Can Github issues be solved with Tree Of Thoughts?

Ricardo La Rosa
ricardo@larosa.dev

Corey Hulse
coreyohulse@gmail.com

Bangdi Liu
buddytt0915@gmail.com

Abstract

While there have been extensive studies in code generation by large language models (LLM), where benchmarks like HumanEval(Chen et al., 2021) have been surpassed with an impressive 96.3% success rate, these benchmarks predominantly judge a model's performance on basic function-level code generation and lack the critical thinking and concept of scope required of real-world scenarios such as solving GitHub issues. This research introduces the application of the Tree of Thoughts (ToT) (Yao et al., 2023b) language model reasoning framework for enhancing the decision-making and problem-solving abilities of LLMs for this complex task. Compared to traditional input-output (IO) prompting and Retrieval Augmented Generation (RAG) techniques, ToT is designed to improve performance by facilitating a structured exploration of multiple reasoning trajectories and enabling self-assessment of potential solutions. We experimentally deploy ToT in tackling a Github issue contained within an *instance* of the SWE-bench(Jimenez et al., 2024a).

However, our results reveal that the ToT framework alone is not enough to give LLMs the critical reasoning capabilities to outperform existing methods. In this paper we analyze the potential causes of these shortcomings and identify key areas for improvement such as deepening the thought process and introducing *agentic*(Ng, 2024) capabilities. The insights of this research are aimed at informing future directions for refining the application of ToT and better harnessing the potential of LLMs in real-world problem-solving scenarios.

1 Introduction

Tree of Thoughts (ToT) (Yao et al., 2023b) is a language model reasoning framework designed to enhance the autonomy and intelligence of language models (LMs) in decision-making and problem-solving tasks. ToT managed to outperform Input-Output prompting (IO), Chain of Thought (CoT) (Wei et al., 2022), and Self Consistency with CoT (CoT-SC) in several reasoning based tasks such as *Game of 24*, *Crosswords* and *Creative Writing*. Despite the promising results of ToT on these basic tasks, there is an absence of studies that apply ToT towards more complex tasks that more closely model the real-world. Our research aims to put this framework to test in one of the most challenging software engineering tasks for large language models: resolving GitHub issues. This task requires an overall sense of scope and understanding of the repository when making changes, which requires stronger critical reasoning skills than previous basic code generation tasks. We anticipate that the ToT method will outperform both IO prompting and Retrieval Augmented Generation (RAG) techniques in this task. This expectation is based on ToT's ability to instill a stronger ability in LLMS for decision-making, evaluating multiple reasoning paths and self-assessing choices to determine the subsequent course of action.

2 Related Work

Prior work on solving Github issues has been done by the Princeton NLP team as part of SWE-Bench(Jimenez et al., 2024a). They released two fine-tuned models, SWE-Llama 7B and SWE-Llama 13B based on CodeLlama(Rozière et al., 2023) with Retrieval Augmented Generation (RAG). Further work then was done with the introduction of SWE-agent(Yang et al., 2024) which is a large language model-based agent system that operates within an Agent-Computer Interface (ACI). Another recent work is LLM-Based Multi-Agent Framework for GitHub Issue ReSolution (MAGIS) (Tao et al., 2024) which introduces Multi-Agency whereby leveraging the collaboration of various agents with distinct roles in the planning and coding process to resolve GitHub issues.

3 Data

3.1 SWE-bench

We utilized the dataset provided by SWE-Bench(Jimenez et al., 2024a) as the basis for the experiments. SWE-bench is a benchmark for evaluating large language models on real world software issues collected from GitHub. Given a code-base and an issue, a language model is tasked with generating a code patch that resolves the described problem. With SWE-Bench, you can:

- Train or fine-tune a model with their pre-processed datasets.
- Run inference on existing models.
- Evaluate a model against the benchmark and determine the correctness of a solution proposed by the model.

This dataset is composed of a wide variety of tasks, such as filing a bug report or making a feature request, that the model will be charged with completing. The main similarity between these tasks is that they all require the model to generate a git patch to an existing code-base based on the problem statement of the Github issue. The revised code-base is then evaluated using the internal testing framework of the repository. If the proposed patch passes these tests then the model's proposed changes are considered successful and the task is counted as passed.

3.2 SWE-bench Lite

In order to reduce costs we used the SWE-bench_Lite(Jimenez et al., 2024b) dataset which is a canonical subset of SWE-bench that has been curated to make evaluation less costly. Instances from the original dataset that match the following criteria are not considered:

1. Include images, external links, references to specific commits, and references to other pull requests
2. Contain problem statements with fewer than 40 words
3. Edit more than one file
4. Have a gold patch with more than three edit hunks
5. Create or delete files
6. Contain tests with error messages checks

After filtering out the instances who violated the above standards, the result is a smaller dataset of 23 instances in the *dev* split and 300 instances in the *test* split.

3.3 Motivation

Traditional benchmarks in Natural Language Processing (NLP) often focus on relatively short input and output sequences that are not representative of real-world tasks.

Table 1: HumanEval Leaderboard

Model	Success Rate(%)
AgentCoder(GPT-4) (Huang et al., 2024)	96.3
LDB + Reflexion(GPT-3.5) (Zhong et al., 2024)	95.1
Language Agent Search Tree(GPT-4) (Zhou et al., 2023)	94.4
L2MAC(GPT-4) (Holt et al., 2024)	90.2

As shown in table 1 LLMs demonstrate remarkable performance on the HumanEval(Chen et al., 2021) benchmark. However, this benchmark exhibits several notable weaknesses: scope limited to function-level code generation, lack of diversity by focusing mainly on algorithmic tasks, and a lack of contextual and environmental interaction. In contrast, we considered that SWE-bench, emphasizes tasks that adequately model real-world scenarios where the interdependencies of the code base as a whole must be take into account when generating new patches, and the testing framework is able to use a built in testing framework to evaluate if the model's code correctly fits into the existing code base.

Table 2: SWE-bench Lite Leaderboard

Model	Success Rate(%)
SWE-agent + GPT 4	17.00
SWE-agent + Claude 3 Opus	11.67
RAG + Claude 3 Opus	4.00
RAG + GPT4	2.67
RAG + Claude 2	2.00
RAG + SWE-Llama 13B	1.67
RAG + SWE-Llama 7B	1.33
RAG + GPT 3.5	0.33
GPT 4	0.00
ChatGPT 3.5	0.00

Table 2 paints a different picture. Even with Retrieval Augmented Generation, performance sees limited improvement due to the difficulties that

LLMs face when handling long context inputs, notably in tasks like resolving GitHub issues at the repository level, where using large portions of the repository as input is impractical.

4 Models

We utilized three open-source models for this research. The first model was **CodeLlama 34B**. (Jimenez et al., 2024a) highlighted that variants of CodeLlama were not capable of following detailed instructions in order to make repository-wide code edits, and typically resorted to outputting placeholder responses or unrelated code. To address this issue, they performed supervised fine-tuning on the 7 billion-parameter and 13 billion-parameter variants. The resulting models were shown to be highly successful at maintaining specialized repositories and could be run on consumer hardware to resolve GitHub issues. Based on these observations, we opted for the 34 billion parameter version of CodeLlama, which had been quantized to 4-bit precision and fine-tuned using a select portion of the SWE-bench dataset.

Acknowledging the highlighted limitations, we broadened our approach by integrating three larger LLMs of 56 billion and 70 billion parameters: **Mixtral-8x7B**, **Llama2 70B** and **Llama3 70B Instruct**. We anticipated these models would be able to efficiently generate patches in the unified diff format, when given few-shot examples.

5 Methods

5.1 Baselines

We use a standard input-output (IO) prompt with 5 in-context examples.

5.2 Tree of Thoughts setup

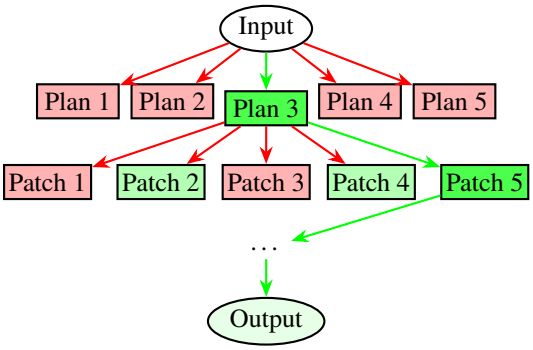

Figure 1: ToT setup with $n = 5$, $k = 5$ and $b = 1$

As shown in figure 1 we built a ToT with depth $d = 2$ with one intermediate thought step. The input is a Chain of Thought style prompt where the model is asked to generate n plans and votes for the best one, then similarly generate k patches based on the best plan. In order to guarantee plan diversity we increased the temperature to a determined value t. The last step is to rate the patches: the patch with the highest score is chosen. The breadth limit is always set to $b = 1$ as a consequence the breadth-first search (BFS) only maintains the most promising state per step in a greedy approach.

We forked the official Tree of Thoughts Github repository(Yao et al., 2023a) and created a branch to add a new `Task` class called `SWETask`. This class was designed to solve the *instances* within the SWE-bench Lite dataset, based on the mentioned ToT setup.

5.2.1 Prompting

A zero-shot vote prompt was used to sample votes for plan selection and zero-shot score prompt is used to make patches scores. Example of prompts used are in appendix A.

5.3 Metrics

We used the SWE-bench metrics which is the percentage of task instances that are correctly solved by the model. In order to judge whether or not the model correctly solves an individual task we will use the following scoring method:

Table 3: SWE-bench Evaluation

Result	Score
Fails at any step	0
Correctly completes all steps	1

If the evaluation produces incorrect outputs for any step in the task then the entire task is treated as a failure. In order to be counted as a success, the evaluation must succeed at every step of and pass all the associated test cases. Taking the total number of successful tasks over the number of attempted tasks will serve as our primary metric.

5.3.1 Evaluation

We will execute generated patches against the corresponding task instances of the benchmark to determine whether or not it resolves the associated Github issue. The SWE-bench refers to such patch

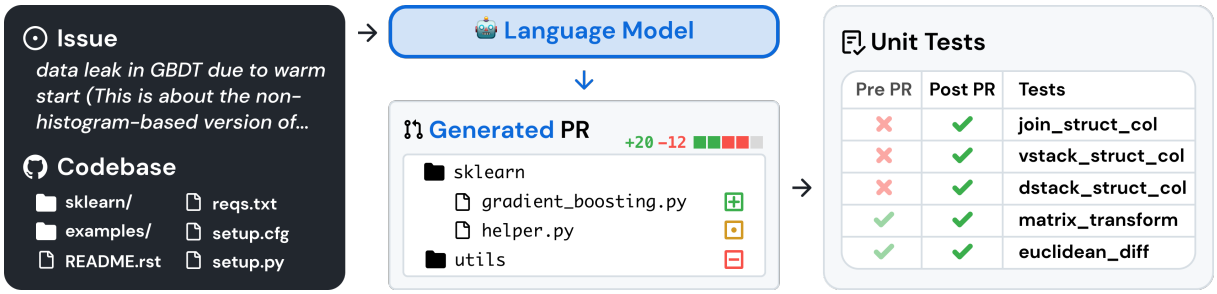

Figure 2: SWE-bench Evaluation (Jimenez et al., 2024a)

generations as the *prediction patch*. The benchmark framework performs the following steps for testing:

1. Installs repository at base commit according to task instructions
2. Applies test patch, *prediction patch* and run tests
3. Checks prediction logs to see the pass/fail status of each test

5.4 Fine-tuning

We loaded the CodeLlama model with *FastLanguageModel* loader from the unsloth(Han and Han, 2023) library that extends Hugging Face's Parameter-Efficient Fine-Tuning (PEFT) (Mangrulkar et al., 2022) which provides several performance optimizations for training and inference.

Table 4: CodeLlama training parameters.

Parameter	Value
Quantization	4-bit
Learning rate	2e-4
Optimizer	AdamW 8-bit
Warmup ratio	0.05
Number of epochs	1
Max seq length	16384
Weight decay	0.01
Per device train batch size	4
Grad accumulation Steps	4

The trainer was setup with the parameters shown in table 4. Additionally, we applied the following techniques:

1. Supervised training: Supervised Fine-tuning Trainer *SFTTrainer* from Hugging Face's Transformer Reinforcement Learning (TRL) (von Werra et al., 2020) library.
2. Quantization: We used a pre-quantized 4-bit model to reduce memory usage.
3. Lower Ranking Adaptation(LoRA): Using QLoRA(Dettmers et al., 2023) we only updated 1 to 10% of all parameters.
4. Rotary Positional Embedding(RoPE) Scaling: the of RoPE (Su et al., 2022) scaling using Kiao Ken's method(Ken, 2023) made the context window flexible.

5.5 Inference via API

To mitigate the computational challenges and time constraints associated with the experiments the Groq API was utilized for both **Mixtral-8x7B** and **Llama2 70B**, achieving an impressive average throughput of 500 tokens per second; A large boost in the generation speed of patches. Looking forward, we are encouraged by the expected arrival of Language Processing Units (LPU) (Abts et al., 2022) Inference Engines which promise to significantly advance the field by facilitating the adoption of frameworks like ToT across a broad spectrum of applications.

6 Results

We conducted the experiments using a subset of 100 instances, representing 33% of the full dataset from the SWE-bench_Lite benchmark. This sample size was selected to provide a representative snapshot of ToT's performance, while also considering the computational constraints and resource requirements associated with evaluating the models on the complete dataset.

As shown in table 5 the three models performed as poorly as input-output (IO) prompting. We observed that while all the generated git patches were syntactically correct (demonstrating the in-context learning capabilities of the LMs), none of them were able to be successfully applied in practice to

Table 5: SWE-bench Lite results.

Model	Success Rate (%)	
	IO	ToT
CodeLlama 34B	0	0
Llama-2 70B	0	0
Mixtral-8x7B	0	0
+Llama-3 70B Instruct	0	0

result in a proper bug fix. In other words, all the proposed patches were *rejected* and could not be integrated into the target code base.

To test our last model **Llama-3 70B Instruct**, we ran a bigger subset of 150 instances (50% of dataset) which yield similar score but in this case 10% of generated patches were successfully applied but failed some tests. In other words, 10% of proposed patches were *accepted* but did not pass the test suite.

7 Analysis

The findings indicate that Tree of Thoughts (ToT) was not effective for the specific task that was examined. This research acknowledges several limitations within the experimental setup that may have impacted the efficacy of the ToT framework. Specifically, we identify the following weaknesses:

1. The use of a relatively shallow thought process tree, consisting of only two thought steps. This was problematic because it did not allow the complex tasks asked of the model to be decomposed into smaller, more manageable sub-tasks that individual reasoning steps could be applied to.

2. Providing only the repository name, problem statement, and git commit is insufficient for the model to fully comprehend and address the requirements of the task. However, as shown in table 2 many complex tasks cannot be accomplished through a single step or a solitary tool invocation. Even with RAG conducting similarity searches over the contents of the sizable Git repository proved to be an ineffective way of supplying the model with the necessary task-specific information. Allowing the model access to explore and examine the file contents of the associated GitHub repository could have facilitated a more informed and accurate approach.

3. The validation of the individual thought steps was conducted through a voting mechanism, rather than comparing the outputs to a defined *ground truth*. Incorporating a symbolic validation component, similar to the utilized by SWE-bench itself, could have provided a more robust means of evaluating the correctness of the generated patches.

8 Conclusion

Despite the limitations observed in the current experimental setup, we hypothesize that with a redesigned approach and addressing the identified weaknesses, the potential of the Tree of Thoughts (ToT) framework could be better realized and more effectively demonstrated. Specifically, a setup with *agentic design patterns*(Ng, 2024) where the large language model is:

1. Leveraged to autonomously break down the objective into smaller sub-tasks. This allows the model to dynamically determine the optimal sequence of steps required to accomplish the resolution of the GitHub issue.

2. Provided with access to tools via *function calling*(Kim et al., 2024). This enables the model to independently make requests for the purpose of gathering information, taking action, or manipulating data. For example, being able to do a code `search` in the git repository and `open` sections of a file like the Agent-Computer Interface (ACI) introduced by SWE-agent.

3. Provided with code search that is project structure aware like AutoCodeRover(Zhang et al., 2024). Instead of searching over files by plain string matching, the model can search for relevant code context (functions/classes) in the syntax tree.

Additional improvements are related to the patch generation, such as:

1. Freeing the large language model from the direct responsibility of generating the code patches. Instead, leveraging the use of tools via function calling to handle the patch generation, allowing the model to focus on planning, debugging and code generation.

2. Incorporating a reliable ground truth for validating the generated patches.

Finally, since the model no longer generates unified patches, fine-tuning as an optimization strategy becomes less effective.

Known Project Limitations

Speed and Cost: It is important to note that while ToT may enhance decision-making and problem-solving capabilities of large language models, this sophistication can result in slower processing times due to the additional computational steps involved. The evaluation of multiple reasoning paths and self-assesses choices, inherently demands more resources and compute time, including a notable rise in prompt and generation tokens. However, the trade-off for this slower speed and higher cost is a potential increase in the accuracy and relevance of the outcomes, particularly in complex tasks. Adjustments to the framework's parameters can offer some mitigation of these issues, allowing users to make a balance between speed/cost and accuracy.
Search Methods: This research leverages the use of classical search algorithms, such as Breadth-First Search (BFS) and the current setup can be considered a form of heuristic search, akin to the A* algorithm, where the heuristic at each search node is provided by the large language model's own self-assessment of the generated thought. While this search strategy is straightforward to implement, it represents a relatively naive approach. We anticipate that the use of more advanced search algorithms, such as Monte Carlo Tree Search (MCTS), could potentially yield improved results. This expectation is informed by prior work, such as the research conducted by (Hao et al., 2023) with Reasoning viA Planning.

Authorship Statement

Ricardo La Rosa conceptualized the core experiments and hypothesis, conducted the majority of the experiments, analyzed the data, and wrote the core of the manuscript. Corey Hulse helped conduct validation of the experiments and oversaw final edits of deliverables. Bangdi Liu provided critical feedback, and assisted with revisions to the manuscript and previous deliverables. All authors read and approved the final version of the manuscript.

Acknowledgements

We would like to thank Christopher Potts, Petra Parikova, and our course facilitator, Jonathan Gomes Selman, for their valuable contributions and support throughout the XCS224U course. We are grateful for their generous availability and support.

References

Dennis Abts, Garrin Kimmell, Andrew Ling, John Kim, Matt Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, and Jeremy Fowers. 2022. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 567–580, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, and Fotios Chantzis. 2021. Evaluating large language models trained on code.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Daniel Han and Michael Han. 2023. Unsloth: 30x faster llm training.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2024. L2mac: Large language model automatic computer for extensive code generation.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024a. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024b. SWE-bench Lite Dataset. Number of rows: 323, File size: 1.29 MB.

Kaio Ken. 2023. Things i'm learning while training superhot. https://kaiokendev.github.io/til.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An llm compiler for parallel function calling.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft.

Andrew Ng. 2024. Agentic design patterns. https://www.deeplearning.ai/the-batch/issue-244.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2022. Roformer: Enhanced transformer with rotary position embedding.

Wei Tao, Yucheng Zhou, Zhang Wenqiang, and Yu Cheng. 2024. MAGIS: Llm-based multi-agent framework for github issue resolution.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, and Shengyi Huang. 2020. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent computer interfaces enable software engineering language models.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Official repo of tree of thoughts. https://github.com/princeton-nlp/tree-of-thought-llm.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023b. Tree of Thoughts: Deliberate problem solving with large language models.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement.

Lily Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models.

A Prompts

```
plan_prompt = '''Given the Repository url, Base commit
      and Problem statement of a github issue. Please
      write a plan to solve it.
Your output must be of the following format:

Plan:
Your plan here.

{input}
'''
```

Snippet 1: Plan prompt

```
patch_prompt = '''Given the Repository url, Base commit,
      Problem statement of a github issue and a plan.
      Please write a correct git patch to solve it.

Your output must be of the following format:

Patch:
```diff
Your patch here.
```

The patch file should be in the unified diff format.
      Example:

```diff
diff --git a/file.py b/file.py
--- a/file.py
+++ b/file.py
@@ -1,27 +1,35 @@
 def euclidean(a, b):
-    while b:
-        a, b = b, a % b
-    return a
+    if b == 0:
+        return a
+    return euclidean(b, a % b)
```

{input}
'''
```

Snippet 2: Patch prompt

```
vote_prompt = '''Given an instruction and several choices
      , decide which choice is most promising. Analyze
      each choice in detail, then conclude in the last
      line "The best choice is {s}", where {s} the
      integer id of the choice.'''
```

Snippet 3: Vote prompt

```
score_prompt = '''Analyze the following patch, then at
      the last line conclude "Therefore the correctness
      score is {s}", where {s} is an integer from 1 to
      10.'''
```

Snippet 4: Score prompt