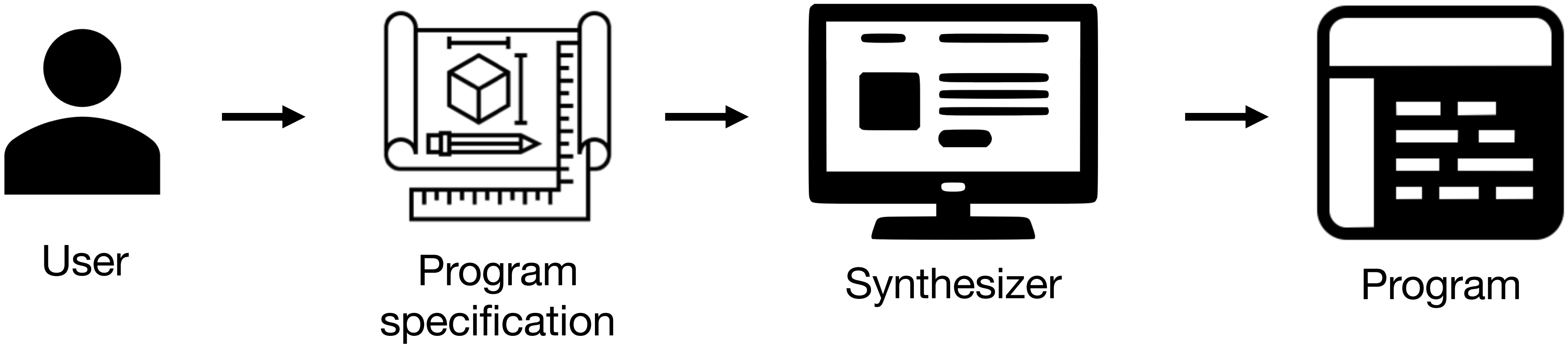
Large Language Models for Code Generation

Xinyun Chen

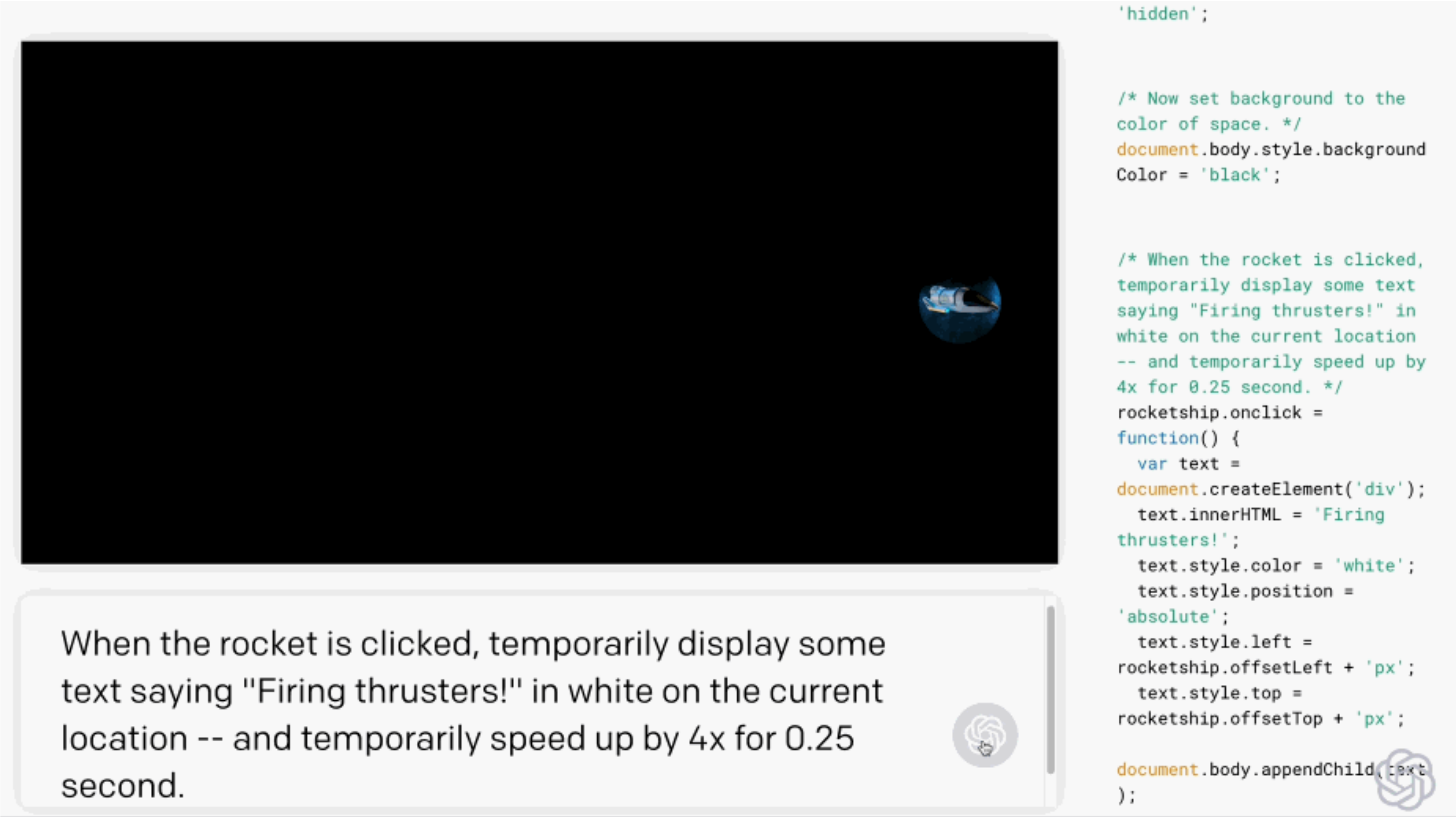
Google DeepMind

xinyunchen@google.com

Code generation



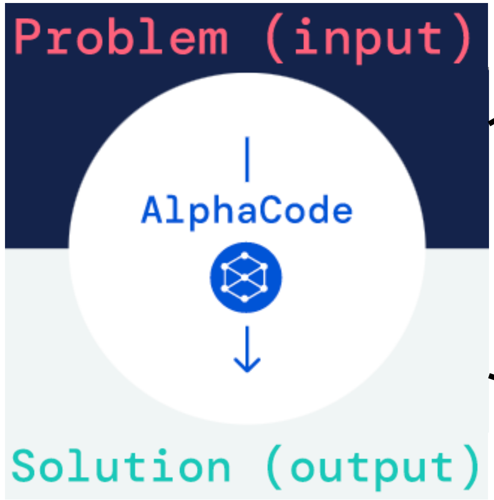
Success of LLMs for text-to-code generation

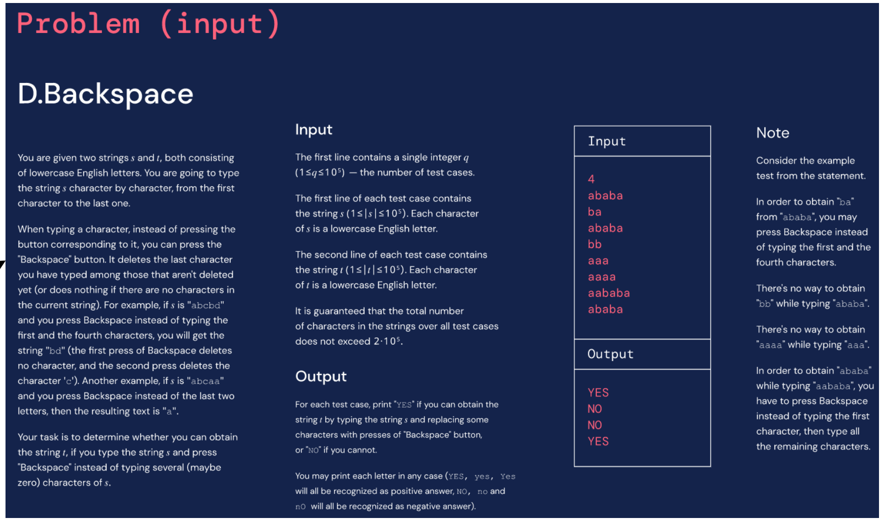


OpenAI Codex

Chen et al., Evaluating Large Language Models Trained on Code.

Code generation from natural language + input-output examples





Problem (input)

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 "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (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 ($1 \le q \le 10^5$) — the number of test cases.

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

The second line of each test case contains the string t ($1 \le |t| \le 10^5$). 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

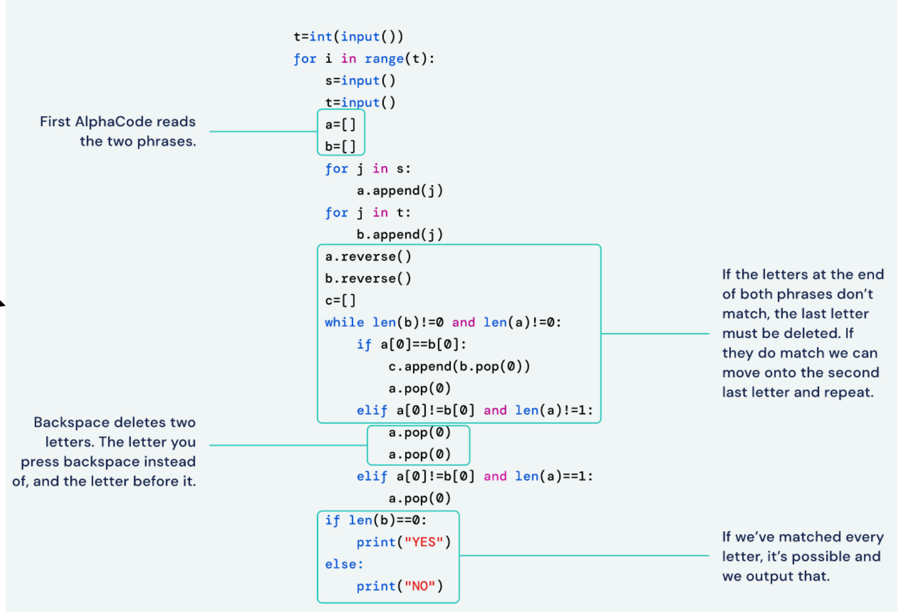
Consider the example test from the statement.

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.



```
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
        if a[0]==b[0]:
            c.append(b.pop(0))
            a.pop(0)
        elif a[0]!=b[0] and len(a)!=1:
            a.pop(0)
            a.pop(0)
        elif a[0]!=b[0] and len(a)==1:
            a.pop(0)
    if len(b)==0:
        print("YES")
    else:
        print("NO")
```


DeepMind AlphaCode: competitive programming

Li, …, **Chen** et al., Competition-level Code Generation with AlphaCode, Science 2022.

Code generation for tool use

S

Find the alkali metal with the highest resistivity

Used **Wolfram**

Among the alkali metals, cesium (Cs) has the highest resistivity. Here is the Wolfram Language code I used to find this result:

wolfram | Copy code

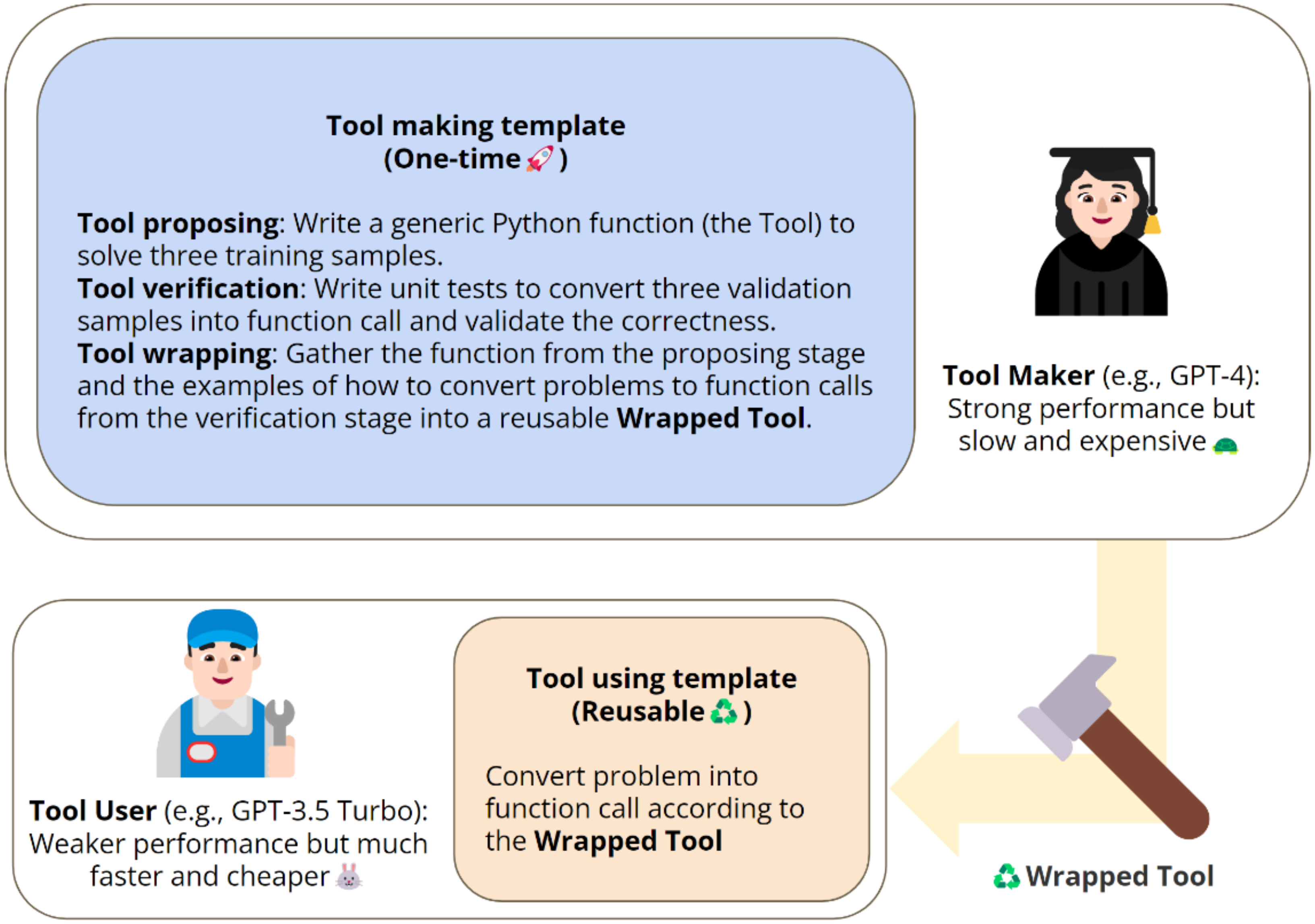
```
alkaliMetals = {Entity["Element", "Lithium"], Entity["Element", "Sodium"], E
resistivities = EntityValue[alkaliMetals, "Resistivity"];
maxResistivity = Max[resistivities];
position = Position[resistivities, maxResistivity];
alkaliMetals[[position[[1, 1]]]]
```

This code defines a list of alkali metal entities (excluding francium due to its instability and lack of data), retrieves their resistivity values, finds the maximum resistivity, and then identifies the alkali metal with the highest resistivity.

ChatGPT generates code for calling APIs

https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/

Code generation for tool making



Cai, Wang, Ma, **Chen**, Zhou, Large Language Models as Tool Makers.

Challenge 1: algorithmic reasoning

Exam	GPT-4
Uniform Bar Exam (MBE+MEE+MPT)	298 / 400 (~90th)
LSAT	163 (~88th)
SAT Evidence-Based Reading & Writing	710 / 800 (~93rd)
SAT Math	700 / 800 (~89th)
Graduate Record Examination (GRE) Quantitative	163 / 170 (~80th)
Graduate Record Examination (GRE) Verbal	169 / 170 (~99th)
Graduate Record Examination (GRE) Writing	4 / 6 (~54th)
USABO Semifinal Exam 2020	87 / 150 (99th - 100th)
USNCO Local Section Exam 2022	36 / 60
Medical Knowledge Self-Assessment Program	75 %
Codeforces Rating	392 (below 5th)

OpenAI, GPT-4 Technical Report.

Challenge 2: predicted code in the first round can be problematic

GPT-4

XI

Write a function that gives loss amount if the given amount has loss else return None. Your Python function should pass the following assertions: assert loss_amount(1500, 1200)=None, assert loss_amount(100, 200)=-100, assert loss_amount(2000, 5000)=-3000

To determine the loss amount, you need to subtract the current value from the original value. If the result is negative, it means there was a loss. If the result is positive or zero, return `None`.

Here's the Python function that follows this logic:

python Copy code

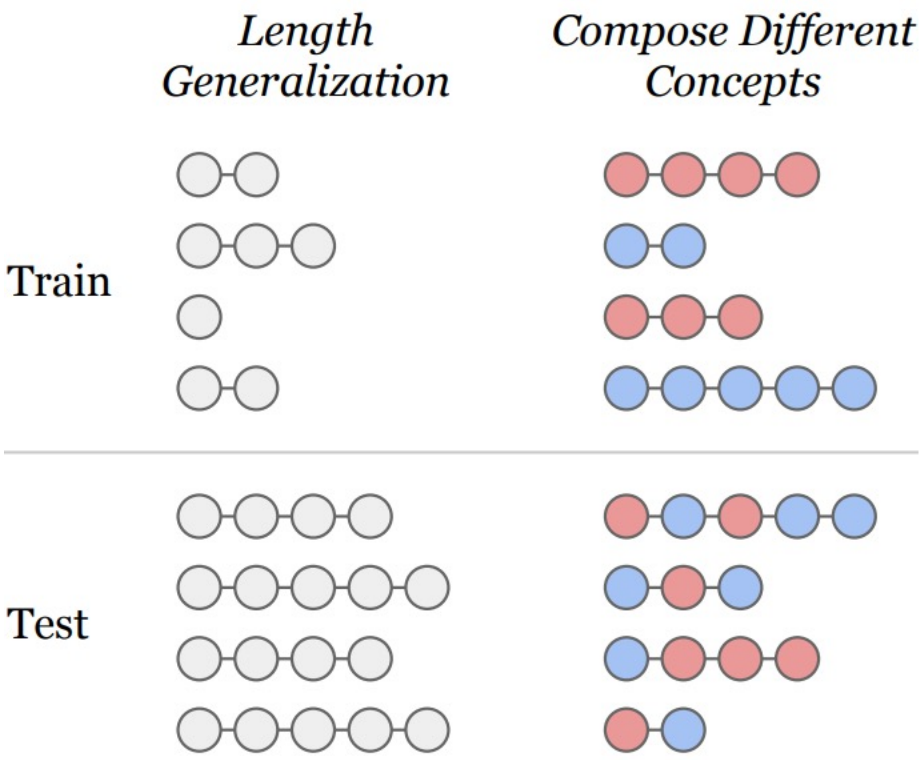
```python
def loss_amount(original, current):
    difference = current - original
    if difference < 0:
        return difference
    else:
        return None

# Tests
assert loss_amount(1500, 1200) == None
assert loss_amount(100, 200) == -100
assert loss_amount(2000, 5000) == -3000
```

This function should pass the provided assertions.

- LLMs might fail to solve simple coding problems specified with input-output examples
- Weakness in code execution understanding

Challenge 3: compositional generalization



Method	Standard prompting	Chain-of-Thought
code-davinci-002	16.7	16.2
text-davinci-002	6.0	0.0
code-davinci-001	0.4	0.0

SCAN length split

- Compositional generalization: given basic components and a few demonstrations of their combinations, generalize to novel combinations.
- Length generalization: generalize to longer test samples than the training ones.
- LLMs fail to achieve length generalization without proper prompt design

Shi et al., ExeDec: Execution Decomposition for Compositional Generalization in Neural Program Synthesis.
Zhou et al., Least-to-most Prompting Enables Complex Reasoning in Large Language Models, ICLR 2023.

Outline

- Background: how large language models change the code generation paradigm
- AlphaCode: large language models for competition-level code generation
- Self-debugging: teach large language models to debug their own predicted code
- Dynamic least-to-most prompting: enable compositional generalization for code generation

Li, …, **Chen** et al., Competition-level Code Generation with AlphaCode, Science 2022.
Chen, Lin, Schärli, Zhou, Teaching Large Language Models to Self-Debug.
Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.

Part 1: Background on code generation

- Learning-based code generation before large language models
  - Specialized models for different applications and programming languages

- Code generation with LLMs
  - One model for multiple coding applications
  - Enable quick adaptations to new tasks via prompting

Code generation: transition with learning-based techniques

Input v_1	*Input* v_2	*Output*
Albania	*355*	*case 355: return "Albania";*
Algeria	*213*	*case 213: return "Algeria";*

String Program:

$$\mathtt{Concatenate}(\mathtt{ConstStr}(\text{"}\mathit{case}\ \text{"}), v_2, \mathtt{ConstStr}(\text{": }\mathit{return}\ \text{""}), v_1, \mathtt{ConstStr}(\text{"";"}))$$

FlashFill:

string processing in Microsoft Excel

[Gulwani 2011; Polozov et al., 2015]

Code generation: transition with learning-based techniques

What states border the state that borders the most states

$$\lambda x.state(x) \wedge borders(x, \arg\max(\lambda y.state(y), \lambda y.count(\lambda z.state(z) \wedge borders(y, z))))$$

Learning probabilistic grammars

[Zettlemoyer et al., 2012; Liang et al., 2013]

Input v_1	*Input v_2*	*Output*
Albania	*355*	*case 355: return "Albania";*
Algeria	*213*	*case 213: return "Algeria";*

String Program:

$$\mathtt{Concatenate}(\mathtt{ConstStr}(\text{"case "}), v_2, \mathtt{ConstStr}(\text{": return ""}), v_1, \mathtt{ConstStr}(\text{"";"}))$$

FlashFill:

string processing in Microsoft Excel

[Gulwani 2011; Polozov et al., 2015]

Code generation: transition with learning-based techniques

What states border the state that borders the most states

$$\lambda x.state(x) \wedge borders(x, \arg\max(\lambda y.state(y), \lambda y.count(\lambda z.state(z) \wedge borders(y, z))))$$

Learning probabilistic grammars

[Zettlemoyer et al., 2012; Liang et al., 2013]

Input v_1	Input v_2	Output
Albania	355	case 355: return "Albania";
Algeria	213	case 213: return "Algeria";

String Program:

$$Concatenate(ConstStr(\text{"case "}), v_2, ConstStr(\text{": return ""}), v_1, ConstStr(\text{"";"}))$$

FlashFill:

string processing in Microsoft Excel

[Gulwani 2011; Polozov et al., 2015]

Support **more complex** languages

[Zhong et al., 2017; Ying et al., 2017; Yu et al., 2018]

Which countries in Europe have at least 3 car manufacturers?

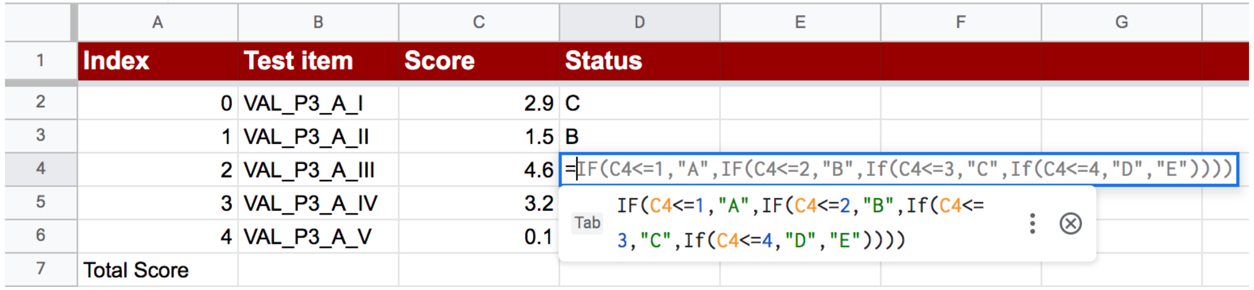
```
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

Code generation: transition with learning-based techniques

What states border the state that borders the most states

$\lambda x.state(x) \wedge borders(x, \arg\max(\lambda y.state(y),$
$\lambda y.count(\lambda z.state(z) \wedge borders(y, z))))$

Learning probabilistic grammars
[Zettlemoyer et al., 2012; Liang et al., 2013]



	A	B	C	D	E	F	G
1	Index	Test item	Score	Status			
2	0	VAL_P3_A_I	2.9	C			
3	1	VAL_P3_A_II	1.5	B			
4	2	VAL_P3_A_III	4.6	=IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))			
5	3	VAL_P3_A_IV	3.2				
6	4	VAL_P3_A_V	0.1				
7	Total Score						

SpreadsheetCoder:
formula prediction from **ambiguous** context
[**Chen** et al., 2021]

Support **more complex** languages
[Zhong et al., 2017; Ying et al., 2017; Yu et al., 2018]

Input v_1	*Input v_2*	*Output*
Albania	*355*	*case 355: return "Albania";*
Algeria	*213*	*case 213: return "Algeria";*

String Program:
$\mathtt{Concatenate}(\mathtt{ConstStr}(\text{"case "}), v_2, \mathtt{ConstStr}(\text{": return ""}), v_1, \mathtt{ConstStr}(\text{"";"}))$

FlashFill:
string processing in Microsoft Excel
[Gulwani 2011; Polozov et al., 2015]

Which countries in Europe have at least 3 car manufacturers?

```sql
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

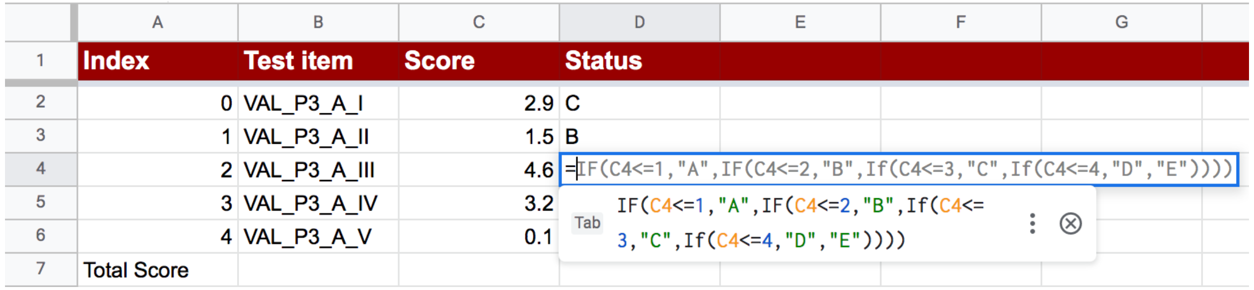
Code generation: transition with learning-based techniques

What states border the state that borders the most states

$$\lambda x.state(x) \wedge borders(x, \arg\max(\lambda y.state(y), \lambda y.count(\lambda z.state(z) \wedge borders(y, z))))$$

Learning probabilistic grammars

[Zettlemoyer et al., 2012; Liang et al., 2013]



	A	B	C	D	E	F	G
1	Index	Test item	Score	Status			
2	0	VAL_P3_A_I	2.9	C			
3	1	VAL_P3_A_II	1.5	B			
4	2	VAL_P3_A_III	4.6	=IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))			
5	3	VAL_P3_A_IV	3.2				
6	4	VAL_P3_A_V	0.1				
7	Total Score						

SpreadsheetCoder:

formula prediction from **ambiguous** context

[**Chen** et al., 2021]

Support **more complex** languages

[Zhong et al., 2017; Ying et al., 2017; Yu et al., 2018]

Input v_1	*Input v_2*	*Output*
Albania	*355*	*case 355: return "Albania";*
Algeria	*213*	*case 213: return "Algeria";*

String Program:

$Concatenate(ConstStr(\text{"case "}), v_2, ConstStr(\text{": return ""}), v_1, ConstStr(\text{"";"}))$

FlashFill:

string processing in Microsoft Excel

[Gulwani 2011; Polozov et al., 2015]

Which countries in Europe have at least 3 car manufacturers?

```
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

Learning-based techniques support more **complex** and **ambiguous** program specifications.

Code generation: transition with large language models

	A	B	C	D	E	F	G
1	Index	Test item	Score	Status			
2	0	VAL_P3_A_I	2.9	C			
3	1	VAL_P3_A_II	1.5	B			
4	2	VAL_P3_A_III	4.6	=IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))			
5	3	VAL_P3_A_IV	3.2				
6	4	VAL_P3_A_V	0.1				
7	Total Score						

Tab IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))

SpreadsheetCoder:
formula prediction from **ambiguous** context
[**Chen** et al., 2021]

CODEGEN

StarCoder

...

Support **more complex** languages
[Zhong et al., 2017; Ying et al., 2017; Yu et al., 2018]

Which countries in Europe have at least 3 car manufacturers?

```
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

GPT

Bard

Code Llama

Code generation: transition with large language models

	A	B	C	D	E	F	G
1	Index	Test item	Score	Status			
2	0	VAL_P3_A_I	2.9	C			
3	1	VAL_P3_A_II	1.5	B			
4	2	VAL_P3_A_III	4.6	=IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))			
5	3	VAL_P3_A_IV	3.2				
6	4	VAL_P3_A_V	0.1				
7	Total Score						

Tab IF(C4<=1,"A",IF(C4<=2,"B",IF(C4<=3,"C",IF(C4<=4,"D","E"))))

SpreadsheetCoder:
formula prediction from **ambiguous** context
[**Chen** et al., 2021]

Support **more complex** languages
[Zhong et al., 2017; Ying et al., 2017; Yu et al., 2018]

Which countries in Europe have at least 3 car manufacturers?

```
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

CODEGEN

StarCoder

...

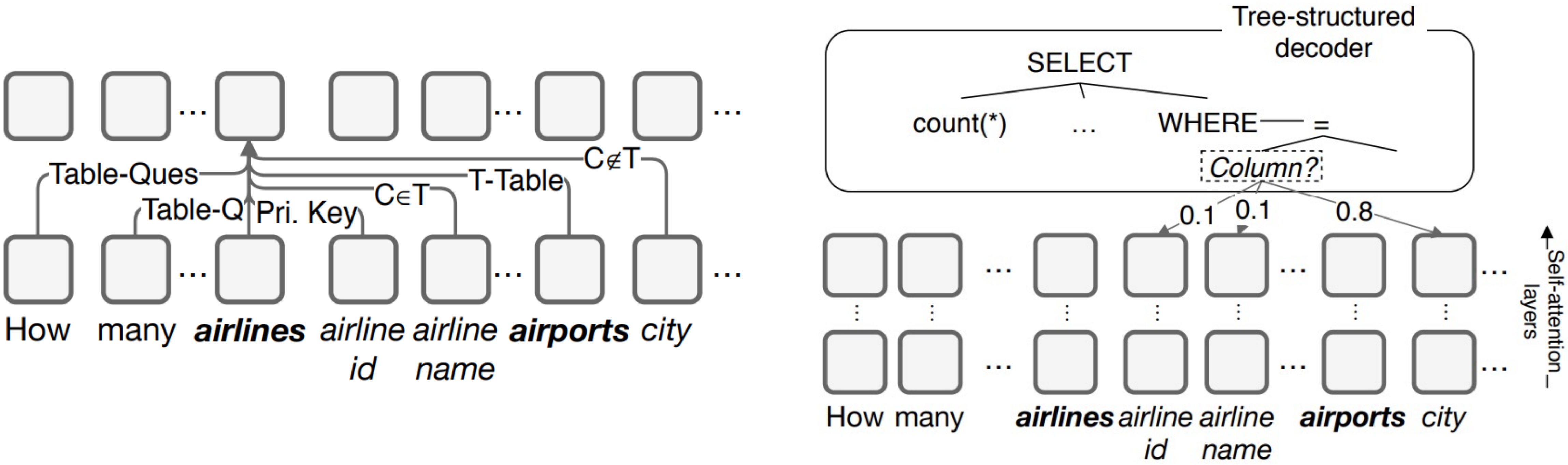
GPT

Bard

Code Llama

Large language models enable more **diverse** code generation applications with **free-form** natural language specification.

Before LLMs: specialized model architectures to represent code structures

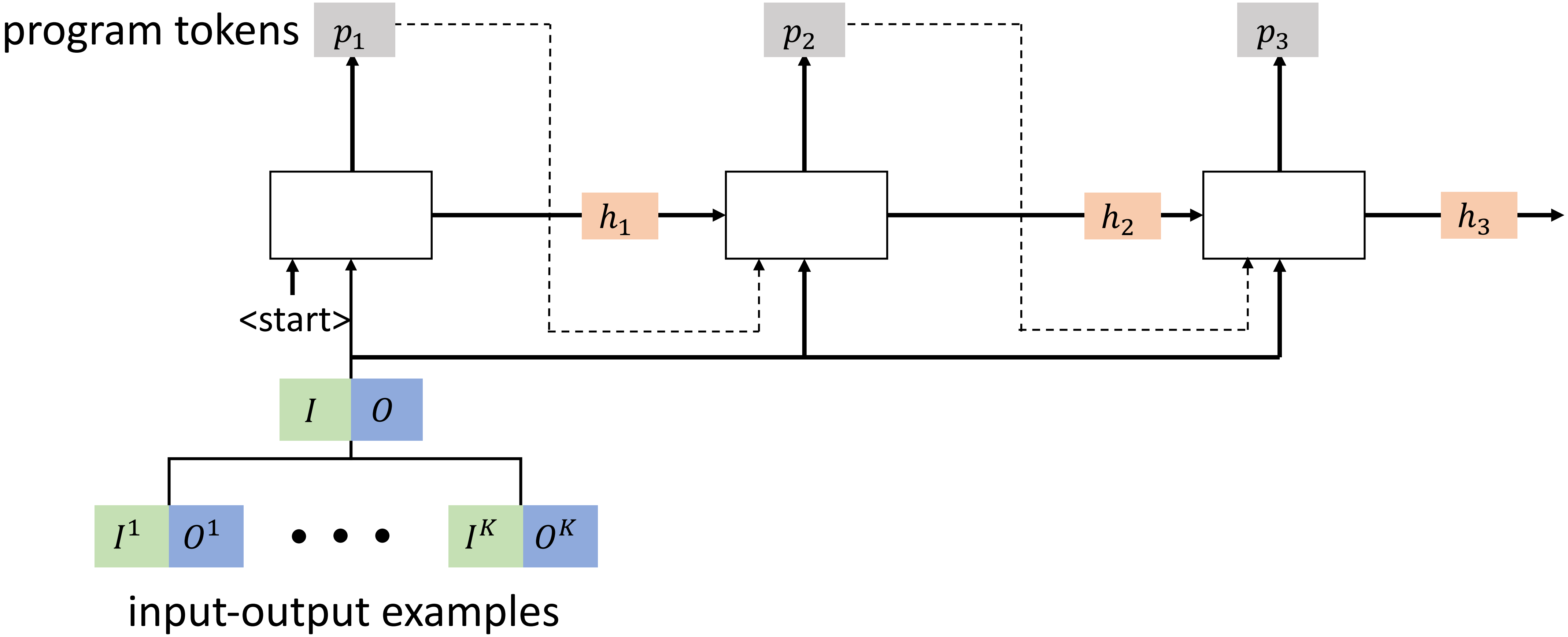


RAT-SQL for text-to-SQL generation

- Encoder: self-attention is biased towards tokens with relations in database schemas
- Decoder: tree-structured decoding based on the SQL grammar

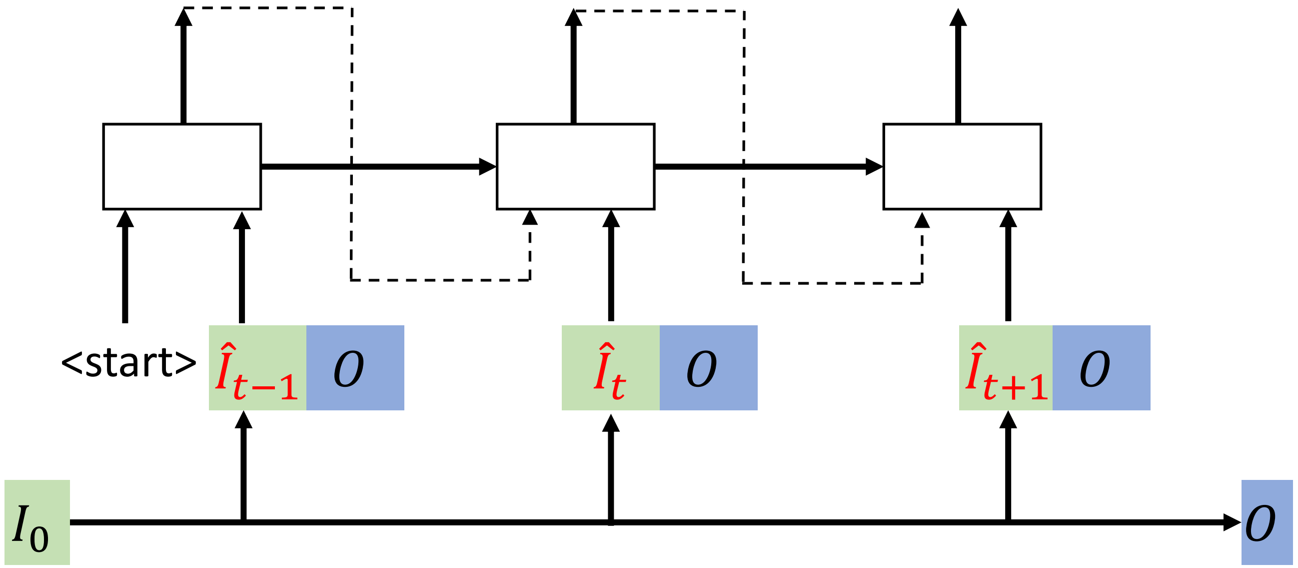
Wang*, Shin* et al., RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers, ACL 2020.

Before LLMs: encoder-decoder architecture for programming by examples



Chen, Liu, Song, Execution-Guided Neural Program Synthesis, ICLR 2019.
Chen, Song, Tian, Latent Execution for Neural Program Synthesis, NeurIPS 2021.

Before LLMs: decoding schemes to utilize the code execution



$\hat{I}_t$: (latent) execution state at step t

(I_0, O): input-output example

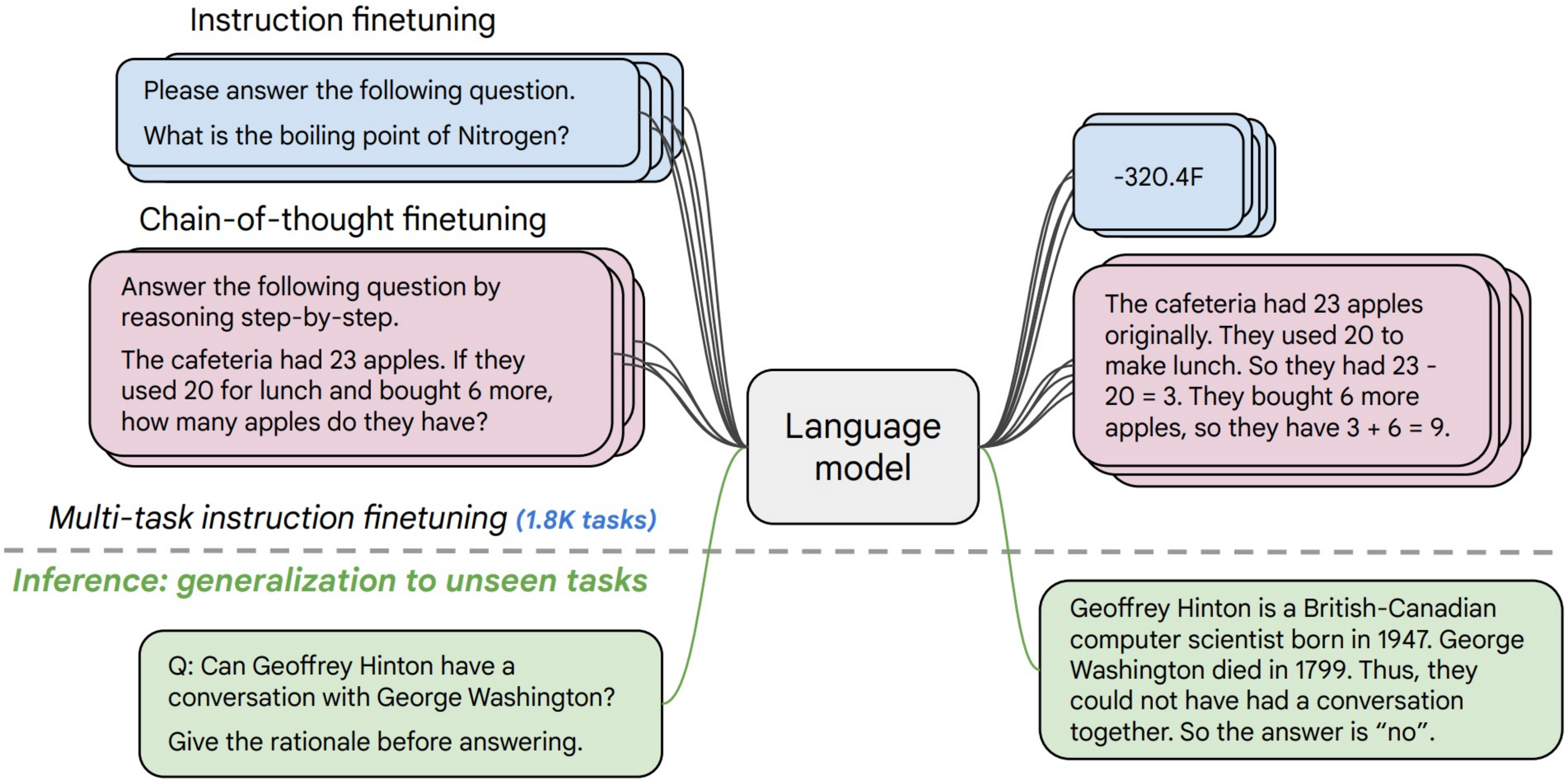
Chen, Liu, Song, Execution-Guided Neural Program Synthesis, ICLR 2019.
Chen, Song, Tian, Latent Execution for Neural Program Synthesis, NeurIPS 2021.

After LLMs: programming languages as foreign languages?

- State-of-the-art language models treat a programming language as another natural language
  - Large-scale training + large-scale off-the-shelf Transformer-based architecture → high-performance generative model for both text and code

- The same training recipe is applied to both text and code
  - Large-scale pretraining
  - Instruction tuning

Instruction tuning: multi-task learning generalizes to unseen tasks



- Scaling up the model size and number of training tasks improves the performance.
- Training with chain-of-thought data triggers rationale generation by default.

Chung, Hou, Longpre, …, **Chen**,…, Wei, Scaling Instruction-Finetuned Language Models.

Bard demo: data processing

https://blog.google/technology/ai/code-with-bard/

Bard demo: code explanation

https://blog.google/technology/ai/code-with-bard/

Bard demo: code debugging

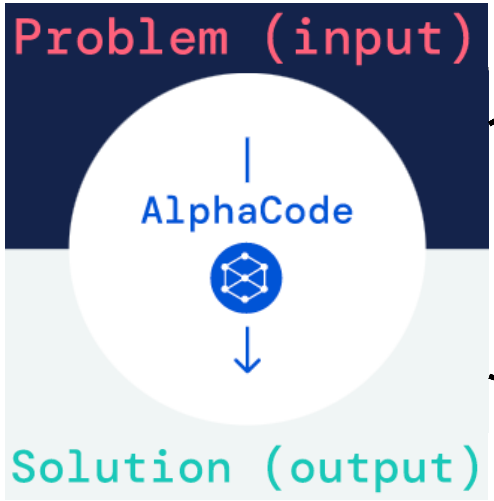
https://blog.google/technology/ai/code-with-bard/

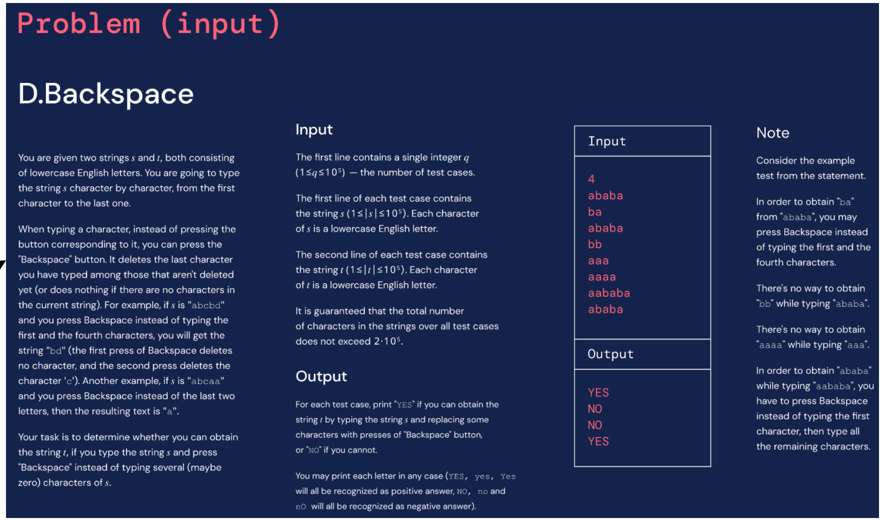
Outline

- Background: how large language models change the code generation paradigm
  - In the era of LLMs, what do we learn from techniques for developing specialized code generation models?
- AlphaCode: large language models for competition-level code generation
  - Leverage execution for code reranking
- Self-debugging: teach large language models to debug their own predicted code
  - Leverage execution to improve the sample efficiency
- Dynamic least-to-most prompting: enable compositional generalization for code generation
  - Leverage syntax structures for problem decomposition

Li, …, **Chen** et al., Competition-level Code Generation with AlphaCode, Science 2022.
Chen, Lin, Schärli, Zhou, Teaching Large Language Models to Self-Debug.
Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.

Part 2: LLMs for competition-level code generation




Problem (input)

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 "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (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 $(1 \le q \le 10^5)$ — the number of test cases.

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

The second line of each test case contains the string t $(1 \le |t| \le 10^5)$. 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

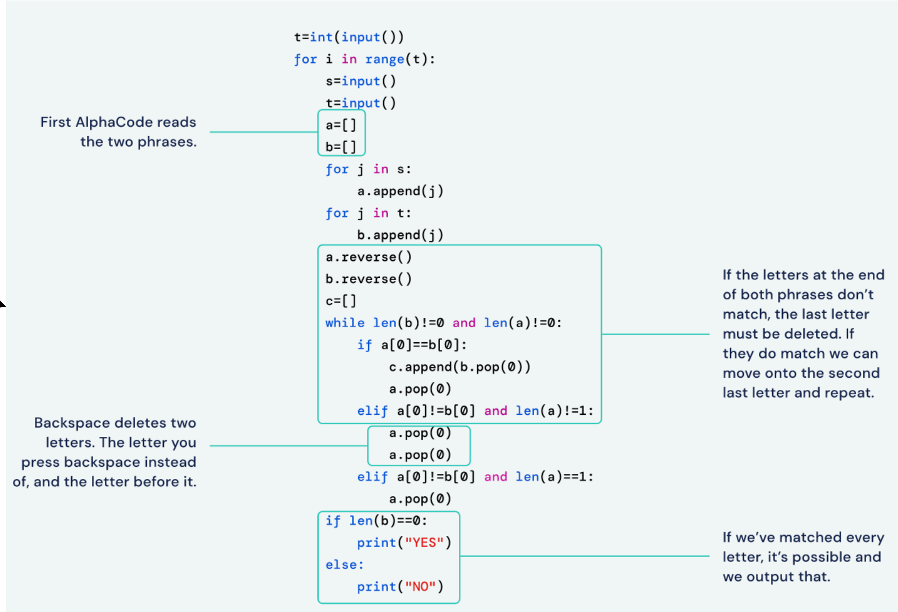
Consider the example test from the statement.

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".

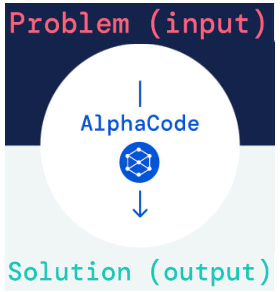
In order to obtain "ababa" while typing "aababa", you have to press Backspace instead of typing the first character, then type all the remaining characters.



```
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
        if a[0]==b[0]:
            c.append(b.pop(0))
            a.pop(0)
        elif a[0]!=b[0] and len(a)!=1:
            a.pop(0)
            a.pop(0)
        elif a[0]!=b[0] and len(a)==1:
            a.pop(0)
    if len(b)==0:
        print("YES")
    else:
        print("NO")
```


Li, …, **Chen** et al., Competition-level Code Generation with AlphaCode, Science 2022.

Competitive programming: input format



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 "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (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 $(1 \le q \le 10^5)$ — the number of test cases.

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

The second line of each test case contains the string t $(1 \le |t| \le 10^5)$. 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.

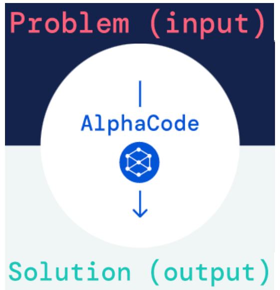
There's no way to obtain "bb" while typing "ababa".

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.

Long and complicated text description + input-output specification

Competitive programming: sample decoded program




```python
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
        if a[0]==b[0]:
            c.append(b.pop(0))
            a.pop(0)
        elif a[0]!=b[0] and len(a)!=1:
            a.pop(0)
            a.pop(0)
        elif a[0]!=b[0] and len(a)==1:
            a.pop(0)
    if len(b)==0:
        print("YES")
    else:
        print("NO")
```

First AlphaCode reads the two phrases.

If the letters at the end of both phrases don't match, the last letter must be deleted. If they do match we can move onto the second last letter and repeat.

Backspace deletes two letters. The letter you press backspace instead of, and the letter before it.

If we've matched every letter, it's possible and we output that.

No direct mapping between the problem description and output code

Challenges

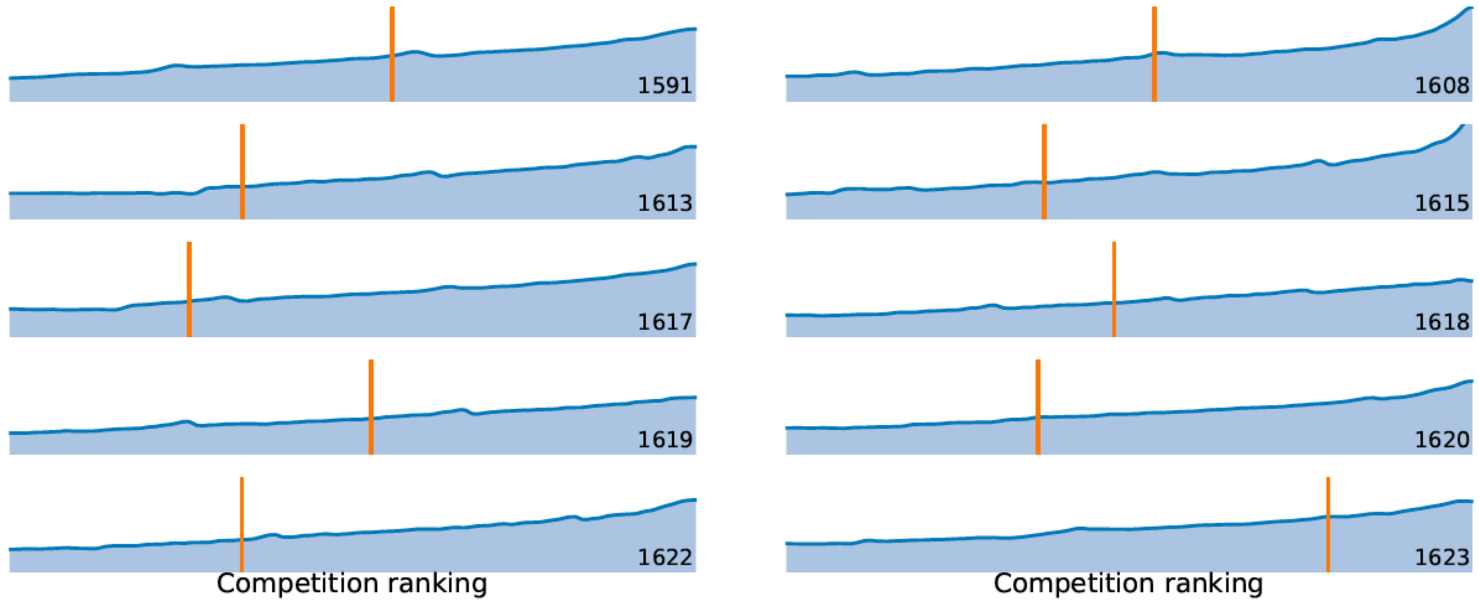
- The model needs to not only understand the intended execution behavior specified in the long and complicated problem description, but also come up with an efficient implementation.
- The given input-output examples are just a few simplest illustrative examples. The predicted code also needs to pass many more hidden test cases, often more complex and larger-scale.
- New competitive programming problems are very different from existing problems on a surface level.

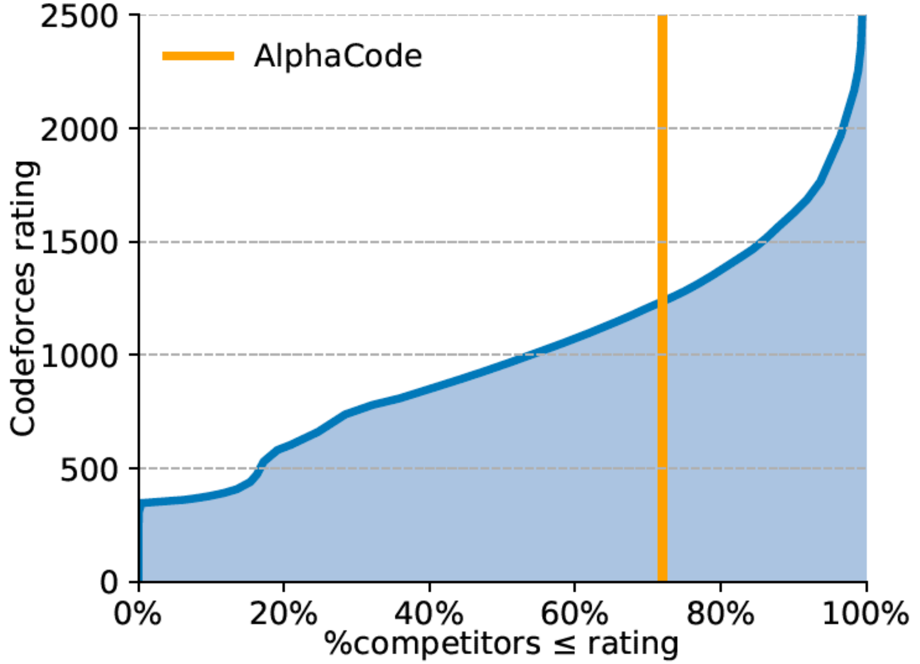
Where we are now?

Evaluation on Codeforces platform, 10 competitions with >5k participants per contest

- An ensemble of 41B and 9B models
- 10 submissions per problem

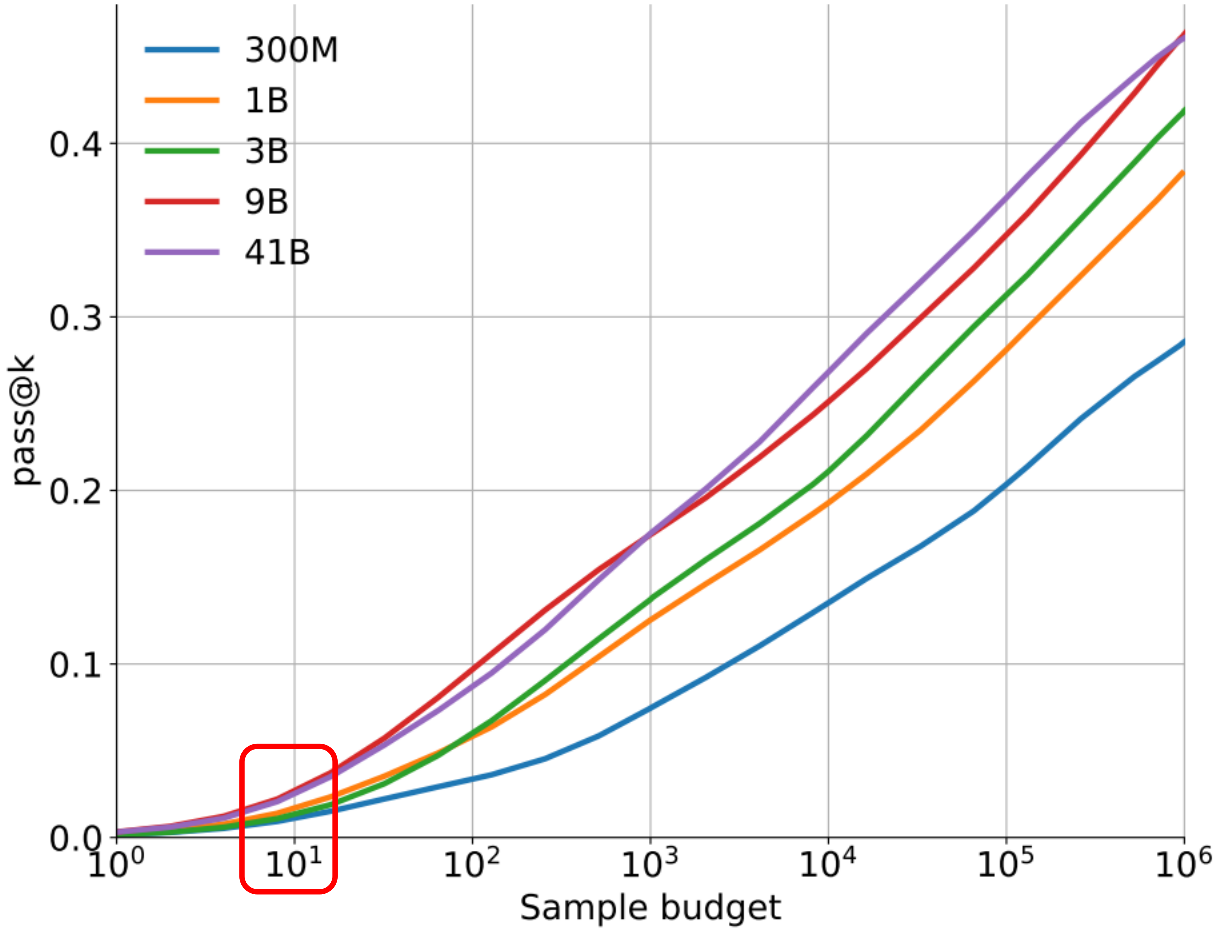
Right is better





- Average ranking across 10 contests: top 54.3%
- Codeforces rating ranking among all Codeforces users: top 28%

Would scaling up the model solve the problem?



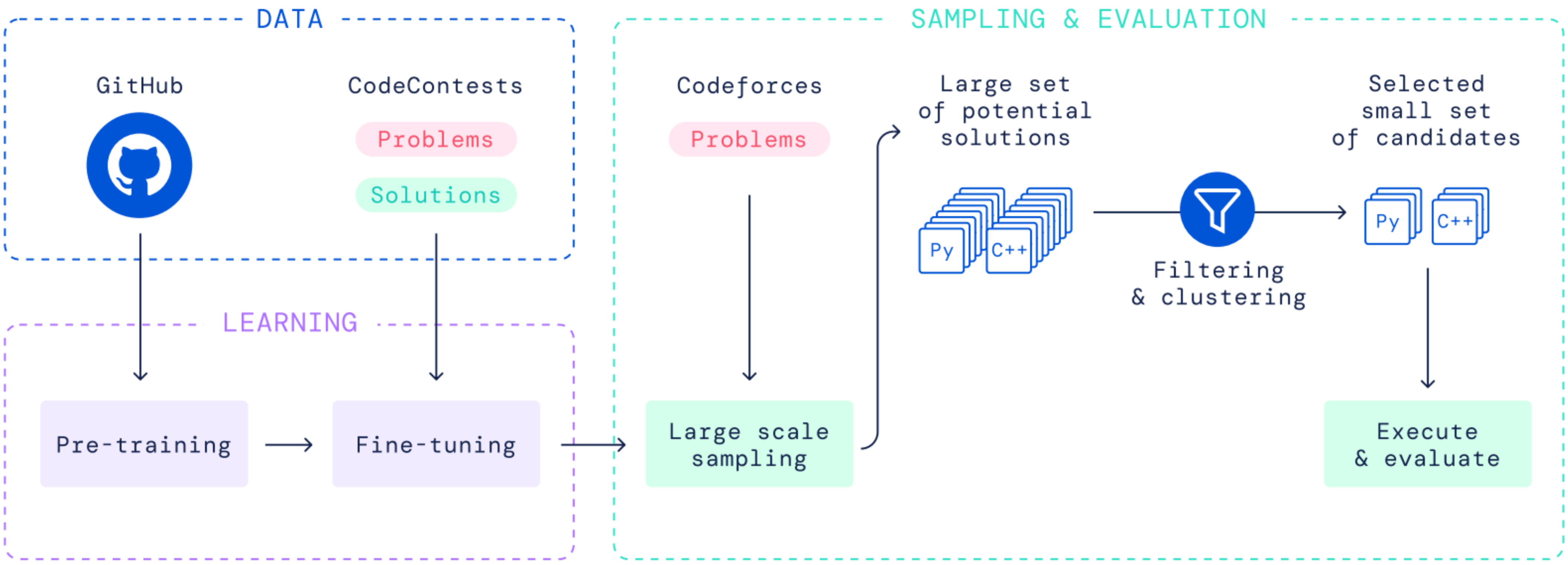
- Performance gain becomes less significant with larger models
- No model achieves decent performance with few samples
- 10 final submissions come from much more samples

GPT-4 still fails on Codeforces problems

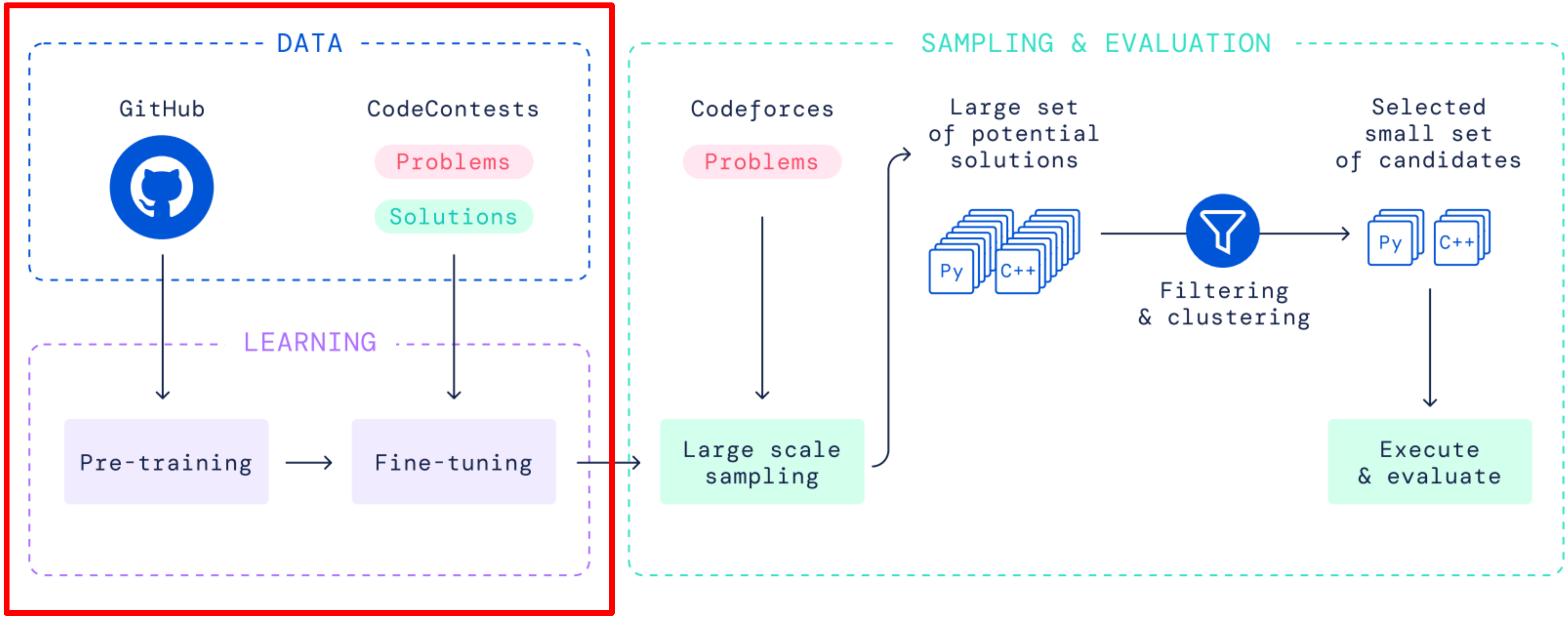
Exam	GPT-4
Uniform Bar Exam (MBE+MEE+MPT)	298 / 400 (~90th)
LSAT	163 (~88th)
SAT Evidence-Based Reading & Writing	710 / 800 (~93rd)
SAT Math	700 / 800 (~89th)
Graduate Record Examination (GRE) Quantitative	163 / 170 (~80th)
Graduate Record Examination (GRE) Verbal	169 / 170 (~99th)
Graduate Record Examination (GRE) Writing	4 / 6 (~54th)
USABO Semifinal Exam 2020	87 / 150 (99th - 100th)
USNCO Local Section Exam 2022	36 / 60
Medical Knowledge Self-Assessment Program	75 %
Codeforces Rating	392 (below 5th)

OpenAI, GPT-4 Technical Report.

AlphaCode overview



AlphaCode overview



Temporal split for training and test data construction

- Pretraining: 715.1GB code from GitHub before 2021/07/14
  - Multiple programming languages: C++, Python, Java, JavaScript, C#, etc.
- Finetuning with CodeContests, containing problem-solution pairs
  - 13,328 competitive programming problems before 2021/07/14
  - Human solutions in C++, Python, and Java
  - ~900 solutions per problem, including correct and incorrect ones (~50% each)
- Validation set: 117 Codeforces problems in 2021/07/14-2021/09/20
- Test set: 165 Codeforces problems after 2021/09/20

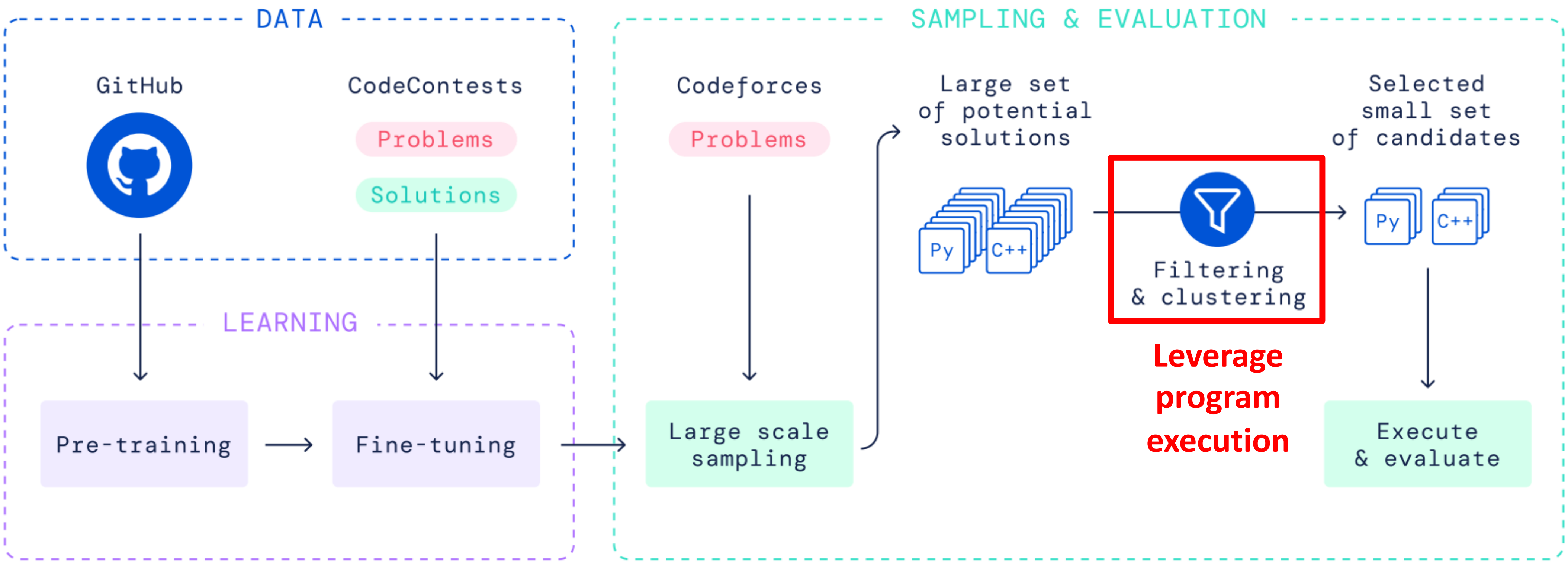
Training with value prediction & conditioning

```
RATING: 1200
TAGS: dp,implementation
LANGUAGE IS python3
CORRECT SOLUTION
Polycarp must pay exactly n burles at the checkout ... (rest of the description)
```

Metadata (provided in the training set)

- Training
  - Metada: include rating (difficulty), tags (solution type) and language (Python3/C++)
  - **Value conditioning**: train on both correct and incorrect solutions
  - **Value prediction**: add an auxiliary loss to predict the solution correctness
- Evaluation
  - Always condition on "CORRECT SOLUTION"
  - **Randomly sample** rating and tags, sample both Python3 and C++ programs
    - Improve the **diversity** of samples

AlphaCode overview



Filtering via execution on example tests

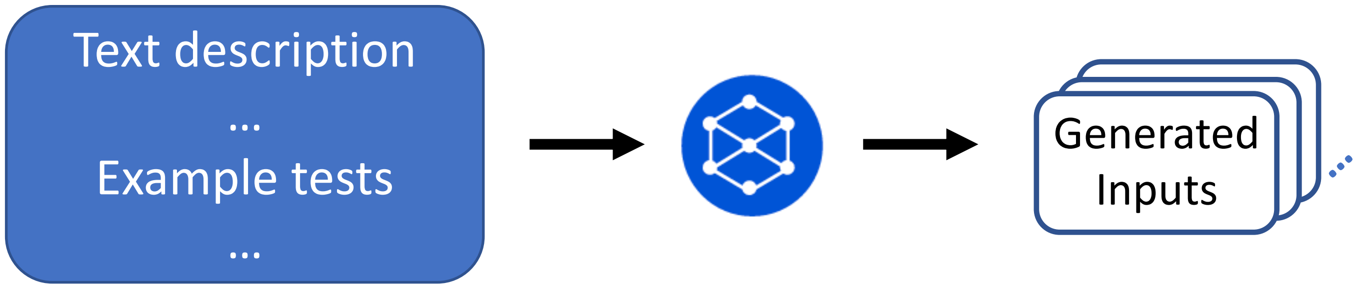
- Execute all programs on example tests in the problem description, filter out those samples that do not pass the tests
  - Note: the solution is correct when passing both example tests and hidden tests (not included in the problem description)

- **>99%** of samples do not pass example tests (1-2 example tests per problem)

Model	% Problems with ≥ 1 samples pass example tests	Average $p_{\text{pass example tests}}$ on all problems	Average $p_{\text{pass example tests}}$ on solved problems
300M	82.05%	0.39%	1.18%
1B	87.18%	0.59%	1.40%
3B	87.18%	0.49%	0.98%
9B	89.74%	0.76%	1.52%
41B	92.31%	0.73%	1.47%

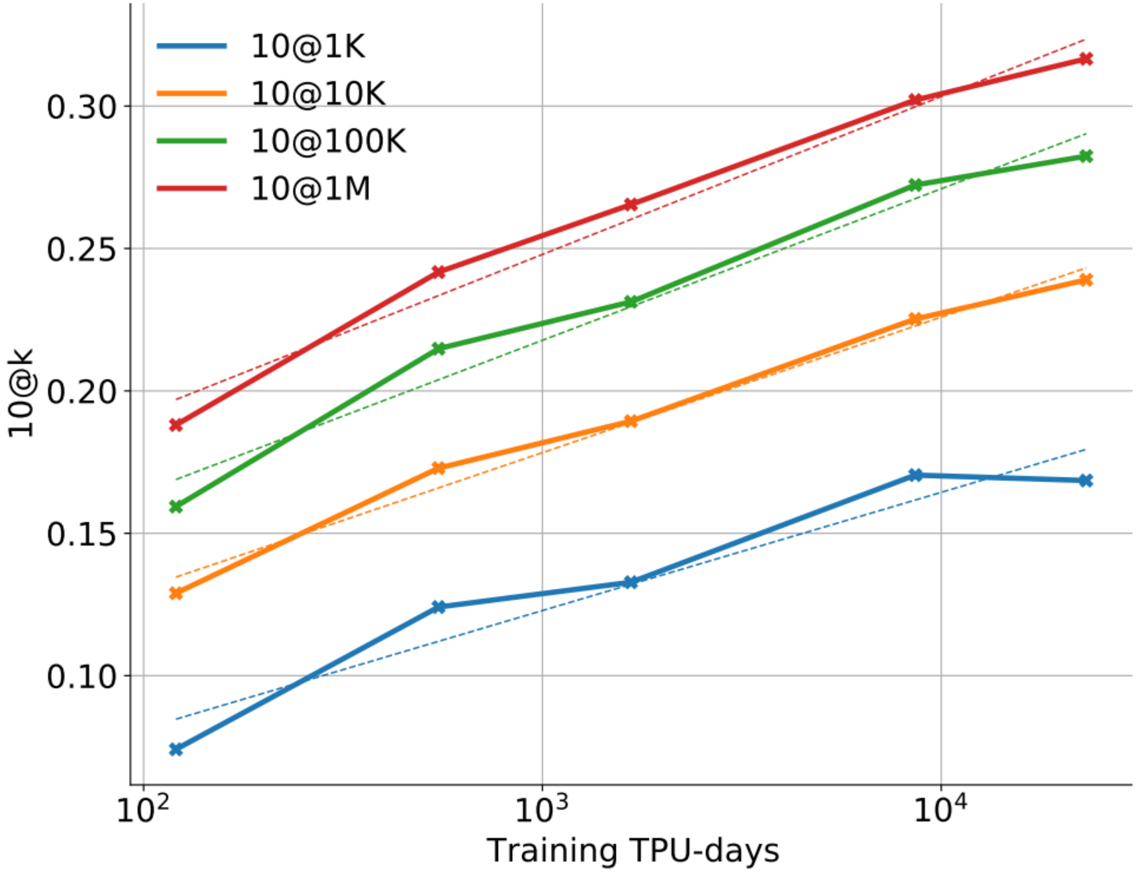
Clustering by execution on generated inputs

- Recall: hidden tests for test problems are not available
- Train a separate model to generate new test inputs

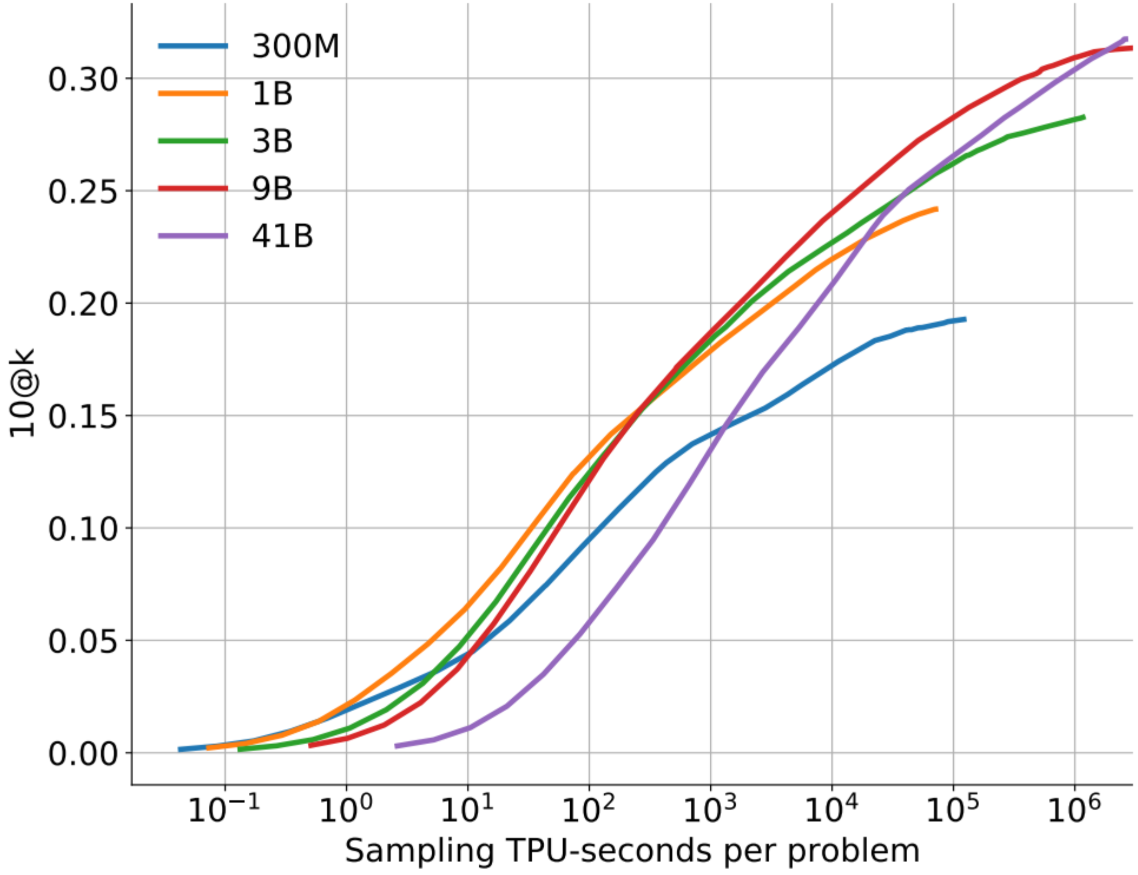


- Execute sampled programs on all generated inputs
- Cluster all programs with the same outputs together
- Sample 1 program from each of the 10 largest clusters

Solve rate scales log-linearly with more compute & model size

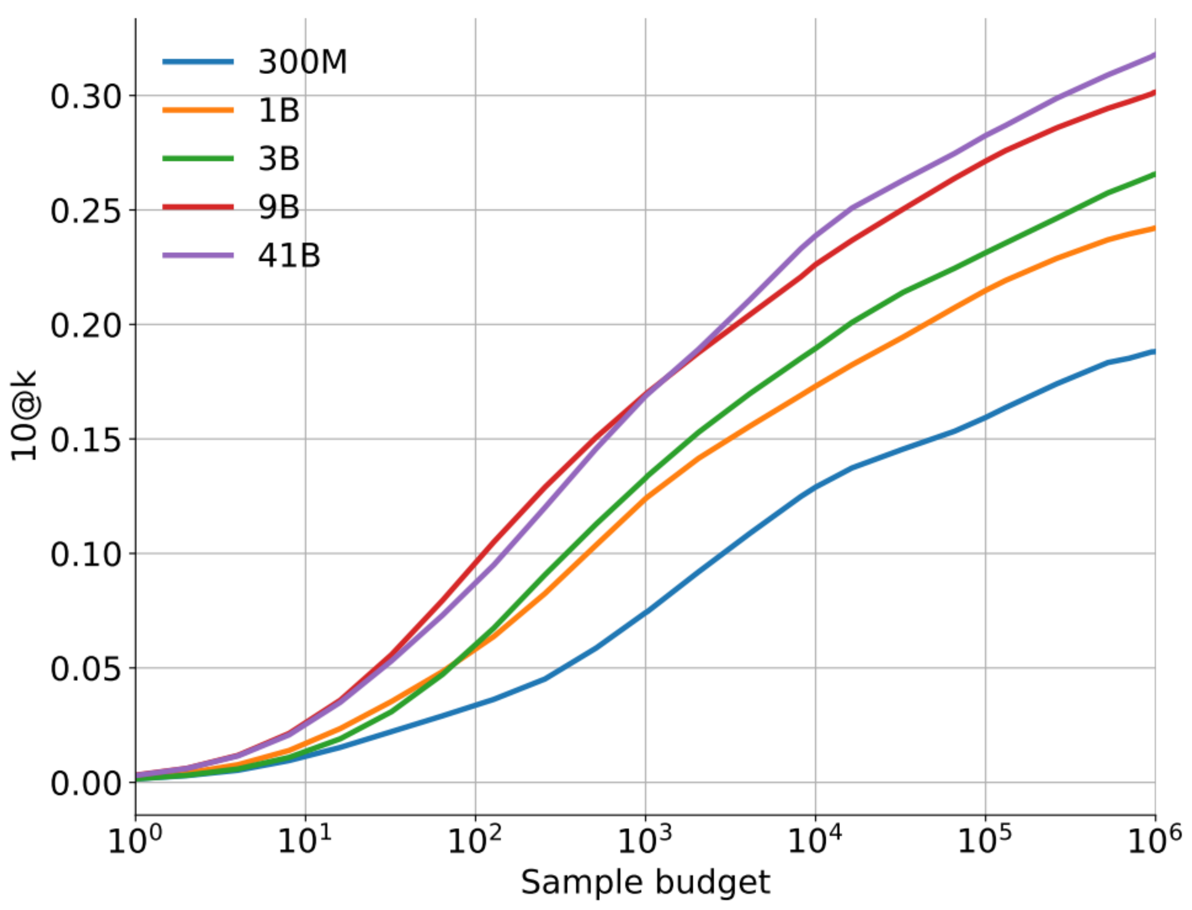


(a) Solve Rate vs. Training Compute

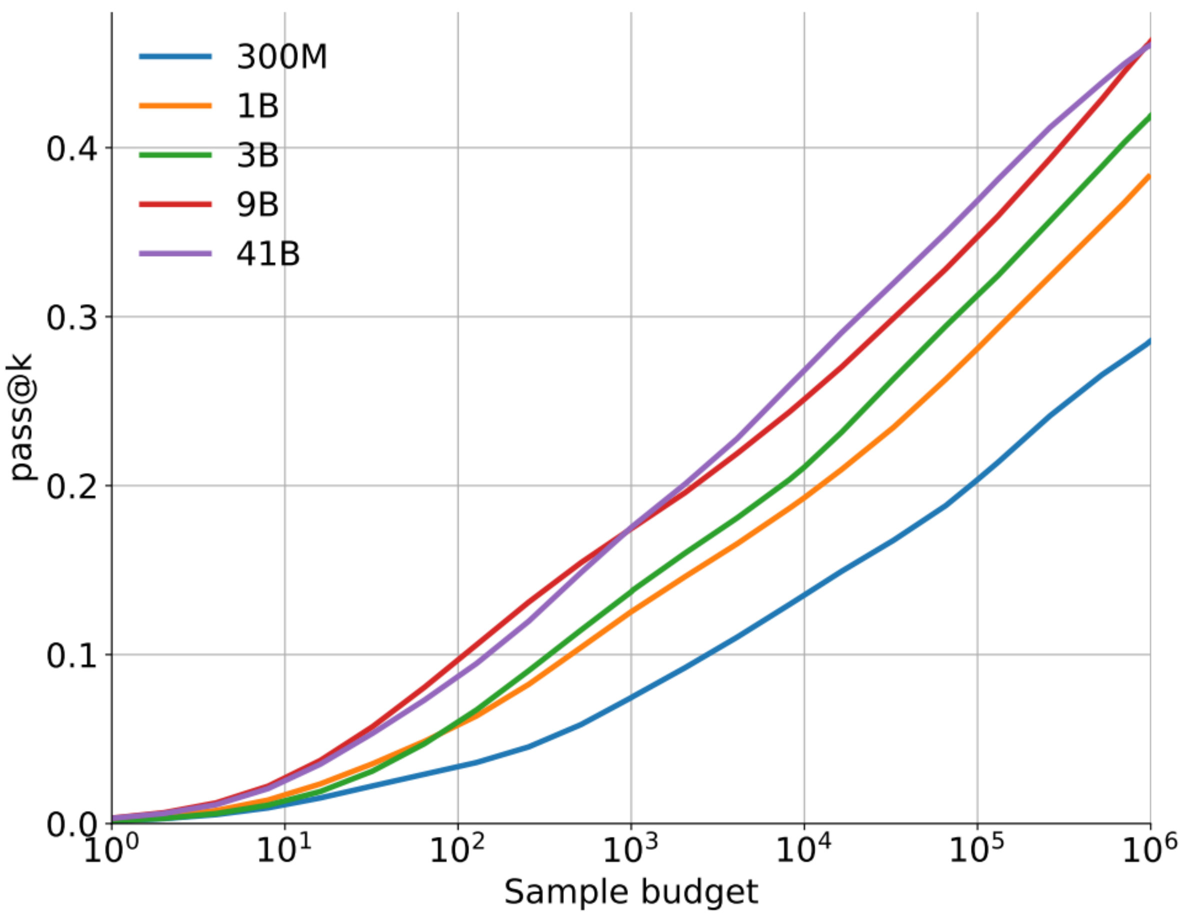


(b) Solve Rate vs. Sampling Compute

Solve rate scales log-linearly with more samples



(a) 10 attempts per problem



(b) Unlimited attempts per problem

Ablation studies: pretraining

Pre-training dataset	Solve rate		
	10@1K	10@10K	10@100K
No pre-training	4.5%	7.0%	10.5%
GitHub (Python only)	5.8%	11.1%	15.5%
MassiveText	9.7%	16.1%	21.2%
GitHub (all languages)	12.4%	17.3%	21.5%

1B encoder-decoder model

- MassiveText: an English text corpus with 3% GitHub code
- Pretraining with multiple programming languages achieves the best performance

Rae et al., Scaling Language Models: Methods, Analysis & Insights from Training Gopher.

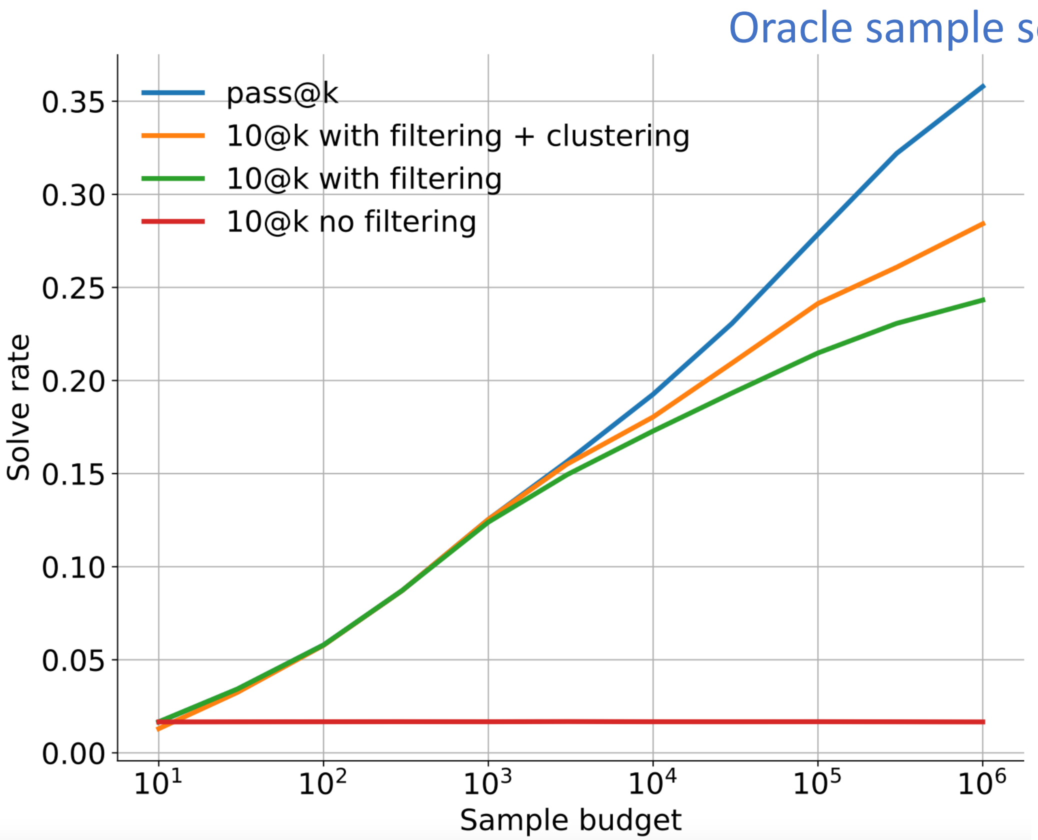
Ablation studies: finetuning

Fine-tuning setting	Solve rate 10@1K	10@10K	10@100K	10@1M
No Enhancements	6.7% (6.5-6.8)	10.4% (9.6-11.0)	15.2% (14.3-15.9)	19.6% (18.2-20.4)
+ MLM	6.6% (6.2-7.0)	12.5% (12.1-12.7)	17.0% (16.5-17.2)	20.7% (19.1-21.3)
+ Tempering	7.7% (7.2-8.5)	13.3% (12.5-13.8)	18.7% (18.0-19.2)	21.9% (20.7-22.6)
+ Tags and Ratings	6.8% (6.4-7.0)	13.7% (12.8-14.9)	19.3% (18.1-20.0)	22.4% (21.3-23.0)
+ Value	10.6% (9.8-11.1)	16.6% (16.4-16.9)	20.2% (19.6-20.7)	23.2% (21.7-23.9)
+ GOLD	12.4% (12.0-13.0)	17.3% (16.9-17.6)	21.5% (20.5-22.2)	24.2% (23.1-24.4)
+ Clustering	12.2% (10.8-13.4)	18.0% (17.3-18.8)	24.1% (23.2-25.0)	28.4% (27.5-29.3)

1B encoder-decoder model

- Value conditioning and prediction significantly improves the performance, especially with fewer samples
- Clustering via execution selects better samples for final evaluation

Ablation studies: filtering and clustering



1B encoder-decoder model

- Filtering with execution is crucial
- Clustering with generated inputs can provide more gain than scaling up models
- Still a gap from the oracle sample selection

Approach	Validation Set				Test Set		
	10@1k	10@10k	10@100k	10@1M	10@1k	10@10k	10@100k
9B	16.9%	22.6%	27.1%	30.1%	14.3%	21.5%	25.8%
41B	16.9%	23.9%	28.2%	31.8%	15.6%	23.2%	27.7%
41B + clustering	21.0%	26.2%	31.8%	34.2%	16.4%	25.4%	29.6%

Breakdown on different algorithms

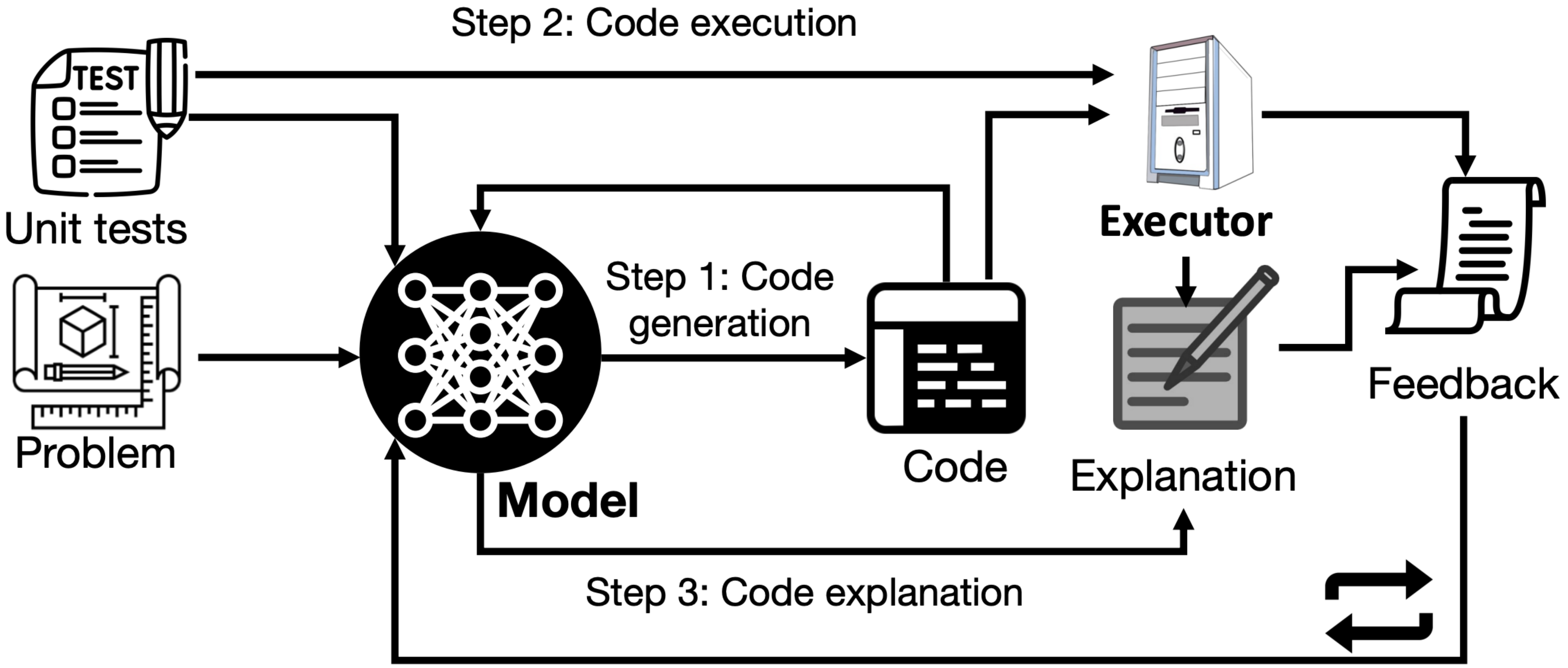
Model	Greedy	Math	DP	Constructive Algorithms	Brute Force	Data Structures	Implem-entation	Graphs	Bitmasks	Sortings
300M	13.1%	19.3%	4.5%	7.5%	9.8%	8.8%	5.0%	0.2%	22.2%	16.9%
1B	19.7%	22.7%	4.5%	9.1%	12.0%	10.5%	14.1%	5.9%	26.8%	21.5%
3B	19.9%	22.7%	4.9%	11.2%	13.2%	11.9%	13.4%	8.8%	25.4%	23.8%
9B	23.7%	29.4%	7.1%	13.8%	19.5%	16.9%	16.4%	16.6%	27.4%	27.8%
41B	25.0%	28.2%	8.8%	14.9%	20.4%	15.7%	16.5%	13.6%	33.8%	25.5%

- Solve rate of 10@10k on most popular problems types
- Dynamic programming and constructive algorithms are particularly challenging, even if there are a lot of related training problems

Takeaway

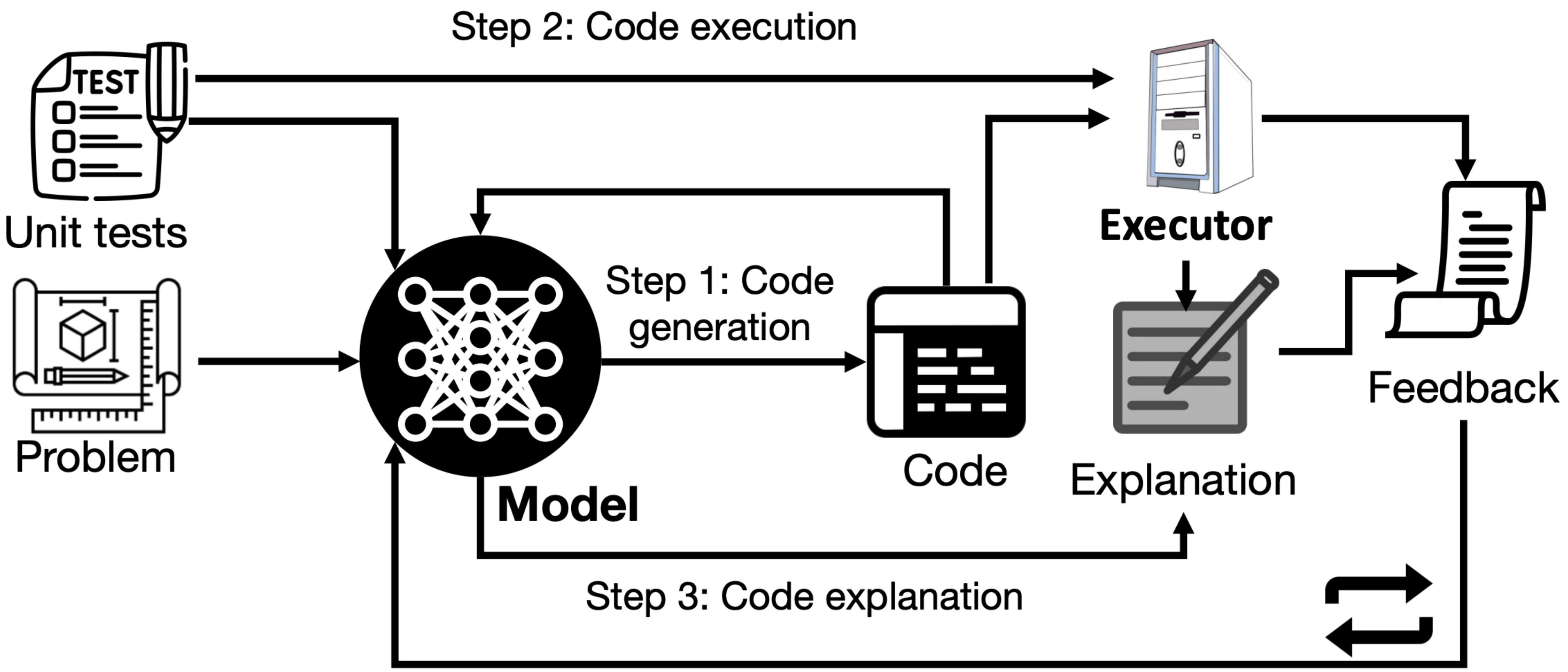
- Scaling enables language models to go beyond memorization and solve unseen challenging algorithmic reasoning problems.
- Leveraging execution results improves the complexity of synthesized programs with input-output examples.
- Still a huge gap to top human programmers.

Part 3: Self-debugging with LLMs



Chen, Lin, Schärli, Zhou, Teaching Large Language Models to Self-Debug.

Part 3: Self-debugging with LLMs



Chen, Lin, Schärli, Zhou, Teaching Large Language Models to Self-Debug.

Recall: LLM-generated code in the first round can be problematic

GPT-4

XI

Write a function that gives loss amount if the given amount has loss else return None. Your Python function should pass the following assertions: assert loss_amount(1500, 1200)=None, assert loss_amount(100, 200)=-100, assert loss_amount(2000, 5000)=-3000

To determine the loss amount, you need to subtract the current value from the original value. If the result is negative, it means there was a loss. If the result is positive or zero, return `None`.

Here's the Python function that follows this logic:

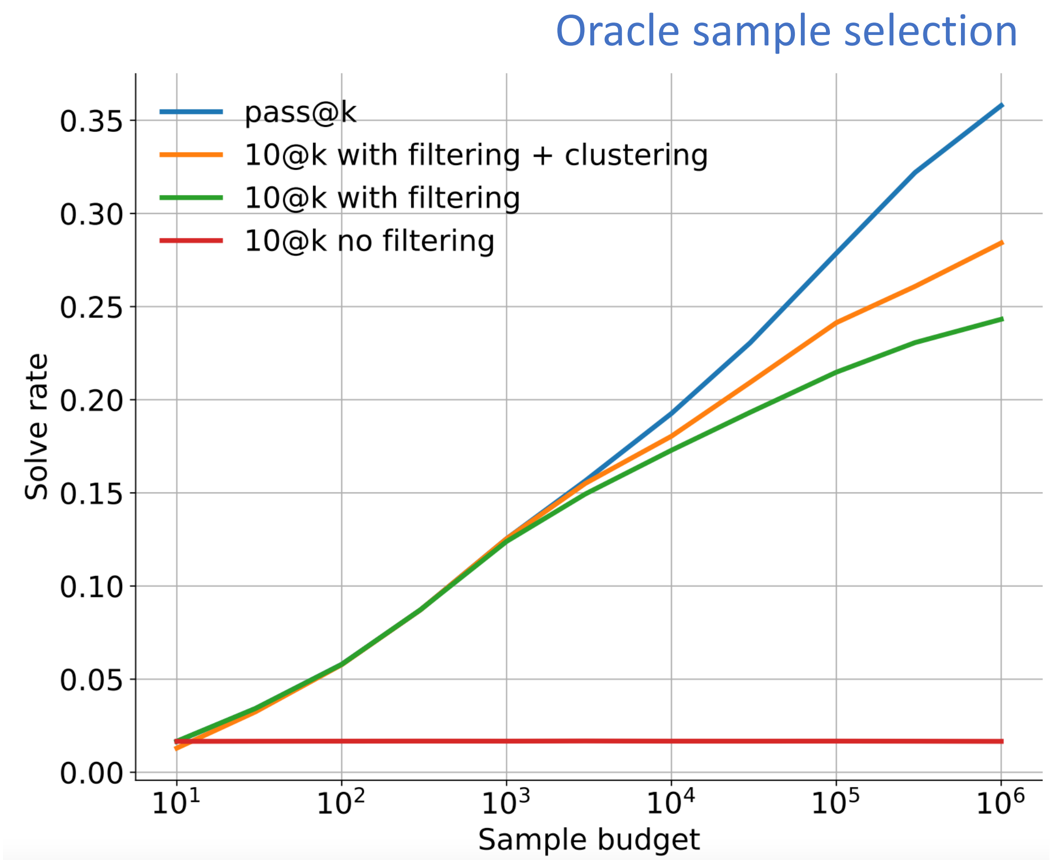
python   Copy code

```python
def loss_amount(original, current):
    difference = current - original
    if difference < 0:
        return difference
    else:
        return None

# Tests
assert loss_amount(1500, 1200) == None
assert loss_amount(100, 200) == -100
assert loss_amount(2000, 5000) == -3000
```

This function should pass the provided assertions.

AlphaCode's approach: code selection using execution results



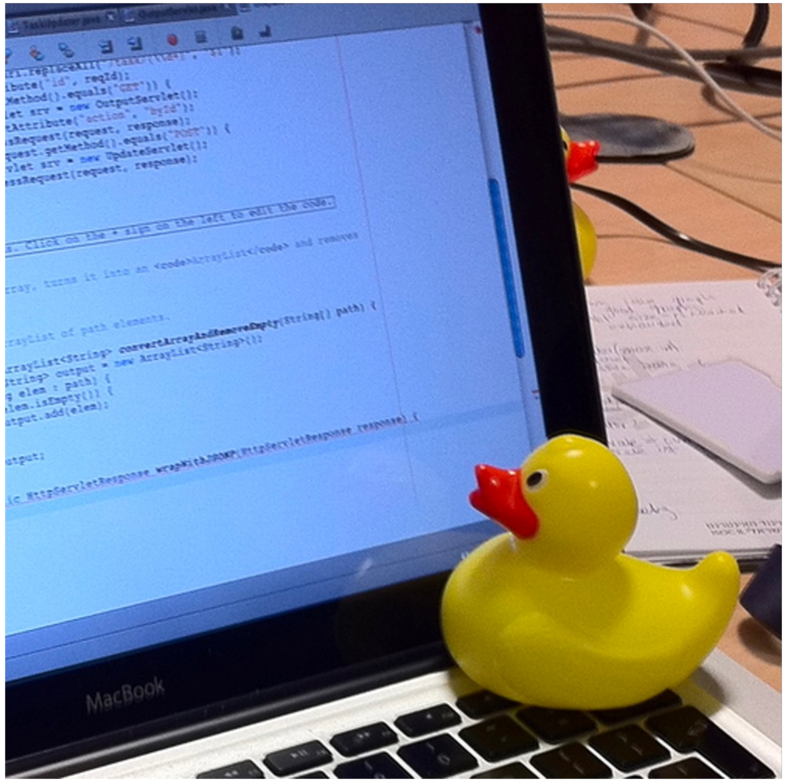
AlphaCode 1B

Limitation: sub-optimal sample efficiency

- Require a lot of model samples
- Does not utilize wrong predictions that can be partially correct

Human-written code in the first round also might not be perfect

- This is why debugging is an important skill for human programmers!
- Good programmers are able to identify bugs by themselves, usually via investigating the execution results and reasoning about the code semantic meaning.
- Self-debug: teach large language models to debug their own predicted code via rubber duck debugging



Chen, Lin, Schärli, Zhou, Teaching Large Language Models to Self-Debug.

Self-debugging overview

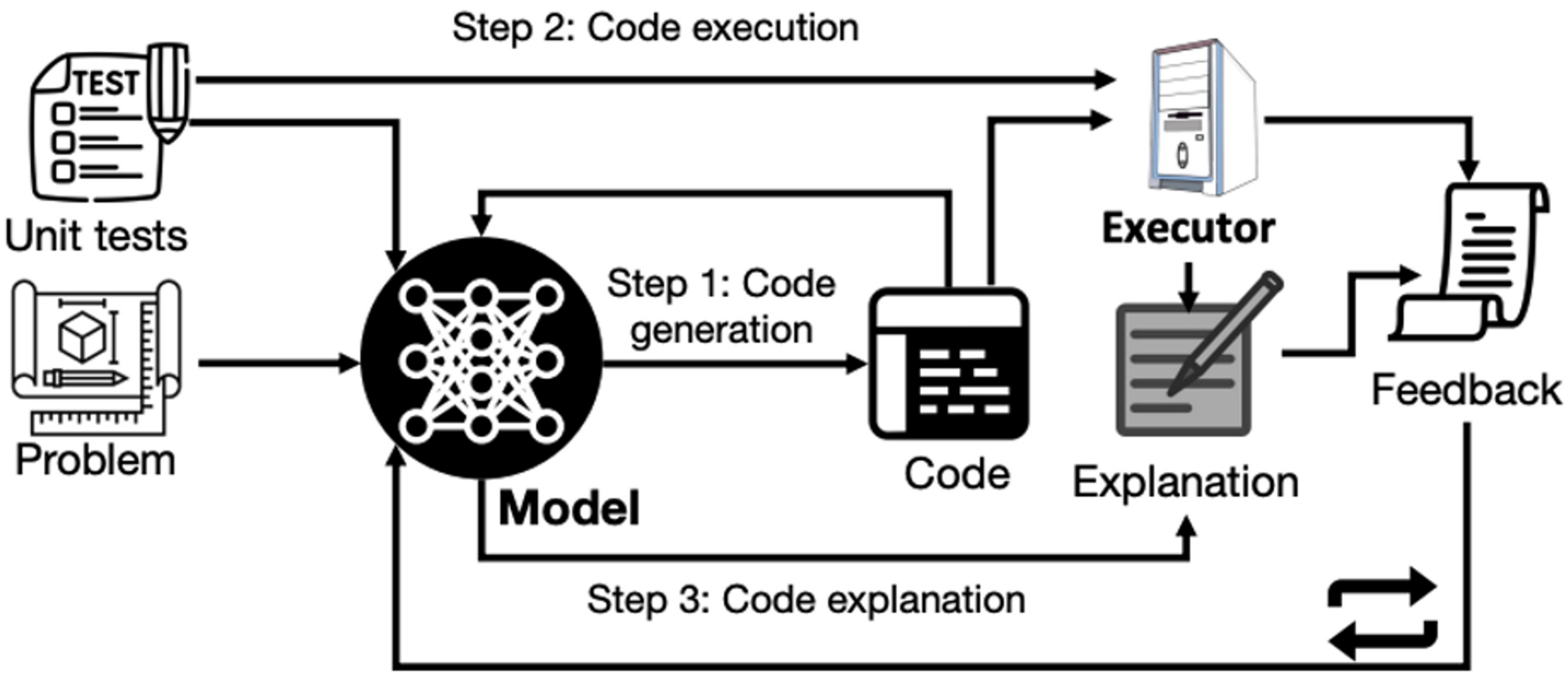


Figure 1: SELF-DEBUGGING for iterative debugging using a large language model. At each debugging step, the model first generates new code, then the code is executed and the model explains the code. The code explanation along with the execution results constitute the feedback message, which is then sent back to the model to perform more debugging steps. When unit tests are not available, the feedback can be purely based on code explanation.

Demo with Bard: code generation

Write a function that gives loss amount if the given amount has loss else return none. Your Python function should pass the following assertions: assert loss_amount(1500,1200)==None , assert loss_amount(100,200)==100 , assert loss_amount(2000,5000)==3000.

Sure, here is the Python function that gives loss amount if the given amount has loss else return none:

Python

```python
def loss_amount(cost_price, selling_price):
  """
  This function calculates the loss amount if the given amount has loss.

  Args:
    cost_price: The cost price of the item.
    selling_price: The selling price of the item.

  Returns:
    The loss amount, if any.
  """

  loss_amount = selling_price - cost_price

  if loss_amount < 0:
    return loss_amount
  else:
    return None
```

Demo with Bard: debugging with code execution

With the above function, loss_amount(1500,1200)==-300. The assertion is " assert loss_amount(1500,1200)==None". So the code does not pass the assertion. Please fix it.

View other drafts

You are correct. The code does not pass the assertion because the function returns -300 instead of None when the cost price is greater than the selling price. To fix this, we can change the `if` statement to:

Python

```python
if loss_amount < 0:
  return None
else:
  return loss_amount
```

Use code with caution. Learn more

This will return None when the cost price is greater than the selling price, and the loss amount when the cost price is less than the selling price.

Self-debugging without unit tests: text-to-SQL generation

Problem description

```
CREATE TABLE customers (
customer_id number ,
customer_name text ,
customer_details text ,
primary key ( customer_id )
)
insert into customers (customer_id, customer_name, customer_details) values (1,
 'Savannah', 'rerum') ;

CREATE TABLE orders (
order_id number ,
customer_id number ,
order_status text ,
date_order_placed time ,
order_details text ,
primary key ( order_id ),
foreign key ( customer_id ) references customers ( customer_id )
)
insert into orders (order_id, customer_id, order_status, date_order_placed,
order_details) values (1, 2, 'Shipped', '2009-02-21 15:26:19', NULL);
...

Question: Which customers have both "On Road" and "Shipped" as order status?
List the customer names.
```

Output SQL

```
SQL: SELECT customers.customer_name FROM customers JOIN orders ON customers.
customer_id = orders.customer_id
WHERE orders.order_status = "On Road"
INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON
customers.customer_id = orders.customer_id
WHERE orders.order_status = "Shipped"
```

- Challenges
  - Require cross-domain generalization
    - Each question can ask about different databases
  - No unit tests
    - The model needs to infer the code correctness by itself
- Approach: self-debugging by explaining the potential inconsistency between the code and the question

Step 1: explain the question and the generated code

Step 1: Question explanation

Infer the return type of the question.

[Question]

Which customers have both "On Road" and "Shipped" as order status? List the customer names.

[Question Explanation]

"List the customer names" returns 1 column. The question returns the customer names who have both "On Road" and "Shipped" as order status. So the question returns 1 column.

Step 2: Code explanation

Summarize the return type of the execution.

[SQL]

SELECT ustomers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.order_status = "On Road" OR orders.order_status = "Shipped"

Execution:

| George |

[SQL Explanation]

The execution of the SQL query above would return a table with 1 column.
The first column, "customers.customer_name" would contain the customer names.
With "customers JOIN orders", the table would contain the data about customers with orders.
With "WHERE orders.order_status = 'On Road' OR orders.order_status = 'Shipped'", the table filters the records to only include customers who have order status "On Road" or "Shipped".
So the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped".

Step 2: self-debugging with the generated explanations

Step 3: Self-Debugging with explanation

[Question]

[Explanation summary]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped". The question returns 1 column, the customer names who have both "On Road" and "Shipped" as order status.

[Feedback]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #1]

SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_status = "On Road" AND orders.order_status = "Shipped"

[Explanation summary #1]

As in your explanation, the SQL query returns an empty table, because the "WHERE" clause contains mutually exclusive conditions. The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped”.

[Feedback #1]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #2]

SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_status = "On Road" INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_status = "Shipped"

[Explanation summary #2]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" and "Shipped". The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped". So the question returns 1 column.

[Feedback #2]

So the SQL prediction above is correct!

Application 2: code translation

C++ Program

```
string caesar_cipher ( string text,
int s ) {
  string result = "";
  for ( int i = 0;
  i < text . length ( );
  i ++ ) {
    if ( isupper ( text [ i ] ) )
    result += char ( int ( text [ i ]
    + s - 65 ) % 26 + 65 );
    else result += char ( int ( text [
     i ] + s - 97 ) % 26 + 97 );
  }
  return result;
}
```

Python Program

```
def caesar_cipher(text, s):
    result = ''
    for i in range(len(text)):
        char = text[i]
        if char.isupper():
            result += chr((((ord(char
            ) + s) - 65) % 26) + 65))
        else:
            result += chr((((ord(char
            ) + s) - 97) % 26) + 97))
    return result
```

All unit tests are available for debugging.

LLM does not need to infer the code correctness, can focus on fixing the wrong code.

Unit Tests

```
assert caesar_cipher('35225904', 2) == 'ikhhkofj'
... (8 unit tests omitted)
assert caesar_cipher('11', 93) == 'tt'
```

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 with different feedback formats

Simple Feedback	Unit Test (UT) Feedback	Unit Test + Explanation (+Expl.)	Unit Test + Trace (+Trace)
Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback. [C++] [Original Python] [Simple Feedback] [Revised Python #1] [Simple Feedback] [Revised Python #2] …	Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback. [C++] [Original Python] [UT Feedback] [Revised Python #1] [UT Feedback] [Revised Python #2] …	Below are C++ programs with incorrect Python translations. *Explain the original code, then explain the translations line by line* and correct them using the provided feedback. [C++] [C++ Explanation] [Original Python] [Python Explanation] [UT Feedback] [Revised Python #1] [Python Explanation] [UT Feedback] [Revised Python #2] [Python Explanation] …	Below are C++ programs with incorrect Python translations. Using the provided feedback, *trace through the execution of the translations to determine what needs to be fixed*, and correct the translations. [C++] [Original Python] [UT Feedback] [Trace] [Revised Python #1] [UT Feedback] [Trace] [Revised Python #2] …

Application 3: text-to-Python generation

```
# These are the assertions for your function:
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)

""" Write a function to find the similar elements from the given two tuple
lists. """
```

```
def similar_elements(test_tup1, test_tup2):
    res = tuple(set(test_tup1) & set(test_tup2))
    return (res)
```

- MBPP: 1 unit test in the prompt, 2 hidden unit tests for evaluation
- Passing the given unit test does not necessarily mean that the predicted code is correct
- The model still needs to infer the code correctness when the predicted code passes the given unit test

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	**72.2**	73.4	**64.9**
+Expl.	**84.1**	**72.2**	**73.6**	**64.9**

(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	**92.7**	88.8	76.4
+ Expl.	**92.5**	**92.7**	**90.4**	**76.6**
+ 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	**80.6**	52.2
+ Expl.	69.8	**74.2**	80.4	52.2
+ Trace.	**70.8**	72.8	80.2	**53.2**

- StarCoder: 15.5B open-source coding LLM
- Self-debugging consistently boosts the performance across different LLMs

More informative feedback improves self-debugging performance

(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	**92.7**	88.8	76.4
+ Expl.	**92.5**	**92.7**	**90.4**	**76.6**
+ 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	**80.6**	52.2
+ Expl.	69.8	**74.2**	80.4	52.2
+ Trace.	**70.8**	72.8	80.2	**53.2**

- Note: simple feedback also utilizes code execution to indicate the code correctness
- Adding execution results (UT) consistently improves the performance over the generic simple feedback
- LLM-generated code explanation can provide additional gain

Self-debugging can be triggered with few-shot prompting

(a) Results on the Spider development set.

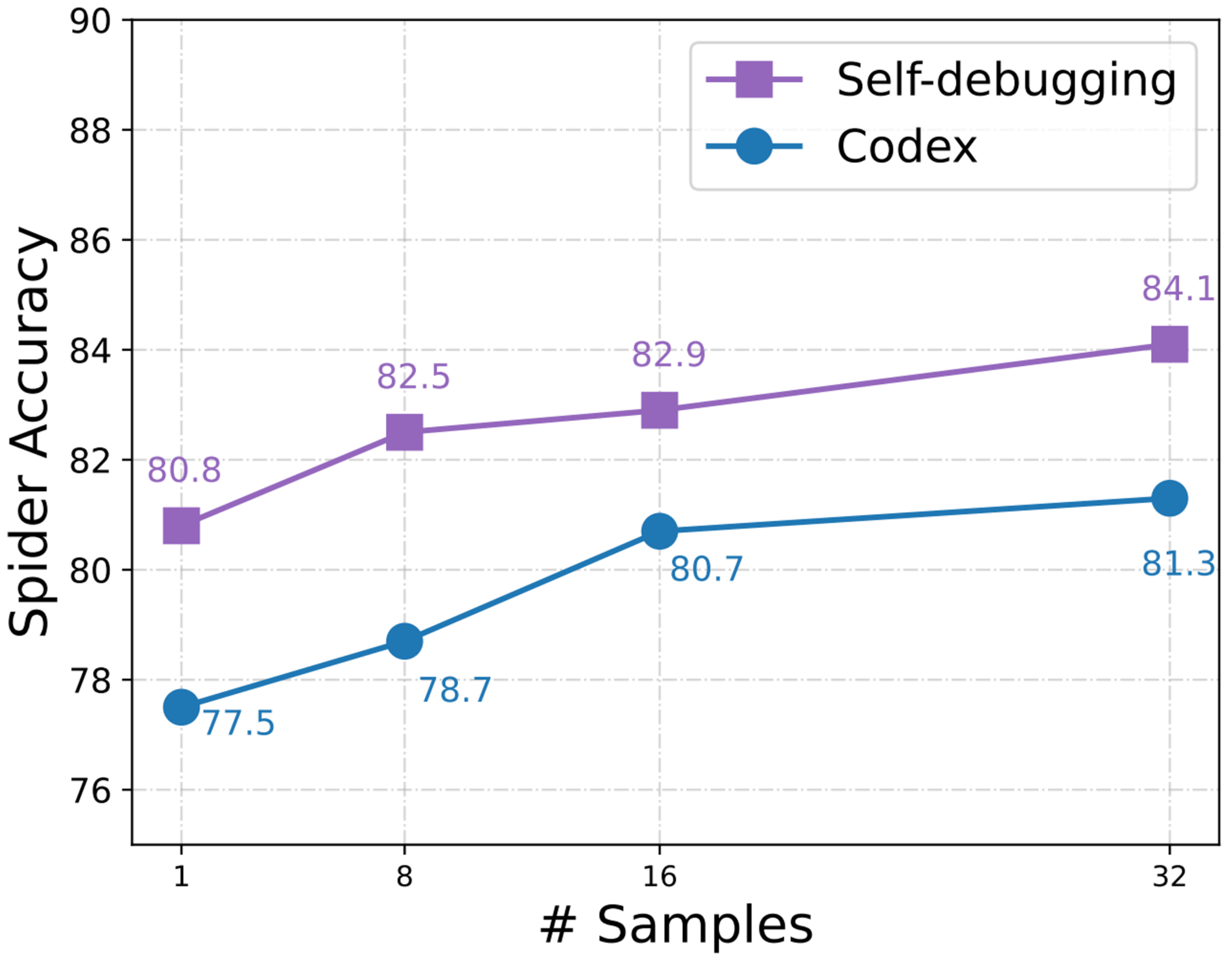
Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	**72.2**	73.4	**64.9**
+Expl.	**84.1**	**72.2**	**73.6**	**64.9**

(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	**80.6**	52.2
+ Expl.	69.8	**74.2**	80.4	52.2
+ Trace.	**70.8**	72.8	80.2	**53.2**

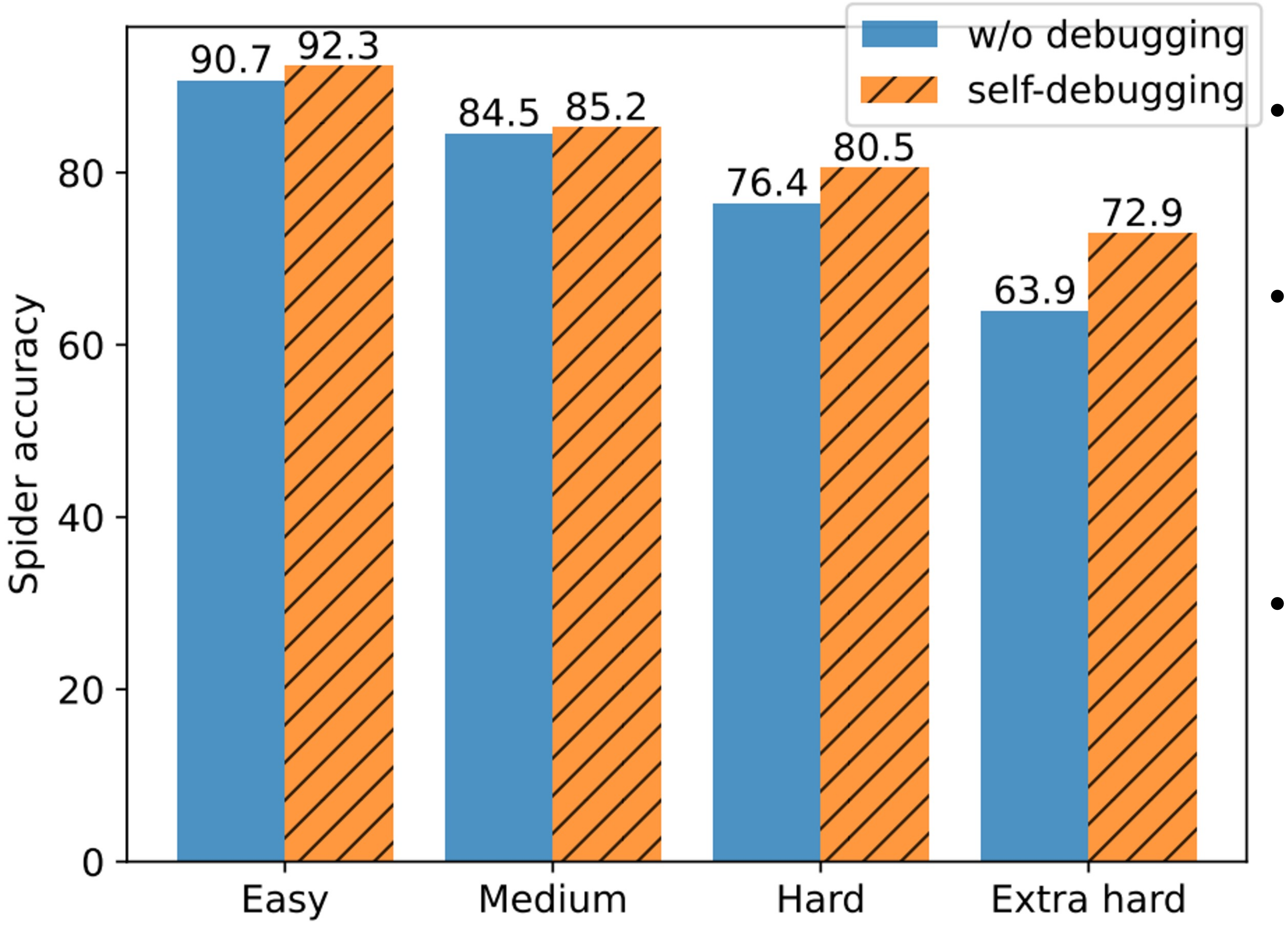
- Codex performs the best at SQL with few-shot prompting
- GPT-4's performance gain using self-debugging is on par with Codex

Self-debugging improves sample efficiency



Self-debugging from greedy decoding can match the baseline performance which utilizes >10x samples

What error types can be fixed by self-debugging?



- 9% improvement on the hardest SQL tasks
- Self-debugging fixes subtle mistakes in code; e.g., missing WHERE conditions in complex SQL queries
- Note: self-debugging does not improve the fundamental coding capability of LLMs

Importance of code execution for self-debugging

(a) Results on Transcoder.

TransCoder	Codex	GPT-3.5	GPT-4
Baseline	80.4	**89.1**	77.3
Simple	83.4	**89.1**	78.2
+ Expl.	**83.9**	**89.1**	78.0
+ Trace.	**83.9**	**89.1**	**78.4**

(b) Results on MBPP

MBPP	Codex	GPT-3.5	GPT-4
Baseline	61.4	67.6	72.8
Simple	57.6	68.2	76.0
+ Expl.	64.4	68.2	76.0
+ Trace.	**66.2**	**69.2**	**76.4**

Self-debugging without code execution

- Self-debugging improvement is less significant w/o code execution, but can still bring up to 5% performance gain using Codex and GPT-4
- Trace feedback simulates the execution outcome and provides helpful information for self-debugging

Discussion: valid external feedback is crucial for self-correction

		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	**75.9**	**75.8**	**26.0**
	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	**95.5**	**82.0**	**49.0**
	Self-Correct (round 1)	3	91.5	79.5	**49.0**
	Self-Correct (round 2)	5	89.0	80.0	43.0

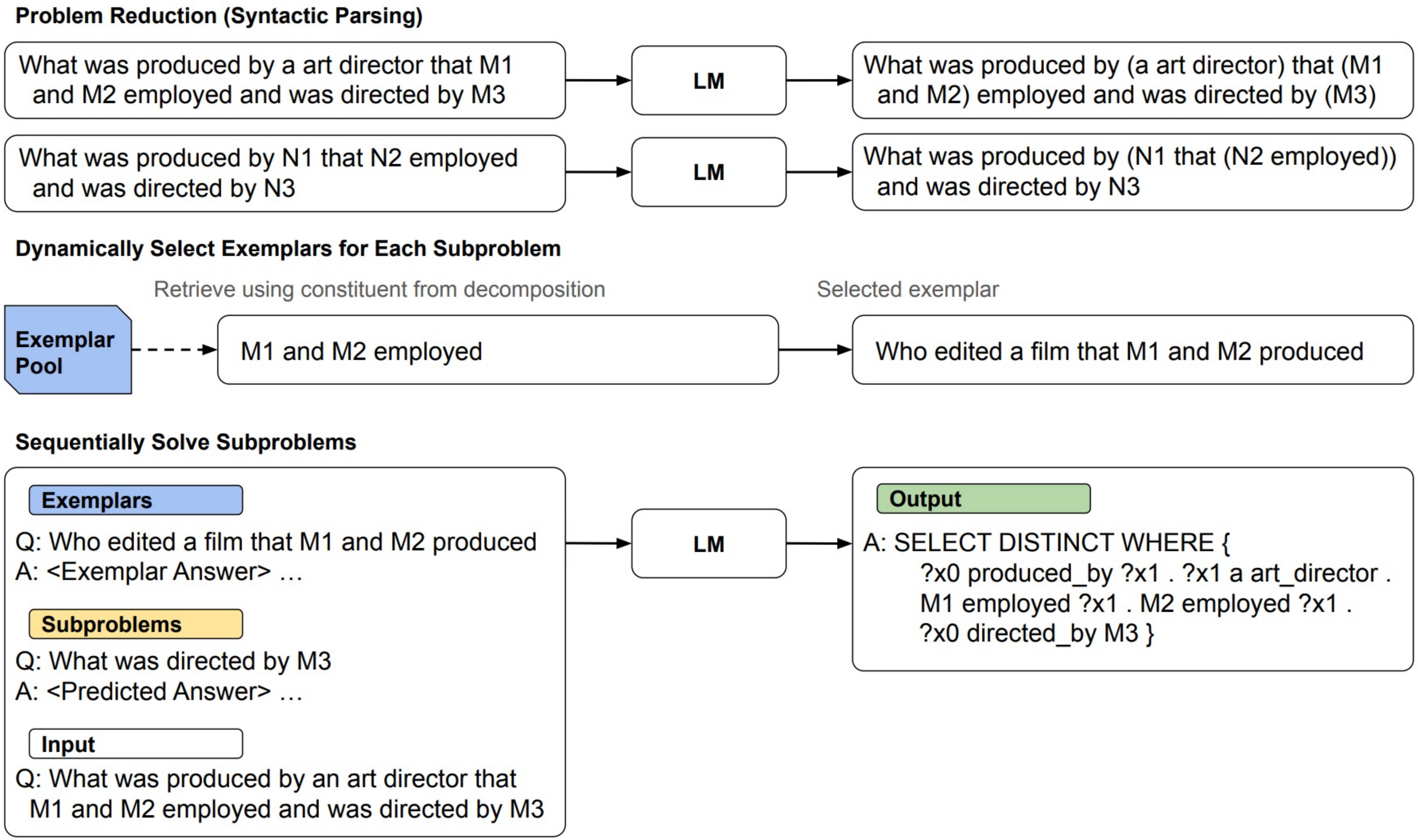
- Oracle: utilize the ground truth answer for correction
- Without oracle feedback for correction, LLMs can wrongly judge the correctness of its predictions for reasoning problems, leading to worse performance after self-correction
- Code execution provides natural external feedback: humans often debug better within an IDE

Huang, **Chen**, Mishra, Zheng, Yu, Song, Zhou, Large Language Models Cannot Self-Correct Reasoning Yet.

Takeaway

- We can teach LLMs to self-debug via few-shot prompting, even if the LLM itself was not specifically tuned for debugging
- Significant improvement across several coding tasks, including those that do not have unit tests
- Self-debugging is not just an approach: it is another indicator of the LLM coding capability

Part 4: Compositional generalization for code generation



Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.

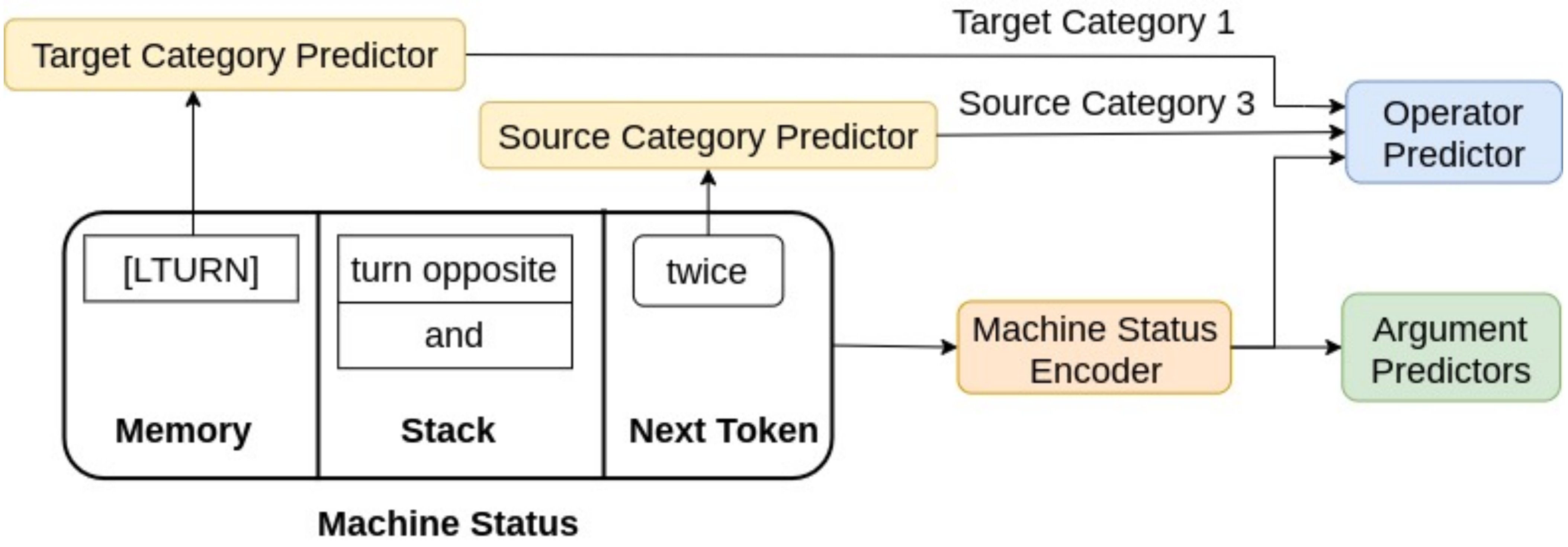
Compositional generalization for sequence-to-sequence learning

- **Compositional generalization**: given basic components and a few demonstrations of their combinations, generalize to novel combinations.

- **Primitive generalization**: generalize to novel combinations of primitives, where the test inputs and outputs have similar lengths to the training samples.
  - Example:
    - Training on how to translate “walk”, “walk twice”, and “jump”.
    - Testing on how to translate “jump twice”.

- **Length generalization**: generalize to longer test samples than the training ones.
  - Example:
    - Training on how to parse “while (x) {x = x + 1;}”, “x = x + 1; y = y + x;”.
    - Testing on how to parse “while (x) {x = x + 1; y = y + x;}”.

Prior successful attempts mainly come from neural-symbolic learning



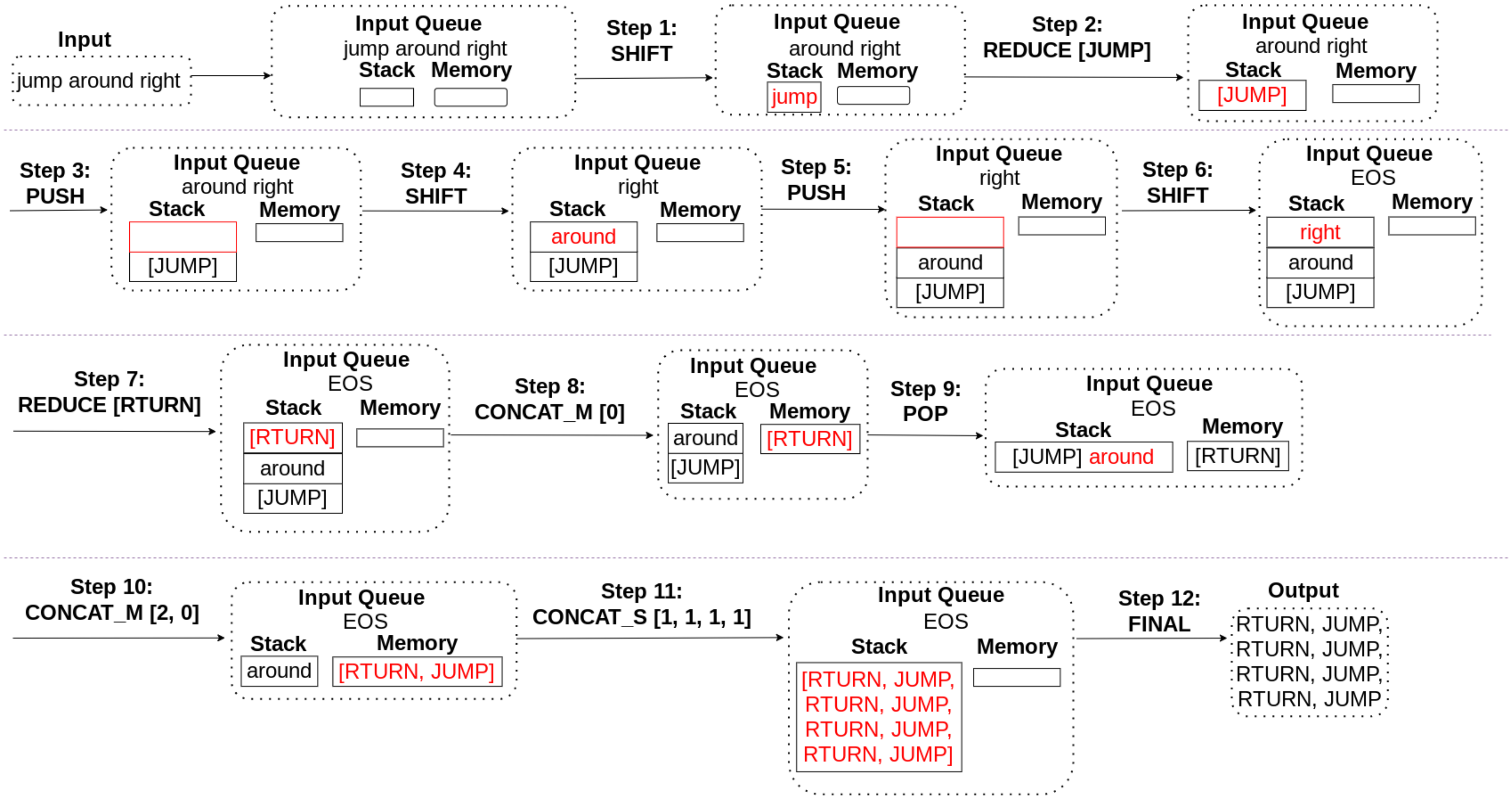
Neural controller: predicts the execution traces to produce the output sequences.

+

Symbolic stack machine: supports symbolic execution of sequence manipulation operations.

Chen, Liang, Yu, Song, Zhou, Compositional Generalization via Neural-Symbolic Stack Machines, NeurIPS 2020.
Chen, Liu, Song, Towards Synthesizing Complex Programs from Input-Output Examples, ICLR 2018.

Demonstration of the symbolic machine



Can LLMs improve compositional generalization without symbolic execution?

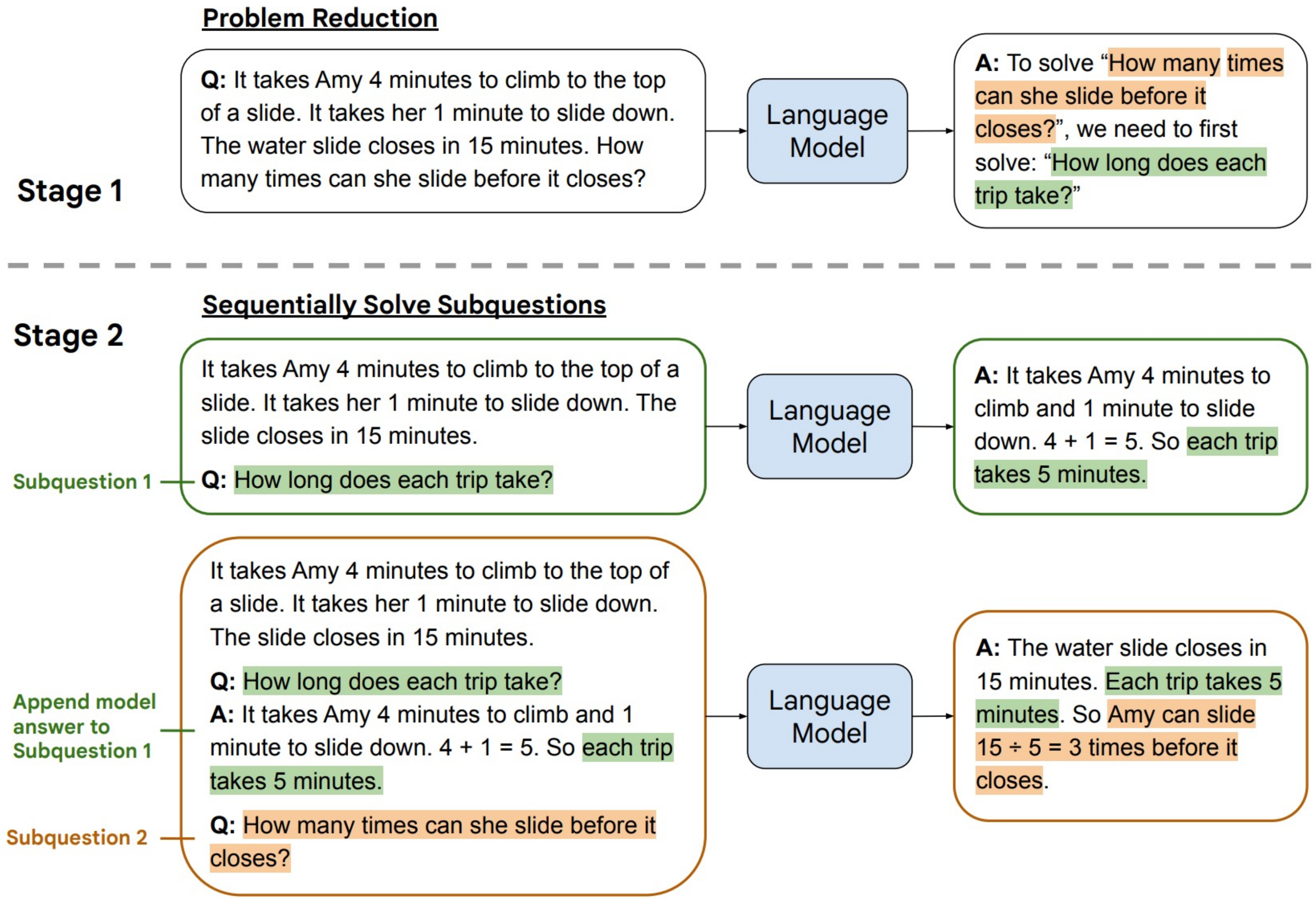
Method	Standard prompting	Chain-of-Thought	Least-to-Most
code-davinci-002	16.7	16.2	**99.7**
text-davinci-002	6.0	0.0	**76.0**
code-davinci-001	0.4	0.0	**60.7**

SCAN length split

- YES, when:
  - with few-shot demonstrations of reasoning rationales
  - the language model has good prior knowledge of both text and code

Zhou et al., Least-to-Most Prompting Enables Complex Reasoning in Large Language Models, ICLR 2023.
Wei et al., Chain of Thought Prompting Elicits Reasoning in Large Language Models, NeurIPS 2022.

Least-to-most prompting



Zhou et al., Least-to-Most Prompting Enables Complex Reasoning in Large Language Models, ICLR 2023.

Least-to-most prompting for compositional generalization

- Stage 1: command reduction

Q: "look opposite right thrice after walk"
A: "look opposite right thrice" can be solved by: "look opposite right", "look opposite right thrice". "walk" can be solved by "walk". So, "look opposite right thrice after walk" can be solved by: "look opposite right", "look opposite right thrice", "walk".

Q: "jump around right"
A: "jump around right" can be solved by: "jump right", "jump around right". So, "jump around right" can be solved by: "jump right", "jump around right".

Q: "look around right thrice and walk"
A: "look around right thrice" can be solved by: "look right", "look around right", "look around right thrice". "walk" can be solved by "walk". So, "look around right thrice and walk" can be solved by: "look right", "look around right", "look around right thrice", "walk".

Zhou et al., Least-to-Most Prompting Enables Complex Reasoning in Large Language Models, ICLR 2023.

Least-to-most prompting for compositional generalization

- Stage 2: command mapping

Q: “jump right”
A: The output of “jump right” concatenates: the output of “turn right”, the output of “jump”. “turn right” outputs “TURN_RIGHT”. “jump” outputs “JUMP”. So concatenating the output of “turn right” and the output of “JUMP” leads to “TURN_RIGHT” + “JUMP”. So the output of “jump right” is “TURN_RIGHT” + “JUMP”.

Q: “jump right twice”
A: The output of “jump right twice” concatenates: the output of “jump right”, the output of “jump right”. “jump right” outputs “TURN_RIGHT” + “JUMP”. So repeating the output of “jump right” two times leads to (“TURN_RIGHT” + “JUMP”) * 2. So the output of “jump right twice” is (“TURN_RIGHT” + “JUMP”) * 2.

Q: **“jump right twice after walk”**
A: The output of “jump right twice after walk” concatenates: the output of “walk”, the output of “jump right twice”. “walk” outputs “WALK”. “jump right twice” outputs (“TURN_RIGHT” + “JUMP”) * 2. So concatenating the output of “walk” and the output of “jump right twice” leads to “WALK” + (“TURN_RIGHT” + “JUMP”) * 2. So the output of “jump right twice after walk” is “WALK” + (“TURN_RIGHT” + “JUMP”) * 2.

Zhou et al., Least-to-Most Prompting Enables Complex Reasoning in Large Language Models, ICLR 2023.

Compositional generalization for code generation

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
  - Single prompt is insufficient to cover all grammar rules

Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.
Keyers et al., Measuring Compositional Generalization: A Comprehensive Method on Realistic Data, ICLR 2020.

Compositional generalization for code generation

Examples in CFQ (Compositional Freebase Questions) benchmark

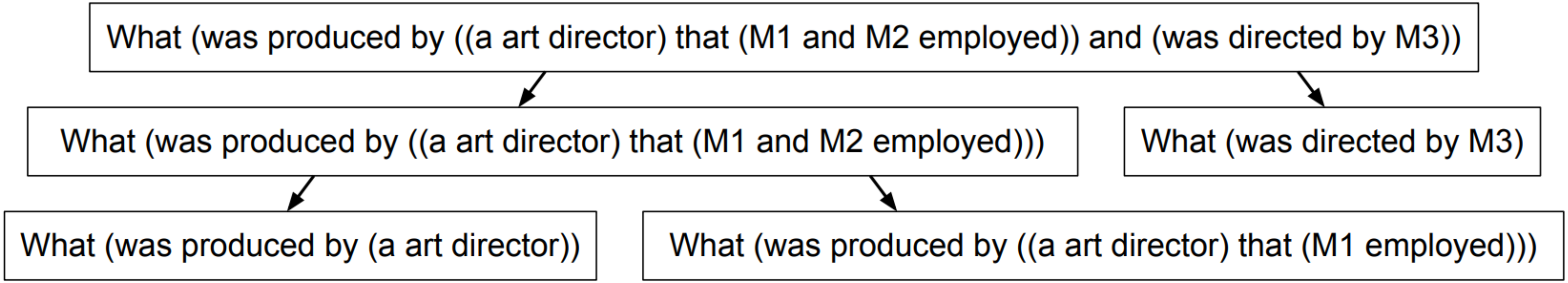
Question: Did M1 star M2 , star M3 , and star **a art director** and editor of M0?
SPARQL: SELECT count(*) WHERE
{ ?x0 edited M0 . **?x0 art directed M0** . M1 starred ?x0 . M1 starred M2 . M1 starred M3 }

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
  - Single prompt is insufficient to cover all grammar rules
  - Context-dependent constituent translation

Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.
Keyers et al., Measuring Compositional Generalization: A Comprehensive Method on Realistic Data, ICLR 2020.

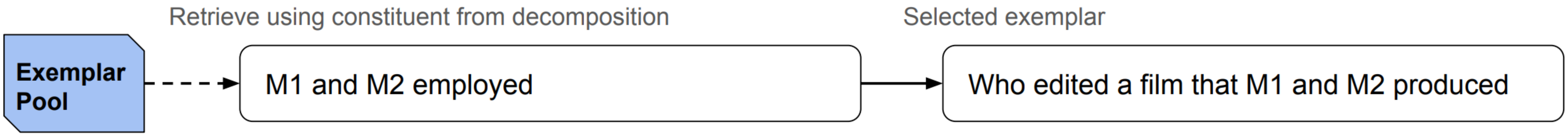
Our approach: problem decomposition with syntactic parsing



- Tree-structured decomposition with LLM prompting
- Each node represents a subproblem linearized as a well-formed sentence

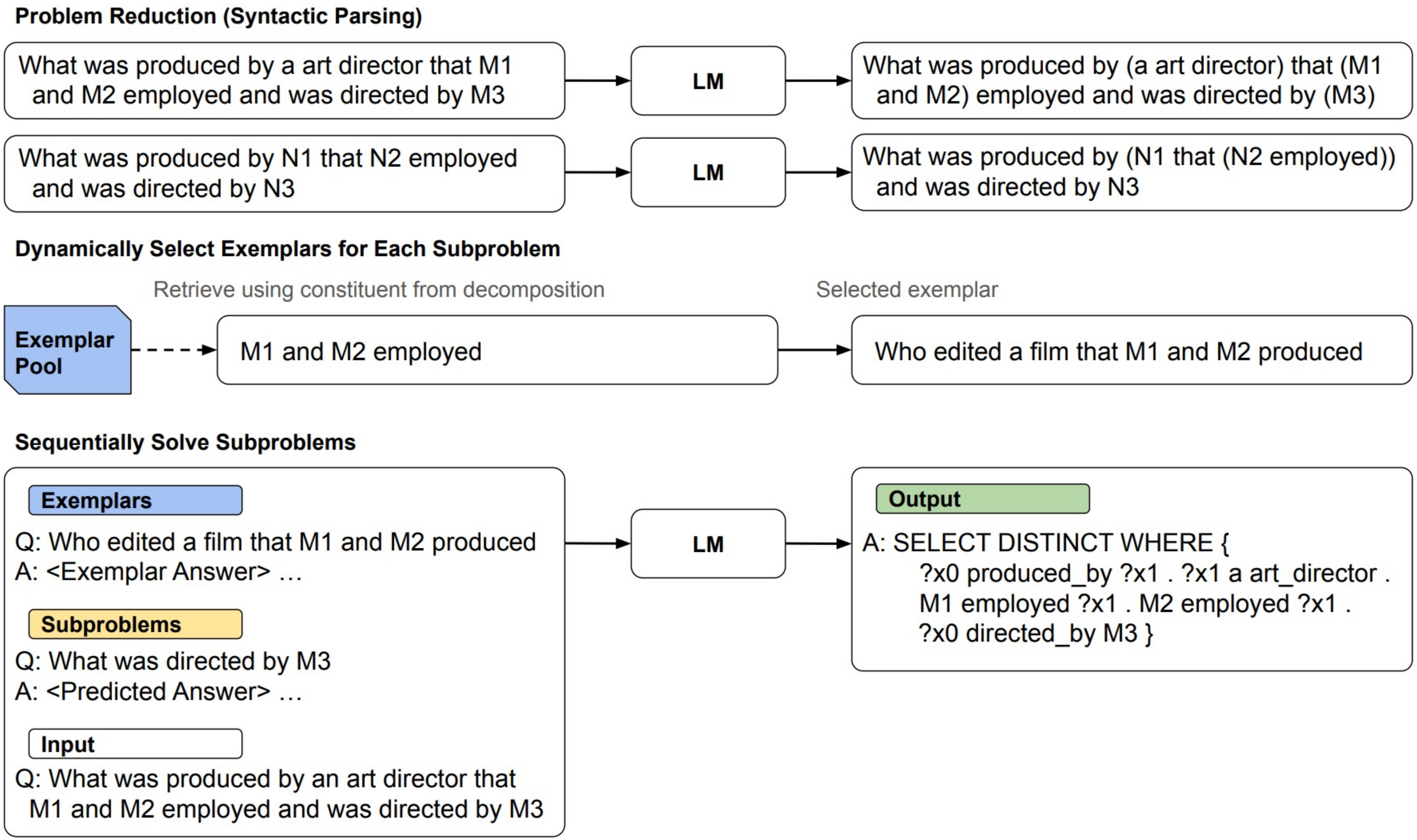
Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.

Our approach: dynamic exemplar selection for each subproblem



- Single prompt is insufficient to cover all grammar rules
- Exemplar selection based on subtree matching

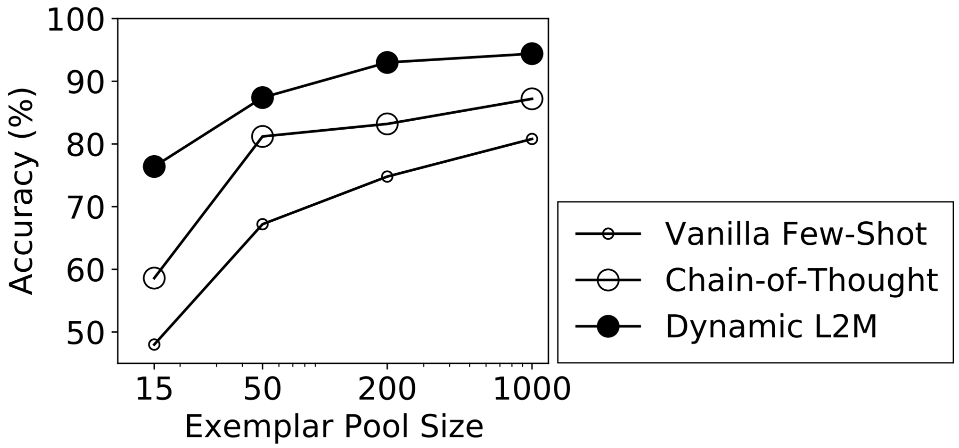
Overview: dynamic least-to-most prompting



Drozdov*, Scharli*, Akyurek, Scales, Song, **Chen**, Bousquet, Zhou, Compositional Semantic Parsing with Large Language Models, ICLR 2023.

Evaluation on CFQ

	MCD1	MCD2	MCD3	Ave.
Fully Supervised				
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
Prompting				
(Ours) Dynamic Least-to-Most	**94.3**	**95.3**	**95.5**	**95.0**



Herzig et al., Unlocking Compositional Generalization in Pre-trained Models Using Intermediate Representations.
Liu et al., Learning Algebraic Recombination for Compositional Generalization, ACL Findings, 2021.
Guo et al., Hierarchical Poset Decoding for Compositional Generalization in Language, NeurIPS 2020.

Evaluation on COGS

<u>Question:</u> James said that a manager liked that Aiden appreciated that Emily believed that the girl was posted a cake beside a table by Olivia.

<u>Parse:</u> say (agent = James , ccomp = like (agent = manager , ccomp = appreciate (agent = Aiden , ccomp = believe (agent = Emily , ccomp = post (recipient = * girl , theme = cake (nmod . beside = table) , agent = Olivia))))) DONE

	Gen.
Fully Supervised	
LeAR (Liu et al., 2021)	97.7
T5-base (Qiu et al., 2022a)	89.8
T5-base + CSL (Qiu et al., 2022a)	**99.5**
Prompting	
(Ours) Dynamic Least-to-Most	99.2

Kim and Linzen, COGS: A Compositional Generalization Challenge Based on Semantic Interpretation, EMNLP 2020.
Liu et al., Learning Algebraic Recombination for Compositional Generalization, ACL Findings, 2021.
Qiu et al., Improving Compositional Generalization with Latent Structure and Data Augmentation, NAACL 2022.

Prompting without exemplar design: analogical prompting

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, …, 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 ≤ k ≤ n-1, and a_1 · a_2 … a_k = a_{k+1} … 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:
```

Yasunaga, **Chen**, Li, Pasupat, Leskovec, Liang, Chi, Zhou, Large Language Models as Analogical Reasoners.

Analogical prompting for code generation

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: ......

Generated example problems can be much simpler than the testing problem

Evaluation on Codeforces problems

Prompting Method	GPT3.5-turbo-16k		GPT4	
	Acc@1	Acc@10	Acc@1	Acc@10
0-shot	8%	24%	16%	30%
0-shot CoT	9%	27%	16%	29%
3-shot CoT	11%	27%	17%	31%
Ours: Self-generated Exemplars	13%	25%	17%	32%
Ours: Self-generated Knowledge + Exemplars	**15%**	**29%**	**19%**	**37%**

- To prevent test set contamination, the test set contains level-A (simplest) problems in 2023 Codeforces contests
- Self-generated exemplars outperform zero-shot prompting and CoT with manually-designed exemplars
- Self-generated knowledge further improves the performance

Evaluation on other reasoning problems

Prompting Method	GSM8K Accuracy			MATH Accuracy	
	GPT3.5-turbo	text-davinci-003	PaLM2	GPT3.5-turbo	PaLM2
0-shot	75.0%	14.8%	60.8%	33.0%	27.1%
0-shot CoT	75.8%	50.3%	78.2%	33.9%	29.8%
5-shot CoT	76.7%	54.0%	80.7%	34.9%	34.3%
Ours: Self-generated Exemplars	**77.8%**	**61.0%**†	**81.7%**	**37.3%**	**34.8%**

Mathematical reasoning

Prompting Method	Word sorting	Logical deduction five objects	Temporal sequences	Reasoning about colored objects	Formal fallacies
0-shot	66.8%	30.0%	40.4%	50.4%	53.6%
0-shot CoT	67.6%	35.2%	44.8%	61.6%	55.6%
3-shot CoT	68.4%	36.4%	**58.0%**	62.0%	55.6%
Ours: Self-generated Exemplars	**75.2%**	**41.6%**	57.6%	**68.0%**	**58.8%**

Big-Bench tasks

Takeaway

- LLMs can be taught to elicit the compositional generalization capability, via demonstrations of problem reduction steps.
- Code pretraining is not only critical for doing code generation tasks, but also plays a key role in achieving compositional reasoning

Summary

- Large language models provide a unified natural language interface for general-purpose code generation
- Language models show promising progress on solving challenging coding problems, with room for improvement on sample efficiency
- Self-debugging enables the LLM to leverage code execution and reflect on the code semantic meaning to fix its own prediction
- Joint pretraining of text and code empowers both code generation and other reasoning capabilities

Thanks!

Xinyun Chen
Google DeepMind
xinyunchen@google.com

- Large language models provide a unified natural language interface for general-purpose code generation
- Language models show promising progress on solving challenging coding problems, with room for improvement on sample efficiency
- Self-debugging enables the LLM to leverage code execution and reflect on the code semantic meaning to fix its own prediction
- Joint pretraining of text and code empowers both code generation and other reasoning capabilities