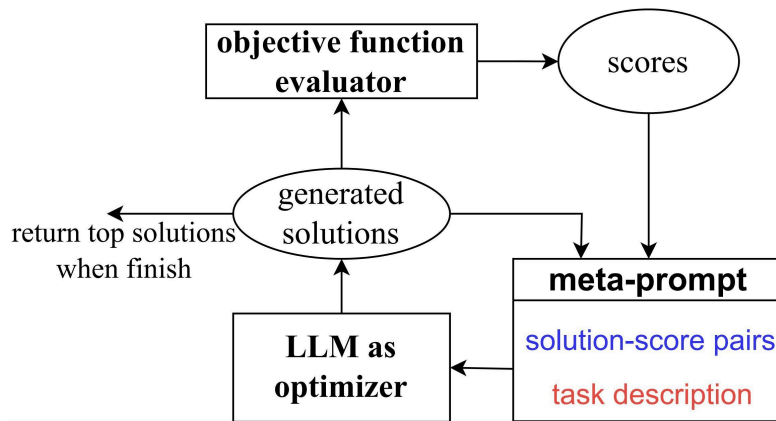
# Large language models for prompt engineering

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



**Proposal generation:** leverage the LLM to generate initial instructions. **Scoring:** score each instruction by 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.
- **Optimizer:** LLM proposes a new instruction. **Evaluator:** LLM evaluates accuracy.

## Meta-prompt for prompt optimization

- The meta-prompt shows the LLM past instructions & their scores, sorted ascending, then asks for a new one.
- **Past instructions & accuracies:** e.g. `Let's figure it out!` → 61 · `Let's solve the problem.` → 63.
- **Exemplars:** input Q–A pairs with `<INS>` as a placeholder for the new instruction.
- **Prompt goal:** *"Write your new text that is different from the old ones and has a score as high as possible."*

## 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. (empty string)	58.8 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: **Let's solve the problem.** at **60.8%**. Best LLM-generated prompt outperforms **Let's think step by step** by **~8%**, matching few-shot CoT (**80.7%**).

## Summary — Part 1

- **CoT generation:** variable computation adapting to different difficulty levels.
- **How to improve CoT at inference time:**
  - · Few-shot prompting with labeled thoughts
  - · Instruction prompting to trigger CoT
  - · Instruct the LLM to automate prompt design
- **Principles that hold:** 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

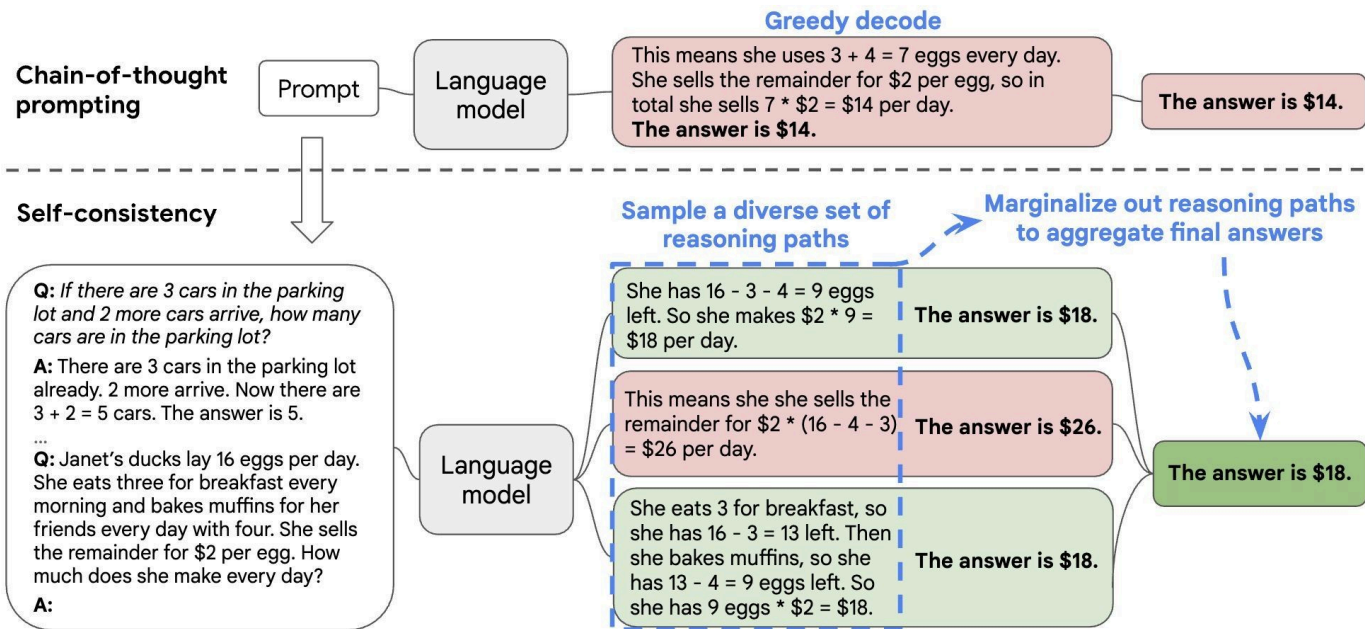
## **Search and selection from multiple candidates**

Increase the width to explore the solution space

## 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, or **multiple next steps** given the current partial thought.
- **Challenge:** how to select the best response from multiple candidates? In most cases we have no oracle scorer at inference time.

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



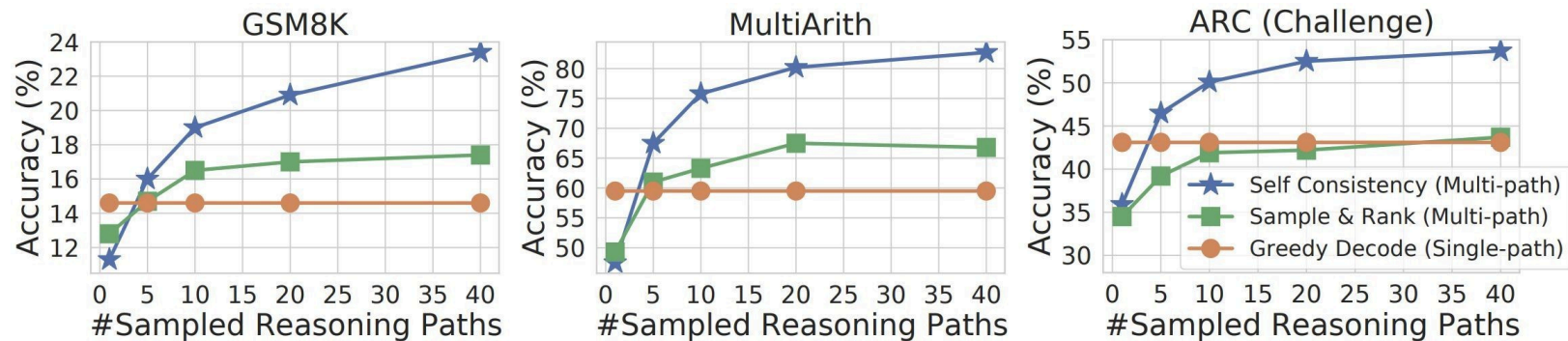
Selection is based only on the final answer — reasoning paths can differ across sampled responses.

## Self-Consistency boosts 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)

Consistent improvements across GSM8K, SVAMP, AQuA and others — for PaLM, GPT-3, UL2, Codex.

## Self-Consistency performance scales with more samples



- **Sample-and-Rank baseline:** select response with highest log probability.
- **Self-Consistency** scales much better than probability-based ranking.

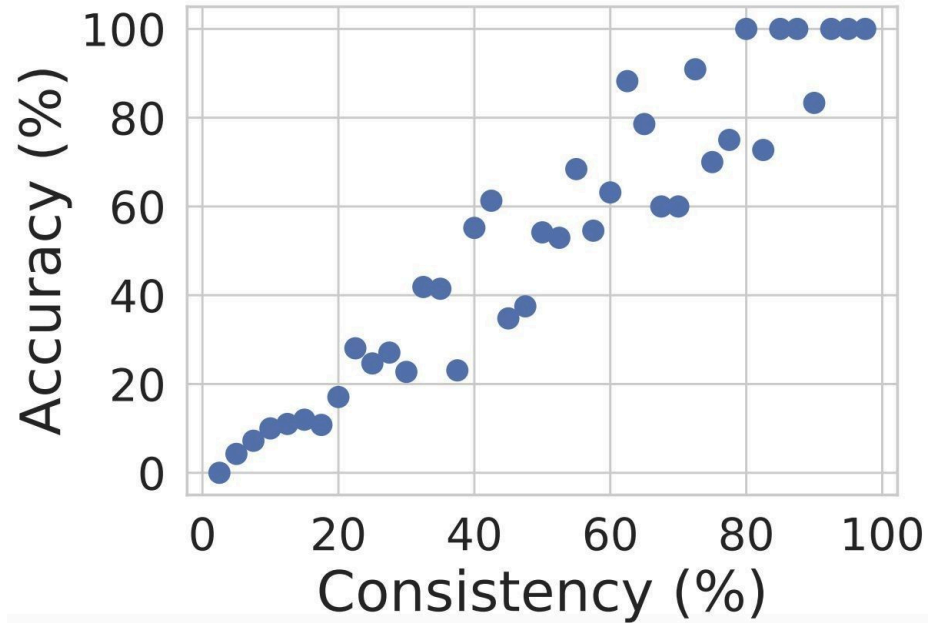
# Sampling diverse responses is crucial to self-consistency

- Beam search vs. sampling: SC needs diversity (high temperature, nucleus sampling).

	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 ± 0.5	57.1 ± 0.7	42.1 ± 0.6	76.6 ± 0.1	57.0 ± 0.2
Ensemble (40 prompt permutations)	19.2 ± 0.1	60.9 ± 0.2	42.7 ± 0.1	76.9 ± 0.1	57.0 ± 0.1
Self-Consistency (40 sampled paths)	<b>27.7 ± 0.2</b>	<b>75.7 ± 0.3</b>	<b>53.3 ± 0.2</b>	<b>79.3 ± 0.3</b>	<b>59.8 ± 0.2</b>

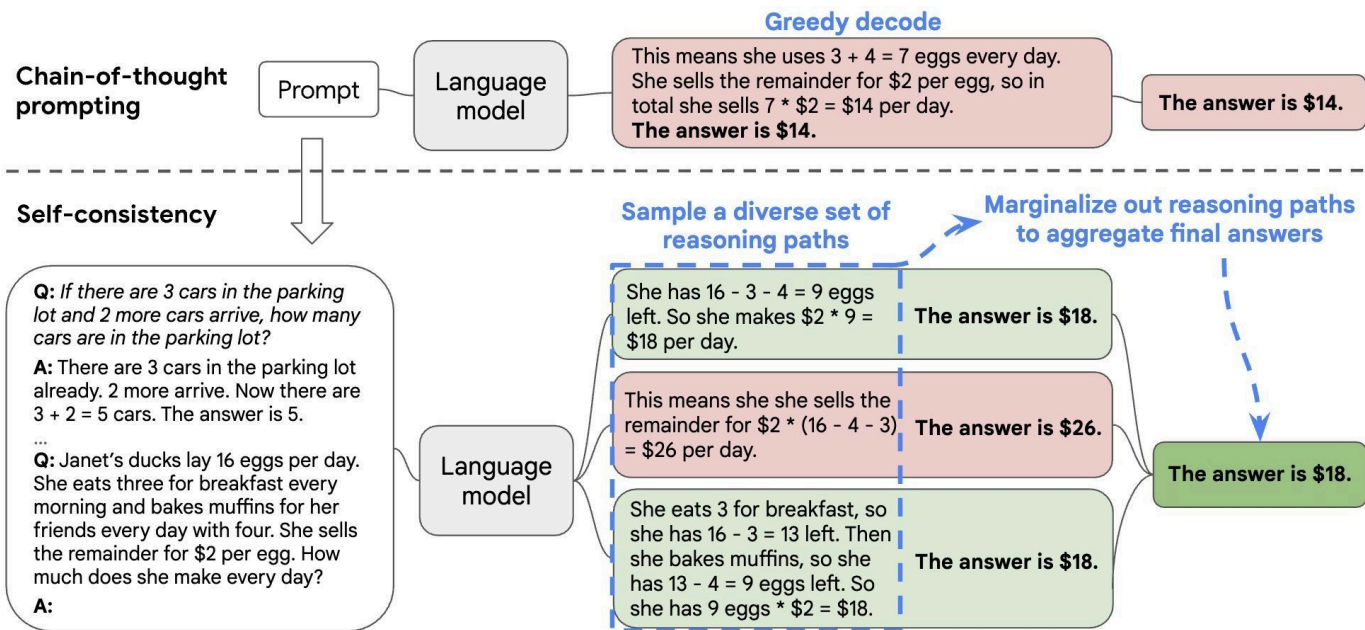
	Beam size / Self-consistency paths	1	5	10	20	40
AQuA	Beam search decoding (top beam)	23.6	19.3	16.1	15.0	10.2
	Self-consistency using beam search	23.6	19.8 ± 0.3	21.2 ± 0.7	24.6 ± 0.4	24.2 ± 0.5
	Self-consistency using sampling	19.7 ± 2.5	<b>24.9 ± 2.6</b>	<b>25.3 ± 1.8</b>	<b>26.7 ± 1.0</b>	<b>26.9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

## Consistency is highly correlated with the accuracy



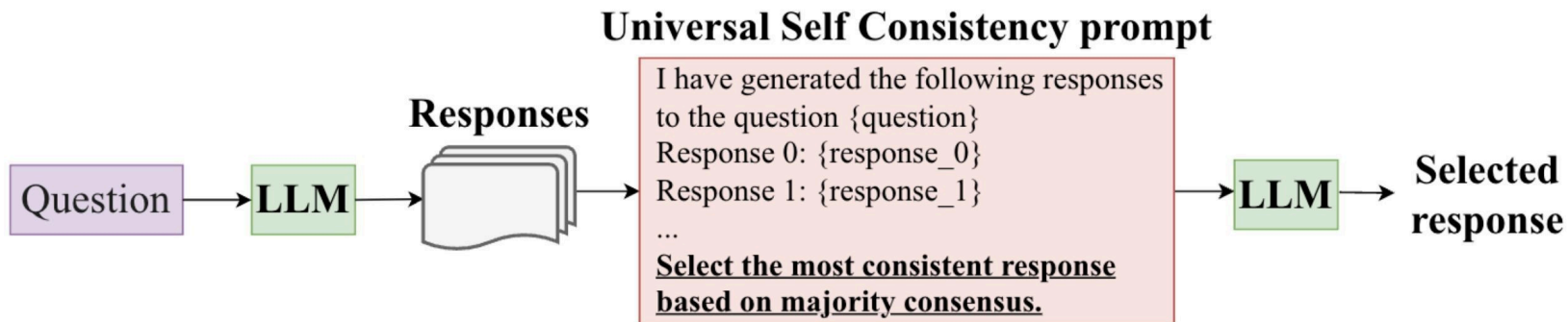
More agreement across sampled responses → LLM is more certain → aggregated solution is more likely correct.

# Limitation of self-consistency decoding



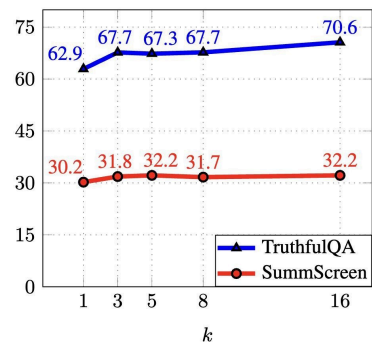
Requires an answer extraction process — can we enable consistency-based decoding for free-form generation?

## Universal self-consistency: LLM performs consistency-based selection

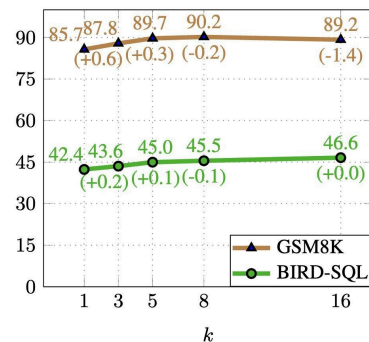


Instead of extracting answers, ask the LLM itself to pick the most consistent response — works for open-ended generation.

## Universal Self-Consistency results



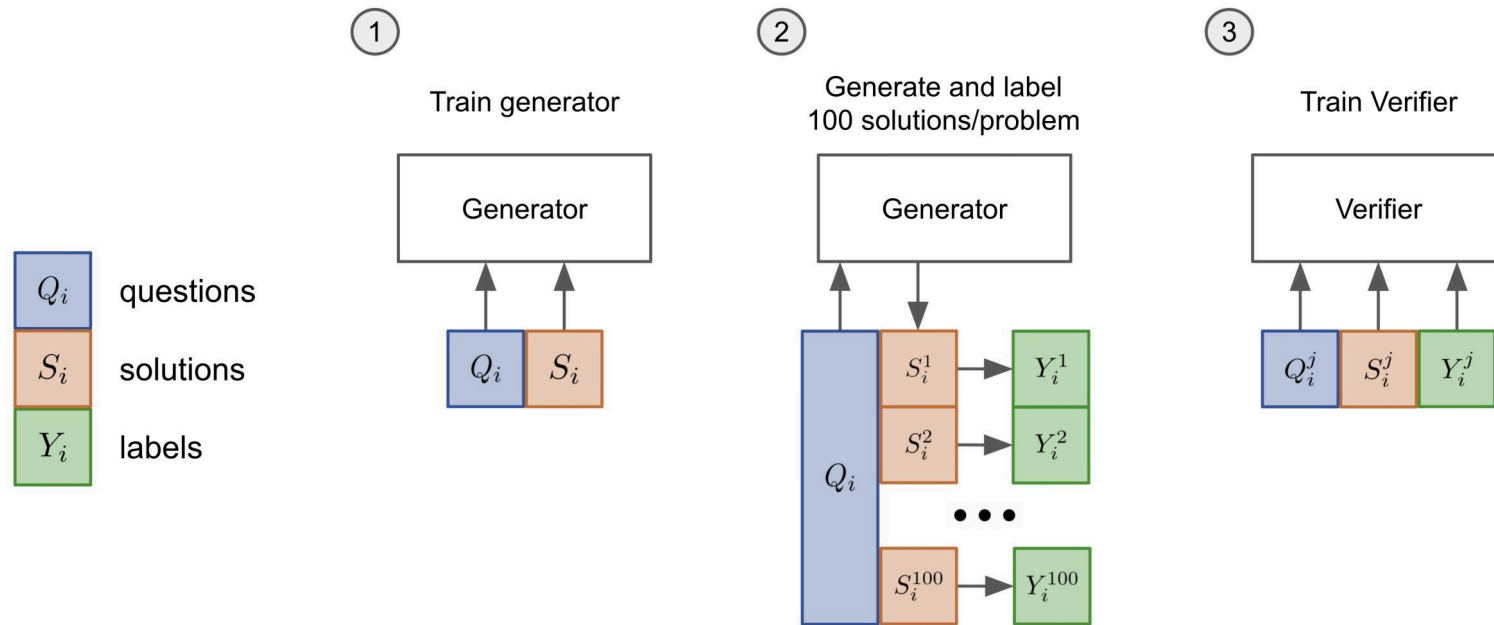
(a) Results on open-ended generation.



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

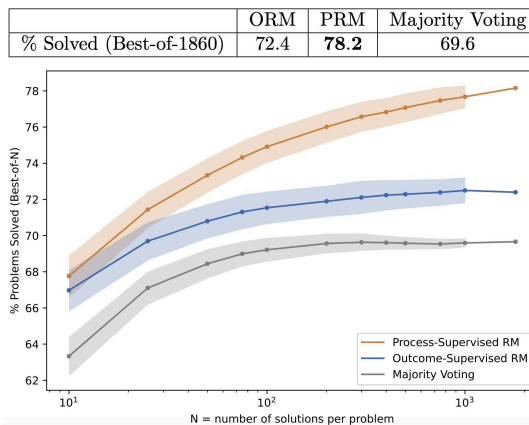
- **USC** improves open-ended generation (summarization, QA) where vanilla SC doesn't apply.
- Matches SC on math & coding. No answer extraction, no code execution. Bounded by long-context capability.

# Training LLM rankers: two types of verifiers



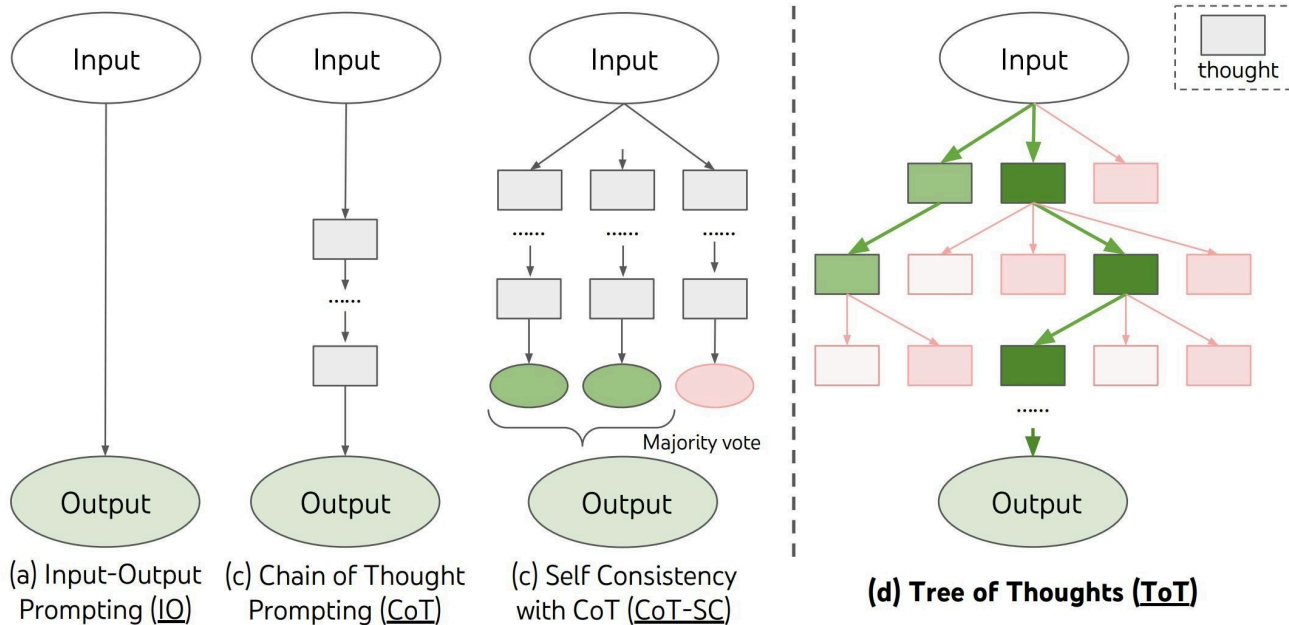
**ORM** (Outcome RM): verify at the solution level. **PRM** (Process RM): verify at the step level.

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



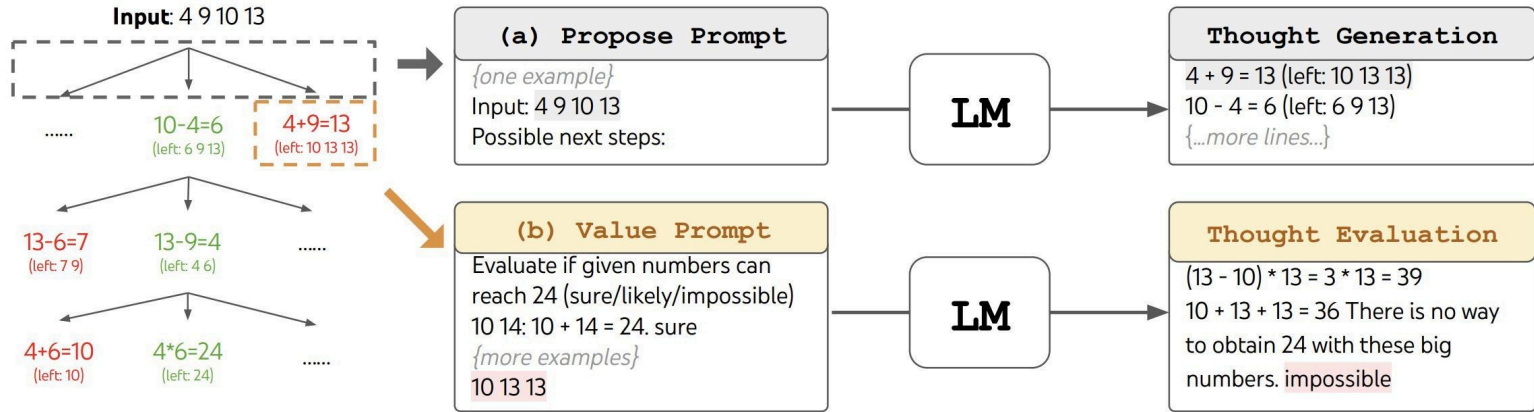
- **PRM** scales better with more samples than ORM or consistency voting.
- Highly dependent on verifier quality — the same verifier may not generalize across tasks.

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



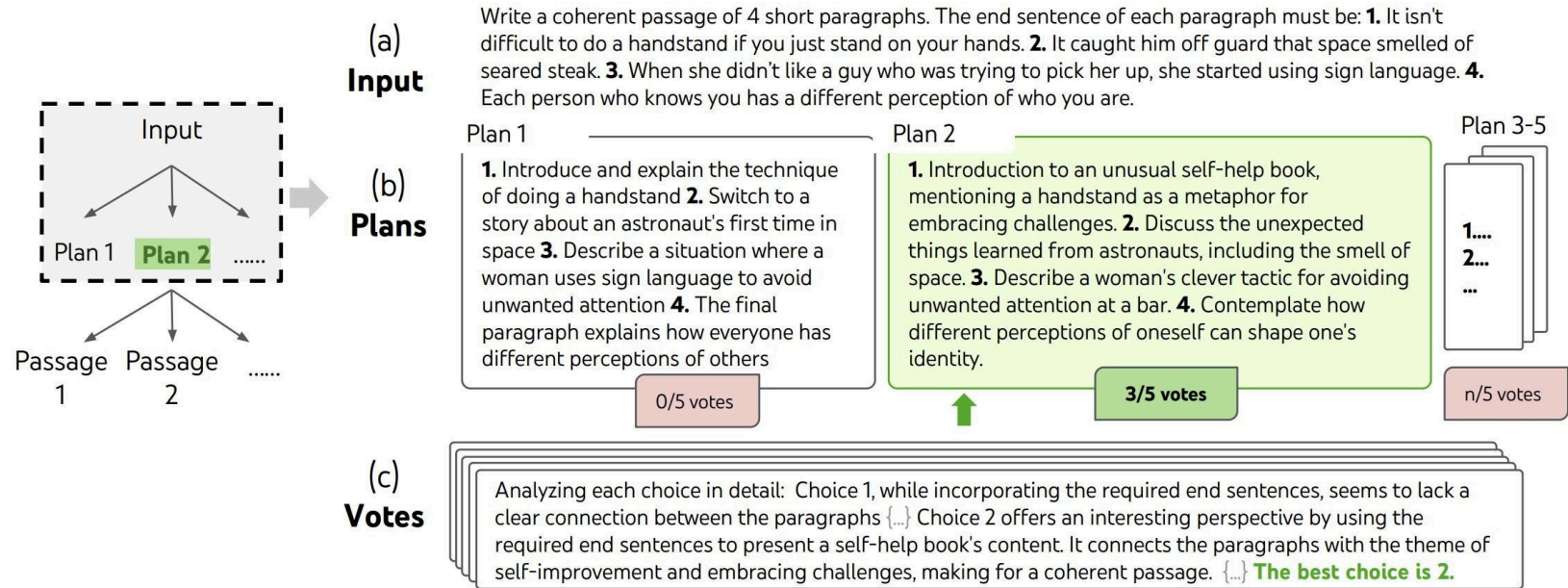
Does **not fully utilize** a step-wise scorer — LLM + tree search prioritizes promising partial solutions.

# Tree-of-thought example: Game of 24



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

# Voting-based state evaluation

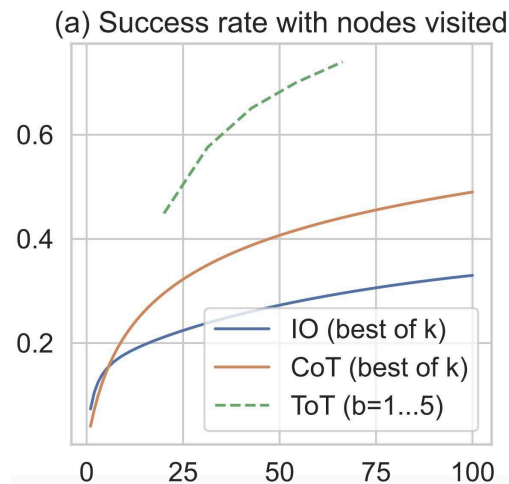


LLM selects the best state among candidates by voting multiple times; majority vote becomes the final choice.

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

- ToT + BFS scales better than standard prompting and CoT w.r.t. token budget.

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>



**Can we reshape the model's output distribution?**

**So that thoughtful responses naturally rank 1st?**

# Supervised Finetuning (SFT)

## STEP 1

Collect problems and their step-by-step solutions from human annotators.

## STEP 2

Maximize the likelihood of human solutions.

## APPLY

Then apply the model everywhere.

## Supervised Finetuning (SFT) — in practice

- **Training data:** hand-annotated problem – rationale – answer triples.
- Example A — last letter of "artificial intelligence": rationale walks through each word's last letter → "le".
- Example B — Elsa has 3 apples... rationale walks through addition → 8.
- **Finetune** the LLM on those pairs, then apply to a test problem (e.g. "How many 'r's in 'strawberry'?").

## Pros and Cons of SFT

### PROS

- Generic.

### CONS

- Does not generalize well.
- Scaling does not help much.



# How to fix the generalization failure from SFT?

Recall the SFT procedure — what should we change?

**STEP 1**

Collect problems and their step-by-step solutions from human annotators.

**STEP 2**

Maximize the likelihood of human solutions.

# RL Finetuning

Repeat this process:

STEP 1

Collect problems and their step-by-step solutions **generated from the model**.

STEP 2

Maximize the likelihood of correct solutions.

# RL Finetuning — full form

Repeat this process:

STEP 1

Collect problems and their step-by-step solutions **generated from the model**.

STEP 2

Maximize the likelihood of correct solutions, **minimize the likelihood of wrong solutions**.

Why “generated from the model” instead of “from humans”?

First principle in ML:

**Directly optimize what we want!**

## So what do we want to optimize here?

- **Optimize “generation quality”!**
- Examples of quality measurement: correctness for solving math problems; BLEU score for machine translation; ...
- All the rest is to compute gradients / backpropagation.

So what do we want to optimize here?

RL finetuning

**= Optimize it by gradient ascent**

$$\max_{\theta} \mathbb{E}[r(\text{response}|\text{problem}, \theta)]$$

$r$  measuring the quality of a response

$\theta$  model parameters

## Summary — Part 2

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