

Prompting, Debugging and Innovation for Code

Ex 1 - Prompting Design

See 4 attached tables for results for chain of thought and chain of thoughts with self debugging. Both prompting methods were tested on "codellama/CodeLlama-7b-Instruct-hf" and "stabilityai/stablelm-zephyr-3b".

For metric pass@k: k=5

	zephyr-3b	zephyr-3b	zephyr-3b	Llama-7b	Llama-7b
Chain of Thought	✓	✓	x	✓	✓
Structured chain of thought	x	x	✓	x	x
Self debugging	x	✓	x	x	✓
pass@k	0.5	0.4	0.3	0.4	0.3

Zephyr-3B runtime for all problems was approximately 30 minutes.

LLaMA 7B took about 1 hour to generate code for 10 problems, repeated 5 times. During this process, I had to restart multiple times due to unsatisfactory results, and once the model got stuck in an infinite loop. Additionally, I had to explicitly include the name of the required function to ensure the tests executed correctly.

Surprisingly, Zephyr produced significantly better results compared to LLaMA, despite this version of LLaMA being specifically trained for programming tasks.

A closer inspection of the outputs from both models reveals some noticeable trends:

- LLaMA often got stuck while generating the chain of thought (CoT) and repeated itself multiple times. This contributed to the longer execution times, as its responses were considerably longer.

- Because LLaMA frequently got stuck generating the CoT, it often failed to produce the required function at all.
- Zephyr-3B, while generally demonstrating better problem understanding, frequently struggled with using the correct number of parameters and return values. To investigate this further, I also tested structured-chain-of-thought prompting. However, Zephyr-3B had significant trouble with this approach; when prompted this way, it mostly generated only code without any accompanying reasoning. This issue can be somewhat mitigated (e.g., in Exercise 2) by explicitly specifying the correct number of input and output parameters.

Another interesting observation is that both models performed best on Problem 8, which involves calculating the square of a given input—a relatively simple task. However, for more complex problems like finding the minimum cost path (a fairly common algorithm), neither model produced the correct answer even once.

When running Zephyr-3B with CoT and self-debugging prompts, the keyword *self-debugging* actually led to worse results, as the model focused on generating test cases instead of code that could be tested. The phrasing, “**Before generating code, use self-debugging to improve your result,**” could be clearer—the model should generate code but apply self-debugging internally before returning the final code to the user.

The same issue applies to LLaMA-7B when using self-debugging.

Ex2 - Debugging

I explored the following two problems, which the generated code generally had issues with:

1. “Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix `cost[][]` and a position (m, n) in `cost[][]`.”
2. “Write a function to find the similar elements from the given two tuple lists.”

For both of these problems in ex1 there were a lot of type mismatches and wrong number of args. Therefore, the first change i did was to include the number of arguments in my prompt:

1.

Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix `cost[][]` and a position (m, n) in `cost[][]`.

Write a Python function named `min_cost` that solves the problem.
 The function requires 3 input parameters and 1 return values.
 Use a chain of thought to explain the solution before giving the code.

Format your response with:

Chain of Thought:

<your reasoning here>

Python:

```
```python
```

<your function here>

```
```
```

Output nothing else.

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

2.

Write a Python function named `similar_elements` that solves the problem.

The function required 2 input parameters and 1 return values Use a chain of thought to explain the solution before giving the code.

Format your response with:

Chain of Thought:

<your reasoning here>

Python:

```
```python
```

<your function here>

```
```
```

Output nothing else.

However, this did not fix the type mismatch, which was the next thing I included the prompt.

1. Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix cost[][] and a position (m, n) in cost[][].

Write a Python function named `min_cost` that solves the problem.

The function requires 3 input parameters of type one matrix and two ints and 1 return values of type int.

Use a chain of thought to explain the solution before giving the code.

Format your response with:

Chain of Thought:

<your reasoning here>

Python:

```
```python
```

<your function here>

```
```
```

Output nothing else.

2.

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

Write a Python function named `similar_elements` that solves the problem.

The function required 2 input parameters of type tuple of int and 1 return values of type tuple of int

Use a chain of thought to explain the solution before giving the code.

Format your response with:

Chain of Thought:

```

<your reasoning here>
Python:
```python
<your function here>
```

Output nothing else.

```

However, the model struggled to fully understand Problem 1, as evidenced by the chain of thought it provided. Another challenge was its difficulty in correctly iterating over the given matrix.

For Problem 2, the model succeeded only about two out of five times using this kind of prompting.

Since Zephyr-3B is not specialized in solving programming problems, I somewhat accepted its inability to solve Problem 1. I did try once more by providing a detailed chain of thought, but realistically, this approach demands too much effort from the user, who would have to analyze the problem thoroughly before prompting.

```

Write a function to find the minimum cost path to reach (m, n) from (0, 0) for
the given cost matrix cost[][] and a position (m, n) in cost[][].

```

```

Write a Python function named `min_cost` that solves the problem.
The function requires 3 input parameters of type one matrix and two ints and 1
return values of type int
Use the following Chain of Thought to guide you.
0. Receive 3 arguments.
1. Compute the minimum cost to reach cell (m, n) from (0, 0) in a cost[][]
matrix, moving only right or down.
2. From any cell (i, j), we can go to (i+1, j) or (i, j+1).
3. The cost to reach (i, j) depends on the minimum cost to reach (i-1, j) or
(i, j-1).
4. Recurrence relation is dp[i][j] = cost[i][j] + min(dp[i-1][j], dp[i][j-1]).
5. Base cases is dp[0][0] = cost[0][0]. For first row dp[0][j] = dp[0][j-1] +
cost[0][j], first column dp[i][0] = dp[i-1][0] + cost[i][0].
6. Use bottom-up DP to fill a 2D dp table based on the recurrence.
7. Return dp[m][n] as the final minimum cost.
Don't forget the required amount of arguments is 3!Format your response with:
Python:
```python
<your function here>
```

Output nothing else.

```

However, even with stating 3 times that 3 arguments are required the generated code now only included one argument...

Finally, I compared the code generated by GPT-4.1-mini. Overall, GPT-4.1-mini demonstrated a better understanding of the problem and produced working code. This is not surprising, given that the GPT-4 model is trained extensively on coding data and likely has a significantly larger number of parameters.

Ex3 - Innovation

For my own strategy, I provided all the necessary parameters discussed in Exercise 2, but also included the model's previous output and the resulting error message (if any) as part of the prompt. This effectively forced a second attempt, giving the model a chance to improve its result. While similar in spirit to self-debugging, this method avoids the issues seen in Exercise 1, where simply prompting the model to "use self-debugging" often led to worse performance. In that sense, this approach is comparable to the concept of *agentic patching* introduced in the guest lecture.

However, in practice, the error messages returned were often vague — commonly just "AssertionError" — offering little to no insight into why the function failed. Occasionally, errors like "index out of range" or "wrong number of arguments" were returned, but even these rarely helped Zephyr-3B improve its output. This is likely due to the model's limited exposure to programming-specific training data.

As a final step, I re-evaluated the LLaMA-7B model using the same strategy — providing full parameter information and returning error messages — to see whether its performance could improve with this additional context. The overall results remained largely unchanged: assertion errors did not help improve solution quality. Interestingly, in cases where the model failed to generate any code and received the error message "<function> is not defined," it was able to correct this mistake on the next attempt. However, even in those instances, the resulting solution was still incorrect. When running LLaMA on Problem 2, there was a single occurrence where the combination of the error message and the revised prompt led to a correct solution. This improvement is likely coincidental, as it happened only once throughout all runs.

Conclusion

Overall, Zephyr-3B demonstrated more consistent task understanding and output stability than LLaMA-7B, despite the latter being specifically trained for programming tasks. This suggests that Zephyr's stronger instruction-following capability can, to some extent, compensate for its lack of code-specific pretraining. Nevertheless, both models failed to solve more complex algorithmic problems, such as the minimum cost path, indicating that smaller models still struggle with multi-step reasoning and dynamic programming logic. While iterative prompting and feedback mechanisms offer limited improvement, they cannot fully compensate for insufficient reasoning capacity or lack of coding-oriented pretraining.