Large Language Models for Code Generation

Xinyun Chen
Google DeepMind
xinyunchen@google.com
Code generation

User → Program specification → Synthesizer → Program
Success of LLMs for text-to-code generation

When the rocket is clicked, temporarily display some text saying "Firing thrusters!" in white on the current location -- and temporarily speed up by 4x for 0.25 second.

OpenAI Codex

Chen et al., Evaluating Large Language Models Trained on Code.
DeepMind AlphaCode: competitive programming

Li, ..., Chen et al., Competition-level Code Generation with AlphaCode, Science 2022.
Code generation for tool use

Find the alkali metal with the highest resistivity

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

```wolfram
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

**Tool making template (One-time)**

- **Tool proposing:** Write a generic Python function (the Tool) to solve three training samples.
- **Tool verification:** Write unit tests to convert three validation samples into function call and validate the correctness.
- **Tool wrapping:** Gather the function from the proposing stage and the examples of how to convert problems to function calls from the verification stage into a reusable **Wrapped Tool**.

**Tool Maker** (e.g., GPT-4):
Strong performance but slow and expensive

**Tool User** (e.g., GPT-3.5 Turbo):
Weaker performance but much faster and cheaper

Convert problem into function call according to the **Wrapped Tool**

Cai, Wang, Ma, Chen, Zhou, Large Language Models as Tool Makers.
## Challenge 1: Algorithmic Reasoning

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

OpenAI, GPT-4 Technical Report.
Challenge 2: predicted code in the first round can be problematic

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

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:

```
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.
Challenge 3: compositional generalization

- 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.
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

<table>
<thead>
<tr>
<th>Input $v_1$</th>
<th>Input $v_2$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>355</td>
<td>case 355: return &quot;Albania&quot;;</td>
</tr>
<tr>
<td>Algeria</td>
<td>213</td>
<td>case 213: return &quot;Algeria&quot;;</td>
</tr>
</tbody>
</table>

String Program:

$\text{Concatenate(\text{ConstStr("case"), }v_2, \text{ConstStr("\text{:\ return ""),} \vspace{0.5em}}$

$v_1, \text{ConstStr(";"))}$

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
\[ \lambda x. \text{state}(x) \land \text{borders}(x, \text{arg max}(\lambda y. \text{state}(y), \lambda z. \text{count}(\lambda z. \text{state}(z) \land \text{borders}(y, z)))) \]

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

<table>
<thead>
<tr>
<th>Input $v_1$</th>
<th>Input $v_2$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>355</td>
<td>case 355: return “Albania”;</td>
</tr>
<tr>
<td>Algeria</td>
<td>213</td>
<td>case 213: return “Algeria”;</td>
</tr>
</tbody>
</table>

String Program:
\[
\text{Concatenate}(\text{ConstStr(“case”), } v_2, \text{ConstStr(“: return “),}\left< v_1, \text{ConstStr(“;“)} \right>)
\]

FlashFill:
string processing in Microsoft Excel
[Gulwani 2011; Polozov et al., 2015]
Code generation: transition with learning-based techniques

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

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

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

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
\( \lambda x.\text{state}(x) \land \text{borders}(x, \text{arg max}(\lambda y.\text{state}(y), \lambda y.\text{count}(\lambda z.\text{state}(z) \land \text{borders}(y, z)))) \)

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

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

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

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Test Item</th>
<th>Score</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>VAL_P3_A_I</td>
<td>2.9 C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VAL_P3_A_II</td>
<td>1.5 B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VAL_P3_A_III</td>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VAL_P3_A_IV</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VAL_P3_A_V</td>
<td>0.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input (v_1)</th>
<th>Input (v_2)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>355</td>
<td>case 355: return “Albania”;</td>
</tr>
<tr>
<td>Algeria</td>
<td>213</td>
<td>case 213: return “Algeria”;</td>
</tr>
</tbody>
</table>

String Program:
\( \text{Concatenate}(\text{ConstStr}(“case”), v_2, \text{ConstStr}(“: return “"), \text{ConstStr}("","")) \)

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

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

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

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

What states border the state that borders the most states?
\[ \lambda x.\text{state}(x) \land \text{borders}(x, \text{arg max}(\lambda y.\text{state}(y), \lambda y.\text{count}(\lambda z.\text{state}(z) \land \text{borders}(y, z)))) \]

Learning probabilistic grammars

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

FlashFill:
string processing in Microsoft Excel
[Gulwani 2011; Polozov et al., 2015]
Code generation: transition with large language models

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?

```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 large language models

<table>
<thead>
<tr>
<th>Index</th>
<th>Test item</th>
<th>Score</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>VAL_P3_A</td>
<td>2.9 C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VAL_P3_A</td>
<td>1.5 B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VAL_P3_A</td>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VAL_P3_A</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VAL_P3_A</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Total Score</td>
<td>6.7</td>
<td></td>
</tr>
</tbody>
</table>

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
```

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

Before LLMs: encoder-decoder architecture for programming by examples

Before LLMs: decoding schemes to utilize the code execution

\[ \hat{I}_t: \text{(latent) execution state at step } t \]
\[ (I_0, O): \text{input-output example} \]


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 debugging
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.
Part 2: LLMs for competition-level code generation

Problem (input)

D. Backspace

You are given two strings a and b, both consisting of lowercase English letters. The strings are separated by a space. The first string is the target string, and the second one is the code to be executed. When typing a character, instead of pressing the backspace key, you are required to execute the code corresponding to that character. Here is an example:

Input:

```
alpha
```

Output:

```
1
```

Note: Consider the example text to be completed.

Part 2: LLMs for competition-level code generation

Solution (output)

```
t Test(input){
 for i in range(n):
   x=input()
   w[i]
   for j in a:
     a.append(j)
   for j in t:
     b.append(j)
   w source
   b source
   w[]
   while len(a)>0 and len(b)>0:
     if a[0]<t[b[0]]:
       b source
     a pop(0)
     w source
     if b[0]<t[a[0]] and len(a)>1:
       a source
     b pop(0)
     w source
     if len(a)==1 and len(w)>1:
       a source
       w pop(0)
     else:
       print("YES")
   else:
     print("NO")
}
```
Competitive programming: input format

D. Backspace

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

When typing a character, instead of pressing the button corresponding to it, you can press the “Backspace” button. It deletes the last character you have typed among those that aren’t deleted yet (or does nothing if there are no characters in the current string). For example, if $s$ is “abcba” and you press Backspace instead of typing the first and the fourth characters, you will get the string “bc” (the first press of Backspace deletes no character, and the second press deletes the character ‘c’). Another example, if $s$ is “abaca” 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 $r$, if you type the string $s$ and press “Backspace” instead of typing several (maybe zero) characters of $s$.

### Input

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

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

The second line of each test case contains the string $r$ ($1 \leq |r| \leq 10^5$). Each character of $r$ 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 $r$ 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).

### 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 “aaaa”.

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>YES</td>
</tr>
<tr>
<td>ababa</td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td></td>
</tr>
<tr>
<td>ababa</td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td></td>
</tr>
<tr>
<td>aaa</td>
<td></td>
</tr>
<tr>
<td>aababa</td>
<td></td>
</tr>
<tr>
<td>ababa</td>
<td></td>
</tr>
</tbody>
</table>

Long and complicated text description + input-output specification
Competitive programming: sample decoded program

```python
# Problem (input)
alpha_code = AlphaCode()

# Solution (output)

# First AlphaCode reads the two phrases.
alpha_code.read_string('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)
ell if a[0]==b[0] and len(a)==1:
    a.pop(0)
if len(b)==0:
    print("YES")
else:
    print("NO")
```

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

- 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

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

**DATA**
- GitHub
  - Problems
  - Solutions
- CodeContests
  - Problems
  - Solutions

**LEARNING**
- Pre-training
- Fine-tuning

**SAMPLING & EVALUATION**
- Codeforces
  - Problems
- Large set of potential solutions
- Filtering & clustering
- Large scale sampling
- Selected small set of candidates
- Execute & evaluate
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

Metadata (provided in the training set)

Polycarp must pay exactly n burles at the checkout ... (rest of the description)

**Training**
- **Metadata**: 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

**DATA**
- GitHub
  - Pre-training
  - CodeContests
    - Problems
    - Solutions
- Codeforces
  - Problems
- Large scale sampling

**LEARNING**
- Fine-tuning

**SAMPLING & EVALUATION**
- Large set of potential solutions
- Filtering & clustering
- Selected small set of candidates
  - Leverage program execution
  - Execute & evaluate
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)

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

<table>
<thead>
<tr>
<th>Pre-training dataset</th>
<th>10@1K</th>
<th>Solve rate</th>
<th>10@10K</th>
<th>10@100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>No pre-training</td>
<td>4.5%</td>
<td>7.0%</td>
<td>10.5%</td>
<td></td>
</tr>
<tr>
<td>GitHub (Python only)</td>
<td>5.8%</td>
<td>11.1%</td>
<td>15.5%</td>
<td></td>
</tr>
<tr>
<td>MassiveText</td>
<td>9.7%</td>
<td>16.1%</td>
<td>21.2%</td>
<td></td>
</tr>
<tr>
<td>GitHub (all languages)</td>
<td>12.4%</td>
<td>17.3%</td>
<td>21.5%</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

- 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

<table>
<thead>
<tr>
<th>Approach</th>
<th>Validation Set</th>
<th>Test Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10@1k</td>
<td>10@10k</td>
</tr>
<tr>
<td>9B</td>
<td>16.9%</td>
<td>22.6%</td>
</tr>
<tr>
<td>41B</td>
<td>16.9%</td>
<td>23.9%</td>
</tr>
<tr>
<td>41B + clustering</td>
<td>21.0%</td>
<td>26.2%</td>
</tr>
</tbody>
</table>
Breakdown on different algorithms

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

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
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

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.
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
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
```
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.

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

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

- **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

---

**Problem description**

```sql
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"
```
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 customers.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

```cpp
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

```python
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.
Self-debugging with different feedback formats

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
Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.

<table>
<thead>
<tr>
<th>Original Python</th>
<th>Revised Python #1</th>
<th>Revised Python #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Simple Feedback]</td>
<td>[Simple Feedback]</td>
<td></td>
</tr>
</tbody>
</table>

### Unit Test (UT) Feedback
Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.

<table>
<thead>
<tr>
<th>Original Python</th>
<th>Revised Python #1</th>
<th>Revised Python #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UT Feedback</td>
<td>UT Feedback</td>
<td></td>
</tr>
</tbody>
</table>

### Unit Test + Explanation (+Expl.)
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.

<table>
<thead>
<tr>
<th>Original Python</th>
<th>Revised Python #1</th>
<th>Revised Python #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Explanation</td>
<td>UT Feedback</td>
<td>[Simple Feedback]</td>
</tr>
</tbody>
</table>

### Unit Test + Trace (+Trace)
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.

<table>
<thead>
<tr>
<th>Original Python</th>
<th>Revised Python #1</th>
<th>Revised Python #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>UT Feedback</td>
<td>[Simple Feedback]</td>
</tr>
<tr>
<td>Revised Python #1</td>
<td>UT Feedback</td>
<td></td>
</tr>
<tr>
<td>Revised Python #2</td>
<td>UT Feedback</td>
<td></td>
</tr>
</tbody>
</table>
Application 3: text-to-Python generation

```python
# 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.

<table>
<thead>
<tr>
<th></th>
<th>Spider</th>
<th>Codex</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
<th>StarCoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>81.3</td>
<td>71.1</td>
<td>73.2</td>
<td>64.7</td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>81.3</td>
<td>72.2</td>
<td>73.4</td>
<td><strong>64.9</strong></td>
<td></td>
</tr>
<tr>
<td>+Expl.</td>
<td><strong>84.1</strong></td>
<td>72.2</td>
<td>73.6</td>
<td><strong>64.9</strong></td>
<td></td>
</tr>
</tbody>
</table>

(b) Results on TransCoder.

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

(c) Results on MBPP.

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

- 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.

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

- 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

(c) Results on MBPP.

<table>
<thead>
<tr>
<th>MBPP</th>
<th>Codex</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
<th>StarCoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>61.4</td>
<td>67.6</td>
<td>72.8</td>
<td>47.2</td>
</tr>
<tr>
<td>Simple</td>
<td>68.2</td>
<td>70.8</td>
<td>78.8</td>
<td>50.6</td>
</tr>
<tr>
<td>UT</td>
<td>69.4</td>
<td>72.2</td>
<td><strong>80.6</strong></td>
<td>52.2</td>
</tr>
<tr>
<td>+ Expl.</td>
<td>69.8</td>
<td><strong>74.2</strong></td>
<td>80.4</td>
<td>52.2</td>
</tr>
<tr>
<td>+ Trace.</td>
<td><strong>70.8</strong></td>
<td>72.8</td>
<td>80.2</td>
<td><strong>53.2</strong></td>
</tr>
</tbody>
</table>
Self-debugging can be triggered with few-shot prompting

(a) Results on the Spider development set.

<table>
<thead>
<tr>
<th>Spider</th>
<th>Codex</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
<th>StarCoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>81.3</td>
<td>71.1</td>
<td>73.2</td>
<td>64.7</td>
</tr>
<tr>
<td>Simple</td>
<td>81.3</td>
<td>72.2</td>
<td>73.4</td>
<td>64.9</td>
</tr>
<tr>
<td>+Expl.</td>
<td>84.1</td>
<td>72.2</td>
<td>73.6</td>
<td>64.9</td>
</tr>
</tbody>
</table>

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

(c) Results on MBPP.

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

<table>
<thead>
<tr>
<th>TransCoder</th>
<th>Codex</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>80.4</td>
<td>89.1</td>
<td>77.3</td>
</tr>
<tr>
<td>Simple</td>
<td>83.4</td>
<td>89.1</td>
<td>78.2</td>
</tr>
<tr>
<td>+ Expl.</td>
<td>83.9</td>
<td>89.1</td>
<td>78.0</td>
</tr>
<tr>
<td>+ Trace.</td>
<td>83.9</td>
<td>89.1</td>
<td>78.4</td>
</tr>
</tbody>
</table>

- Self-debugging improvement is less significant w/o code execution, but can still bring up to 5% performance gain using Codex and GPT-4

(b) Results on MBPP

<table>
<thead>
<tr>
<th>MBPP</th>
<th>Codex</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>61.4</td>
<td>67.6</td>
<td>72.8</td>
</tr>
<tr>
<td>Simple</td>
<td>57.6</td>
<td>68.2</td>
<td>76.0</td>
</tr>
<tr>
<td>+ Expl.</td>
<td>64.4</td>
<td>68.2</td>
<td>76.0</td>
</tr>
<tr>
<td>+ Trace.</td>
<td>66.2</td>
<td>69.2</td>
<td>76.4</td>
</tr>
</tbody>
</table>

- Trace feedback simulates the execution outcome and provides helpful information for self-debugging

Self-debugging without code execution
**Discussion: valid external feedback is crucial for self-correction**

<table>
<thead>
<tr>
<th></th>
<th>GSM8K</th>
<th>CommonSenseQA</th>
<th>HotpotQA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPT-3.5</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Prompting</td>
<td>75.9</td>
<td>75.8</td>
<td>26.0</td>
</tr>
<tr>
<td>Self-Correct (Oracle)</td>
<td>84.3</td>
<td>89.7</td>
<td>29.0</td>
</tr>
<tr>
<td><strong>GPT-4</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Prompting</td>
<td>95.5</td>
<td>82.0</td>
<td>49.0</td>
</tr>
<tr>
<td>Self-Correct (Oracle)</td>
<td>97.5</td>
<td>85.5</td>
<td>59.0</td>
</tr>
</tbody>
</table>

- 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

---

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

Problem Reduction (Syntactic Parsing)

- What was produced by a art director that M1 and M2 employed and was directed by M3
- What was produced by N1 that N2 employed and was directed by N3

Dynamically Select Exemplars for Each Subproblem

- Retrieve using constituent from decomposition
- Selected exemplar

Exemplar Pool

M1 and M2 employed

Who edited a film that M1 and M2 produced

Sequentially Solve Subproblems

Exemplars

Q: Who edited a film that M1 and M2 produced
A: <Exemplar Answer> ...

Subproblems

Q: What was directed by M3
A: <Predicted Answer> ...

Input

Q: What was produced by an art director that M1 and M2 employed and was directed by M3

Output

A: SELECT DISTINCT WHERE {
  ?x0 produced_by ?x1 . ?x1 a art_director .
  M1 employed ?x1 . M2 employed ?x1 .
  ?x0 directed_by M3 }

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, Liu, Song, Towards Synthesizing Complex Programs from Input-Output Examples, ICLR 2018.
Demonstration of the symbolic machine

Input
jump around right

Input Queue
jump around right
Stack
Memory

Step 1:
SHIFT

Input Queue
around right
Stack
[jump]
Memory

Step 2:
REDUCE [JUMP]

Input Queue
around right
Stack
[jump]
Memory

Step 3:
PUSH

Input Queue
around right
Stack
[JUMP]
Memory

Step 4:
SHIFT

Input Queue
right
Stack
around
[JUMP]
Memory

Step 5:
PUSH

Input Queue
right
Stack
around
[JUMP]
Memory

Step 6:
SHIFT

Input Queue
EOS
Stack
right
around
[JUMP]
Memory

Step 7:
REDUCE [RETURN]

Input Queue
EOS
Stack
[RETURN]
around
[JUMP]
Memory

Step 8:
CONCAT_M [0]

Input Queue
EOS
Stack
around
[RETURN]
Memory

Step 9:
POP

Input Queue
EOS
Stack
[JUMP] around
[RETURN]
Memory

Step 10:
CONCAT_M [2, 0]

Input Queue
EOS
Stack
around
[RETURN, JUMP]
Memory

Step 11:
CONCAT_S [1, 1, 1]

Input Queue
EOS
Stack
[RETURN, JUMP, RETURN, JUMP, RETURN, JUMP, RETURN, JUMP]
Memory

Step 12:
FINAL

Output
RETURN, JUMP,
RETURN, JUMP,
RETURN, JUMP,
RETURN, JUMP,
Can LLMs improve compositional generalization without symbolic execution?

<table>
<thead>
<tr>
<th>Method</th>
<th>Standard prompting</th>
<th>Chain-of-Thought</th>
<th>Least-to-Most</th>
</tr>
</thead>
<tbody>
<tr>
<td>code-davinci-002</td>
<td>16.7</td>
<td>16.2</td>
<td>99.7</td>
</tr>
<tr>
<td>text-davinci-002</td>
<td>6.0</td>
<td>0.0</td>
<td>76.0</td>
</tr>
<tr>
<td>code-davinci-001</td>
<td>0.4</td>
<td>0.0</td>
<td>60.7</td>
</tr>
</tbody>
</table>

SCAN length split

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

Wei et al., Chain of Thought Prompting Elicits Reasoning in Large Language Models, NeurIPS 2022.
Least-to-most prompting

Problem Reduction

Q: It takes Amy 4 minutes to climb to the top of a slide. It takes her 1 minute to slide down. The water slide closes in 15 minutes. How many times can she slide before it closes?

A: To solve “How many times can she slide before it closes?”, we need to first solve: “How long does each trip take?”

Stage 1

Stage 2

Sequentially Solve Subquestions

It takes Amy 4 minutes to climb to the top of a slide. It takes her 1 minute to slide down. The slide closes in 15 minutes.

Q: How long does each trip take?

A: It takes Amy 4 minutes to climb and 1 minute to slide down. 4 + 1 = 5. So each trip takes 5 minutes.

Subquestion 1

Append model answer to Subquestion 1

It takes Amy 4 minutes to climb to the top of a slide. It takes her 1 minute to slide down. The slide closes in 15 minutes.

Q: How long does each trip take?

A: It takes Amy 4 minutes to climb and 1 minute to slide down. 4 + 1 = 5. So each trip takes 5 minutes.

Subquestion 2

Q: How many times can she slide before it closes?

A: The water slide closes in 15 minutes. Each trip takes 5 minutes. So Amy can slide 15 \div 5 = 3 times before it closes.
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”.

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.

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
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
Our approach: problem decomposition with syntactic parsing

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

Problem Reduction (Syntactic Parsing)

What was produced by a art director that M1 and M2 employed and was directed by M3 → LM → What was produced by (a art director) that (M1 and M2) employed and was directed by (M3)

What was produced by N1 that N2 employed and was directed by N3 → LM → What was produced by (N1 that (N2 employed)) and was directed by N3

Dynamically Select Exemplars for Each Subproblem

Retrieve using constituent from decomposition

Exemplar Pool

M1 and M2 employed

Selected exemplar

Who edited a film that M1 and M2 produced

Sequentially Solve Subproblems

Exemplars

Q: Who edited a film that M1 and M2 produced
A: <Exemplar Answer> ...

Subproblems

Q: What was directed by M3
A: <Predicted Answer> ...

Input

Q: What was produced by an art director that M1 and M2 employed and was directed by M3

LM

Output

A: SELECT DISTINCT WHERE {
    ?x0 produced_by ?x1 . ?x1 a art_director .
    M1 employed ?x1 . M2 employed ?x1 .
    ?x0 directed_by M3 }

## Evaluation on CFQ

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

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

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

Parse: \[
\text{say ( agent = James, ccomp = like ( agent = manager, ccomp = appreciate ( agent = Aiden, ccomp = believe ( agent = Emily, ccomp = post ( recipient = * girl, theme = cake ( nmod . beside = table ), agent = Olivia ) ) ) ) ) DONE}
\]

<table>
<thead>
<tr>
<th></th>
<th>Gen.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fully Supervised</strong></td>
<td></td>
</tr>
<tr>
<td>LeAR (Liu et al., 2021)</td>
<td>97.7</td>
</tr>
<tr>
<td>T5-base (Qiu et al., 2022a)</td>
<td>89.8</td>
</tr>
<tr>
<td>T5-base + CSL (Qiu et al., 2022a)</td>
<td><strong>99.5</strong></td>
</tr>
<tr>
<td><strong>Prompting</strong></td>
<td></td>
</tr>
<tr>
<td>(Ours) Dynamic Least-to-Most</td>
<td>99.2</td>
</tr>
</tbody>
</table>

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 \( a_1, a_2, \ldots, a_n \). 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 \( a_1 \cdot a_2 \cdots a_k = a_{(k+1)} \cdots a_n \). 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:
Analogical prompting for code generation

Generated example problems can be much simpler than the testing problem
### Evaluation on Codeforces problems

<table>
<thead>
<tr>
<th>Prompting Method</th>
<th>GPT3.5-turbo-16k</th>
<th>GPT4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acc@1</td>
<td>Acc@10</td>
</tr>
<tr>
<td>0-shot</td>
<td>8%</td>
<td>24%</td>
</tr>
<tr>
<td>0-shot CoT</td>
<td>9%</td>
<td>27%</td>
</tr>
<tr>
<td>3-shot CoT</td>
<td>11%</td>
<td>27%</td>
</tr>
<tr>
<td><strong>Ours: Self-generated Exemplars</strong></td>
<td>13%</td>
<td>25%</td>
</tr>
<tr>
<td><strong>Ours: Self-generated Knowledge + Exemplars</strong></td>
<td><strong>15%</strong></td>
<td><strong>29%</strong></td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Prompting Method</th>
<th>GSM8K Accuracy</th>
<th>MATH Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPT3.5-turbo</td>
<td>text-davinci-003</td>
</tr>
<tr>
<td>0-shot</td>
<td>75.0%</td>
<td>14.8%</td>
</tr>
<tr>
<td>0-shot CoT</td>
<td>75.8%</td>
<td>50.3%</td>
</tr>
<tr>
<td>5-shot CoT</td>
<td>76.7%</td>
<td>54.0%</td>
</tr>
<tr>
<td><strong>Ours: Self-generated Exemplars</strong></td>
<td><strong>77.8%</strong></td>
<td><strong>61.0%</strong>†</td>
</tr>
</tbody>
</table>

Mathematical reasoning

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

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
• 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