DATA410 Final Report:
Large Language Models are Zero-Shot Short-Coders

Sangmin Hwang **Heechul Ryu** **Jeongeun Lee** **Boyoung Kim**

Abstract

In this study, we aimed to enhance the user experience in code recognition through the utilization of a language model (LLM) for code generation. Our focus encompassed optimizing convenience, performance, and cost efficiency. Employing prompt tuning to accommodate various programming languages, we sought to minimize code length without compromising performance. Through a combination of self-planning and in-context learning methodologies, we conducted experiments involving planning, implementation, and summarization steps. While achieving a commendable 22% reduction in code length with sustained performance, challenges in the planning phase necessitate further exploration and analysis. This research highlights the need for diverse approaches in future investigations, presenting opportunities for advancement.

1 Introduction

In recent years, LLMs demonstrates impressive code generation abilities, attracting attention from various fields. Among code generation, we specifically paid attention to "short code generation". We focused on short codes for two reasons. The first is the utility of short codes, and the second is how future LLMs will be used.

According to the survey [1], the number of input and output tokens is the most adopted as LLM's pricing metrics. This trend in the industry shows that programmers need to be concerned about code length when using LLMs for code generation.

The main reasons why we need to create short code are as follows: Firstly, because of ease of understanding and maintenance, and because of optimization of resource usage, such as more efficient compilation time. However, we also thought that "generating excessively short code" should be avoided. In the case of Python miniifiers or "short coding" challenges that "compress" the code, there is a disadvantage that the code becomes difficult for humans to understand.

Therefore, we wanted to use the capabilities of LLM to generate code as short and concise as possible while avoiding these problems. As a way to generate short code using LLM, we chose the prompt engineering method. This is because we believe that the prompted engineering approach is the most efficient and economical way to generate short code for LLM. We also expect the benefit of the prompt approach to be able to easily switch between languages in the field of code generation.

2 Related Work

code generation Conventional methods of code generation rely on supervised learning techniques. With the advent of pre-training methods, models like CodeT5 [2] and UniXcoder [3] have incorporated pre-trained models for the task of code generation. The paper that introduced GPT-3 [4] noted the limitations of existing fine-tuning approaches and the need for few shot transfers of large language models. In 2023, Llama, a large language model that used public datasets for learning, was released [5]. And there have been efforts to create a model specialized for code generation tasks by fine tuning LLM with code data, a representative example is Code Llama [6].

prompt techniques Prompting methods can sometimes work better than fine tuning methods, [7] showed that a prompted tuning approach to LLM performed better than a model fine-tuned with medical knowledge in the medical knowledge domain. Few-shot prompting [8] is a technique that emerged as the number of model parameters exploded. Instead of fine-tuning a separate language model for each new task, prompting can be utilized by simply providing the model with a limited number of input-output examples that illustrate the task. "Emergent Abilities of Large Language Models" [9] showed that when the size of the language model exceeds a certain level, zero shot and few shot performance suddenly increases. In addition, it was shown that the performance of the language model can be improved by performing chain of thought prompts or instruction tuning, and that performance is also improved on untrained tasks.

3 Method

We propose four methodologies to generate concise code while effectively reflecting complex intentions, as shown in Figure 1. The prompts used in the actual implementation of the methodology can be found in the Appendix A.

3.1 2-stage Summarization

The code summarization process is divided into two stages: (1) Implementation stage, where the code is implemented to reflect the given intent, and (2) Summarization stage, where the code from the implementation stage is compressed. In the summarization stage, various prompt engineering techniques, including zero-shot, one-shot with In-context examples, and prompting with strategies helpful for summarization, are applied to enhance the summarization process.

In-context examples For one-shot 2-stage summarization, we use in-context example that contains code summarization results. We utilized the GPT-4 API to generate summary results for code implementation problems crawled from LeetCode, resembling the format of the evaluation dataset. Subsequently, the pairs of code summaries before and after the summarization were employed as In-Context examples. While considering options such as Python Minifier or Human Annotation for generating summary results, LLM-generated summary were found to exhibit superior quality, leading to their selection. See Appendix B for the in-context examples used in the prompts.

Concrete strategies Inspired by the Chain-of-Thought prompts[10], we provide prompts with detailed techniques that could be useful for code summarization, instead of in-context examples. These techniques were derived from search results based on keywords such as 'Python Short coding tips' on the web. The strategies include transforming for-loops into list comprehensions, using dictionaries, and other similar approaches. See Appendix B for examples of the strategies used in the prompts.

3.2 1-stage Summarization

Unlike the previous 2-stage summarization method, this methodology adds a constraint to generate concise code simultaneously with the implementation, aiming to produce short code in a single step.

3.3 3-stage Planning Summarization

We propose this method to create well-reflected code implementations for complex intents using the Self-Planning approach introduced in [11]. Proceeding similarly to the 2-stage Summarization methodology, this approach refines the Implementation stage by dividing it into (1) the Planning stage, which involves transforming given requirements into a step-by-step plan in natural language, and (2) the Implementation stage, which prompts the model with this plan alongside the original implementation prompts. Due to the absence of In-Context examples in the original paper, we conducted zero-shot prompting in both the Planning and Implementation stages.

3.4 2-stage Planning Summarization

The methodology integrates the Planning-Implementation stages into a single step for code summarization, as opposed to the previous 3-stage Planning Summarization approach. In this method, the model is prompted to plan the code implementation and then proceed to implement the functions, with the prompt guiding it to output only the final implementation.

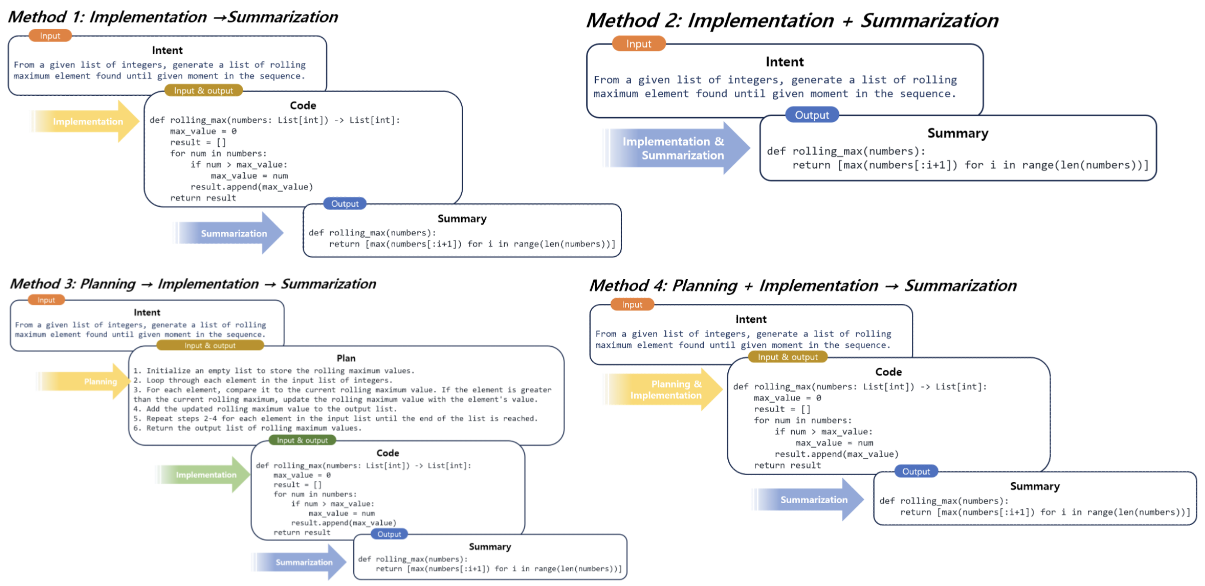
4 Experiment

4.1 Experiment Setup

Datasets and Model In this study, we utilized the HumanEval dataset[12] for evaluation, along with the CodeLLaMA-34b-instruct model.

The HumanEval dataset serves as a benchmark for evaluating the model's ability to generate functional code based on provided function descriptions. Comprising 164 Python programming problems, it spans diverse domains, including language understanding, algorithms, and elementary mathematical operations. Each problem includes a function description, the target function for implementation, and a set of input-output pairs for evaluation. Notably, some problems mirror typical IT technical interview questions.

To address the tasks of code summary, code generation, and the interpretation of code into nat-



From a given list of integers, generate a list of rolling maximum element found until given moment in the sequence.

```
def rolling_max(numbers: List[int]) -> List[int]:
    max_value = 0
    result = []
    for num in numbers:
        if num > max_value:
            max_value = num
        result.append(max_value)
    return result
```

```
def rolling_max(numbers):
    return [max(numbers[:i+1]) for i in range(len(numbers))]
```

```
1. Initialize an empty list to store the rolling maximum values.
2. Loop through each element in the input list of integers.
3. For each element, compare it to the current rolling maximum value. If the element is greater
than the current rolling maximum, update the rolling maximum value with the element's value.
4. Add the updated rolling maximum value to the output list.
5. Repeat steps 2-4 for each element in the input list until the end of the list is reached.
6. Return the output list of rolling maximum values.
```

Figure 1: Overall methodology.

ural language, including Self-planning, we employed the CodeLLaMA-34b-instruct model. We utilized a reference API provided by Replicate, and the parameter configuration for CodeLLaMA-34b-instruct was set with Greedy-decoding, using top p=1.0 and temperature=0. This parameter choice aimed to optimize the decoding process during both model training and evaluation. Refer to Appendix D for details on other parameters.

Evaluation The evaluation process is guided by two primary criteria. Firstly, consideration is given to the code's length. During the code generation process, a tendency was observed to generate non-functional elements, such as typing or annotations, which were unrelated to the code's operation. Consequently, the length measurement focused solely on portions directly associated with the code's functionality, excluding extraneous elements.

$$\textbf{pass@k} := \mathbf{E}_{\text{problems}}\left[1 - \frac{{}_{n-c}C_k}{{}_{n}C_k}\right] \qquad (1)$$

The second criterion centers on code performance. This aspect is assessed using the Pass@k metric above. Pass@k involves randomly selecting k test cases and computing the average probability that at least one of them is correct. Typically, values for k include 1, 10, and 100. A lower k value yields a more precise evaluation, while a higher k value enhances evaluation robustness.

4.2 Result and Analysis

Quantitative evaluation results for the experiments are presented in Table 1 and Table 2. Examples for the analysis described in the following are all available in Appendix B.

Method	**pass@1**	**pass@10**
Baseline	0.5776	0.8617
2-stage$_{0shot}$	0.5137	**0.8676**
2-stage$_{1shot}$	0.3710	0.7216
2-stage$_{concrete}$	0.5106	0.8235
1-stage	**0.5757**	0.8542
Planning$_{2stage}$	0.5289	0.7744
Planning$_{1stage}$	0.5238	0.7332

Table 1: Comparison of different methods' performance on pass@k metric.

Method	**Code Length**
Baseline	166.91
2-stage$_{0shot}$	**130.58**
2-stage$_{1shot}$	166.30
2-stage$_{concrete}$	146.99
1-stage	150.74

Table 2: Average code length of different methods.

4.2.1 Baseline

Generally, the model exhibited satisfactory performance on overall easy problems; however, it struggled with more complex problems that involved intricate instructions. Additionally, the generated code appeared to have some room for abbreviation by leveraging techniques such as List comprehension or built-in functions.

4.2.2 2-stage Summarization

0-shot In general, there was a significant reduction in code length. The model utilized various techniques to condense the code, including converting for/if statements into List comprehensions, shortening variable names, using Built-in functions, and consolidating multiple lines of declarations into a single line. Quantitatively, there was approximately a 22% reduction in code length compared to the baseline.

However, errors in the code propagated from the Implementation stage to the Summarization stage. Furthermore, for some problems, the model struggled to maintain the original functionality of the given function, resulting in a 11% decrease in the pass@1 metric. Remarkably, pass@10 increased compared to the baseline. Although in a highly restricted context, it appears that the model corrected errors in the original source code during the abbreviation process.

Based on these experimental results, it was concluded that the model did not have a strong understanding of the Summarization Task. As a response, the introduction of In-Context examples and the adoption of prompt engineering techniques involving detailed requirements were implemented.

1-shot Contrary to expectations, there were issues where approximately 22.7% of the evaluation data failed to summarize the provided code and instead outputted the example code. Furthermore, even in tasks where summarization was successfully performed, there were problems with the process leading to more loss of original functionality.

This performance decline is presumed to be attributed to the absence of an appropriate few-shot prompt template for the code summarization task. It appears that the model struggled to properly distinguish between In-context examples and actual input that needs to be summarized. Moreover, given the nature of the task, prompting examples as pairs before and after summarization would require a long context of around 500 tokens for each example. This lengthy context likely hindered the model's effective understanding of the given task. Lastly, there was a lack of similarity between the provided examples and the problems to be solved. Each scenario had different code lengths, variable names, and required a variety of summarization techniques (List Comprehension, Built-in function, etc.). With only a few In-Context examples, it seems insufficient to aid in understanding the task in situations where various techniques need to be considered.

Concrete strategies The expectation was that providing not only a request for code summarization but also techniques to assist in the summarization process would be more helpful. However, the experimental results showed that pass@k benchmark of this method were similar with 0-shot method, with the code length being measured longer. Qualitative evaluations indicated that, for some problems, better summarization performance was observed compared to the 0-shot method. However, for the majority of problems, the model either struggled to generate meaningful summaries or even produced code longer than the original. It appears that the model did not effectively leverage the techniques provided in the prompts.

4.2.3 1-stage Summarization

For the 1-stage summarization method, it was observed that most of the code outputs closely resembled the baseline, indicating that the model tend to generate its implementation without effectively shortening the code. Consequently, while achieving pass@k performance at a level similar to the baseline, this method lagged behind in terms of code length compared to other summarization methods. The prompt's added requirement for concise code seemed to introduce more constraints, diminishing the model's understanding of the summarization task. Successful summarization was limited, and detailed results can be found in Appendix B.

4.2.4 Self-Planning Approach

Method	pass@1	pass@10
Baseline	0.5776	0.8617
Planning$_{2stage}$	0.5289	0.7744
Planning$_{1stage}$	0.5238	0.7332

Table 3: Self-planning Approach's performance on pass@k metric.

For both the 3-stage planning summarization and 2-stage planning summarization methods, a Self-planning approach was employed before the summarization. Since the basic structure of methods was built on a 2-stage summarization method in the form of Implementation - Summarization, it was deemed necessary to achieve pass@k performance beyond the baseline in the Implementation stage for the subsequent summarization to be meaningful. Therefore, the decision was made to first test up to the Implementation stage and proceed to the Summarization stage only if performance improvement was observed.

However, the experimental results revealed a failure to achieve performance beyond the baseline in the Implementation stage. As a result, the Summarization stage was not pursued. This report details the results up to the Implementation stage.

2-stage Planning During the Planning stage, the model often struggled to articulate its requirements in natural language, as demonstrated in many instances. For example, in Listing 14, there is a problem where the model failed to specify the necessary formula during the rescale process. This type of issue in the Planning stage propagated to the Implementation stage, resulting in the generation of incorrect code. Additionally, as seen in Listing 18, there were cases where the Planning stage was executed correctly, but the model failed to generate code that accurately followed the plan during the implementation. While there were some instances where the model successfully addressed problems that the Baseline failed to solve, the performance decline due to the aforementioned issues outweighed such improvements.

The primary reason for the deviation from the results of the original paper can be attributed to the fact that, unlike the original paper, we had to craft our own prompt templates for the Planning phase and use them arbitrarily, as no specific templates or In-context examples were provided. The original paper mentions that the best performance was achieved when using a 4-shot prompt, and a decrease in performance was observed as the number of shots decreased. This suggests that the specific prompts used during the Planning stage significantly influenced the model's performance.

1-stage Planning 1-stage Planning also yielded similar results to 2-stage Planning. While the baseline only required the execution of the implementation task, 1-stage Planning added the constraint to perform planning before implementation. Based on the results of 2-stage planning, it can be inferred that adding such a constraint, while the model has not properly understood the Planning task, likely confused the model.

5 Conclusion

5.1 Summary of experiment results

Results We proposed several prompt engineering approaches for short code generation using LLM. We mainly tried the self-planning method and the multi-stage method, and although they did not improve performance as expected, they were successful in maintaining a certain level of efficiency while reducing the code length. This was particularly evident in the 2-stage summarization, where a reduction of approximately 40 characters on average was observed, making it the most efficient model evaluated. However, the in-context learning and 2-stage concrete approaches introduced in this study for more effective code summarization were found to be less effective than the 2-stage summarization. This led to the conclusion that simpler prompt templates may be more beneficial in aiding the model's understanding of the task.

Limitations Regarding self-planning, due to time constraints and the rarity of information disclosure, the desired level of performance improvement was not achieved. Furthermore, the disclosed prompt templates were applicable to LLaMA rather than Code LLaMA, showing no significant difference in quantitative assessments. Consequently, this study indicates that a different approach, specifically related to code generation and prompt engagement, might have been necessary.

5.2 Future work

We found that finding the right prompts for our experiments through human experimentation was more difficult and time-consuming than expected. We also found it difficult to discover them through experimentation alone because good prompt templates are not widely available. Therefore, further research is needed on how to efficiently discover prompts in a limited amount of time. Future endeavors might include: utilizing LLMs for efficient prompt discovery rather than human experimentation with prompts [13] ; and reinforcement learning approaches that think of a prompt as a trial and the output of the LLM as the outcome of the experiment. [14]

References

[1] Steven Forth. *Net Revenue Retention Strategy 2023*. 2023.

[2] Yue Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: *Empirical Methods in Natural Language Processing* (2021), pp. 8696–8708.

[3] Daya Guo et al. “UniXcoder: Unified Cross-Modal Pre-training for Code Representation”. In: *Association for Computational Linguistics* (2022), pp. 7212–7225.

[4] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: (2020). arXiv: 2005.14165.

[5] Hugo Touvron et al. “LLaMA: Open and Efficient Foundation Language Models”. In: (2023). arXiv: 2302.13971.

[6] Baptiste Rozière, Jonas Gehring, and Fabian Gloeckle. “Code Llama: Open Foundation Models for Code”. In: (2023).

[7] Harsha Nori et al. “Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine”. In: (2023). arXiv: 2311.16452.

[8] Pengfei Liu et al. “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. In: *ACM Computing Surveys* (2021).

[9] Jason Wei et al. “Emergent Abilities of Large Language Models”. In: (2022). arXiv: 2206.07682.

[10] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 24824–24837.

[11] Xue Jiang et al. *Self-planning Code Generation with Large Language Models*. 2023. arXiv: 2303.06689 [cs.SE].

[12] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[13] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023. arXiv: 2211.01910 [cs.LG].

[14] Mingkai Deng et al. *RLPrompt: Optimizing Discrete Text Prompts with Reinforcement Learning*. 2022. arXiv: 2205.12548 [cs.CL].

A Prompt Templates

7.1.1. Baseline

```
def match_parens(lst):
    """
    Given a list of two strings consisting only of open '(' and close ')'
    parentheses,
    checks if it's possible to concatenate the two strings in some order so that the
    resulting string is balanced. A string is balanced if all parentheses in it are
    matched.

    Returns 'Yes' if a balanced string can be formed, otherwise returns 'No'.

    Examples:
    - match_parens(['()(', ')']) -> 'Yes'
    - match_parens([')', ')']) -> 'No'
    - match_parens(['(()(())', '())())']) -> 'No'
    - match_parens([')())', '(()()(']) -> 'Yes'
    - match_parens(['(())))', '(()())((']) -> 'Yes'
    - match_parens(['()', '())']) -> 'No'
    - match_parens(['(()(', '()))()']) -> 'Yes'
    - match_parens(['((((', '((())']) -> 'No'
    - match_parens([')(()', '(()(']) -> 'No'
    - match_parens([')(', ')(']) -> 'No'
    - match_parens(['(', ')']) -> 'Yes'
    - match_parens([')', '(']) -> 'Yes'
    """
```

7.1.2. 2-stage summarization

```
Rewrite the code below as short as possible.
[PYTHON]
def get_odd_collatz(n):
    if n <= 0:
        raise ValueError("n should be a positive integer.")

    collatz_list = [n]
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        collatz_list.append(n)

    return sorted([x for x in collatz_list if x % 2 != 0])
[/PYTHON]
```

Listing 1: A prompt for re-entering code into the model for summarization, after entering instructions.

7.1.3. Baseline and summarization

```
Implement the code below as short as possible.
[PYTHON]
def get_odd_collatz(n):
    if n <= 0:
        raise ValueError("n should be a positive integer.")

    collatz_list = [n]
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        collatz_list.append(n)

    return sorted([x for x in collatz_list if x % 2 != 0])
[/PYTHON]
```

Listing 2: A prompt that involves entering instructions and simultaneously conducting summarization and implementation to create the shortest code possible.

7.1.4. 1-shot

```
Your task is to condense Python function that must be between the [PYTHON] and [\
    PYTHON] tag. Do not change the function name. Please do not repeat. Here is an
    example :
def minLengthAfterRemovals(self, nums: List[int]) -> int:
    counter = Counter(nums)
    heap = [ -cnt for cnt in counter.values()]
    heapify(heap)
    while len(heap) > 1:
        largest = -heappop(heap)
        second = -heappop(heap)
        if largest > 1:
            heappush(heap, 1-largest)
        if second > 1:
            heappush(heap, 1-second)
    return -heap[0] if heap else 0
def minLengthAfterRemovals(self, nums: List[int]) -> int:
    scount = Counter(nums)
    arr = sorted([scount[num] for num in sorted(scount.keys())], key=lambda x: -x)

    return (arr[0] - sum(arr[1:])) if arr[0] > sum(arr[1:]) else sum(arr) % 2
problem :
[PYTHON]
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if, in the given list of numbers, any two numbers are closer to each other
     than the given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
[/PYTHON]
```

Listing 3: This prompt was used for the 1shot. The examples used for 1-shot learning are sourced from LeetCode, with the top one being the original code and the bottom one being the summarized version, condensed using GPT-4.

7.1.5. 2-stage concrete

```
Your task is
to condense Python function that must be between the [PYTHON] and [/PYTHON] tag.
Do not change the function name. Please do not repeat.
You should apply following tips:
1. Erase comments.
2. Use dictionaries.
3. Change for statements to comprehensions.
4. Change if statements to conditional expressions or comprehensions.
5. Use built-in functions.
6. Use short circuit.
7. Use "join" to print everything at once.
8. Use short variable names.
problem :
[PYTHON]
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if, in the given list of numbers, any two numbers are closer to each other
     than the given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
[/PYTHON]
```

Listing 4: The idea was inspired by the chain of thought method, and it entailed a thorough personal analysis and compilation of short coding techniques and actual short codes gathered through extensive internet research.

7.1.6. Planning

```
[INST] Your task is to
Plan the steps to implement the following Python function.
Put a number in front of each step.
Response with ONLY natural language plan steps, with numbering.
DO NOT print python code. Print ONLY plans.
[\INST]

[PYTHON]
def make_a_pile(n):
    """
    Given a positive integer n, you have to make a pile of n levels of stones.
    The first level has n stones.
    The number of stones in the next level is:
        - the next odd number if n is odd.
        - the next even number if n is even.
    Return the number of stones in each level in a list, where element at index
    i represents the number of stones in the level (i+1).

    Examples:
    >>> make_a_pile(3)
    [3, 5, 7]
    """
[\PYTHON]
```

Listing 5: A prompt that takes instructions as input and returns planning.

7.1.7. Planning to implementation

```
Here is a plan to implement the `make_a_pile` function in Python:

1. Define a function called `make_a_pile` that takes an integer argument `n`.
2. Initialize a list called `stones` to store the number of stones in each level.
3. Set the first element of `stones` to `n`.
4. If `n` is odd, set the next element of `stones` to `n+1`, which is the next odd
   number. Otherwise, set it to `n+2`, which is the next even number.
5. Repeat step 4 until the length of `stones` is equal to `n`.
6. Return `stones`.
```

Listing 6: A prompt that takes planning as input and returns the implemented code.

7.1.8. Planning and implementation

```
[INST] Your task is to
Plan the steps to implement the following Python function.
Next, implement the function according to the created steps.
The Python code must be between [PYTHON] and [\PYTHON] tags.
You do not need to print plan steps.
[/INST]

[PYTHON]
from typing import List

def all_prefixes(string: str) -> List[str]:
    """ Return list of all prefixes from shortest to longest of the input string
    >>> all_prefixes('abc')
    ['a', 'ab', 'abc']
    """
[/PYTHON]
```

Listing 7: A prompt that provides instructions as input and simultaneously conducts planning and implementation.

B Qualitative Results

7.2.1. Baseline : different instructions

```python
def is_palindrome(string: str) -> bool:
    """ Test if given string is a palindrome """

def make_palindrome(string: str) -> str:
    """
    Find the shortest palindrome that begins with a supplied string.
    Algorithm idea is simple:
    - Find the longest postfix of supplied string that is a palindrome.
    - Append to the end of the string reverse of a string prefix that comes before
   the palindromic suffix.

    Examples:
    make_palindrome('') returns ''
    make_palindrome('cat') returns 'catac'
    make_palindrome('cata') returns 'catac'
    """
def is_palindrome(string: str) -> bool:
    return string == string[::-1]

def make_palindrome(string: str) -> str:
    if not string or len(string) == 1:
        return string
    postfix_start = 0
    for i in range(len(string) - 1, 0, -1):
        if is_palindrome(string[i:]):
            postfix_start = i
            break
    return string + string[postfix_start:len(string) - 1][::-1]
```

7.2.2.1. 2-stage summarization

```python
def parse_nested_parens(paren_string: str) -> List[int]:
    """
    Input to this function is a string represented multiple groups for nested
   parentheses separated by spaces.
    For each of the group, output the deepest level of nesting of parentheses.
    E.g. (()()) has maximum two levels of nesting while ((())) has three.
    >>> parse_nested_parens('(()()) ((())) () ((())()())')
    [2, 3, 1, 3]
    """
    def parse_parens(s: str) -> int:
        level = 0
        max_level = 0
        for c in s:
            if c == '(':
                level += 1
                max_level = max(max_level, level)
            elif c == ')':
                level -= 1
        return max_level

    return [parse_parens(s) for s in paren_string.split()]

def parse_nested_parens(s):
    return [max(0, s.count('(') - s.count(')')) for s in s.split()]
```

Listing 8: 2-stage summarization model can condense the given code well.

7.2.2.2. 2-stage summarization : the model cannot correct the wrong code.

```python
def search(lst):
    # create a dictionary to store the count of each integer
    freq = {}
    for num in lst:
        if num in freq:
            freq[num] += 1
        else:
            freq[num] = 1

    # find the maximum integer that satisfies the condition
    max_int = -1
    for num, count in freq.items():
        if num > 0 and count >= num and num > max_int:
            max_int = num

    return max_int
def search(lst):
    freq = {}
    for num in lst:
        freq[num] = freq.get(num, 0) + 1
    return max(freq, key=lambda x: x if freq[x] >= x > 0 else -1)
```

Listing 9: The above implementation is incorrect, which consequently leads to errors in the summarized version as well.

7.2.2.3. 2-stage summarization : the model can make errors.

```python
def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """
    For a given list of integers, return a tuple consisting of a sum and a product
    of all the integers in a list.
    Empty sum should be equal to 0 and empty product should be equal to 1.
    """
    sum = 0
    product = 1
    for num in numbers:
        sum += num
        product *= num
    return sum, product
def sum_product(numbers):
    return sum(numbers), prod(numbers) if numbers else (0, 1)
```

Listing 10: The initial implementation presented was correct; however, the subsequent abbreviated version of the code contained errors.

7.3.1. 1-shot : outputs the example as is

```python
def minLengthAfterRemovals(self, nums: List[int]) -> int:
    scount = Counter(nums)
    arr = sorted([scount[num] for num in sorted(scount.keys())], key=lambda x: -x)

    return (arr[0] - sum(arr[1:])) if arr[0] > sum(arr[1:]) else sum(arr) % 2
```

Listing 11: The model fails to solve its own problem and just outputs the example as is.

7.2.3.2. 1-shot : The model is not a good summarizer.

```python
def sort_even(l):
    return sorted(l[1::2]) + l[::2]
```

Listing 12: Even if it successfully summarizes, it loses the original functionality.

7.2.4 2-stage concrete : The model does not summarize that well.

```python
def parse_nested_parens(s):
    return [max(0, s.count('(') - s.count(')')) for s in s.split()]
def parse_nested_parens(paren_string: str) -> List[int]:
    def parse_parens(s: str) -> int:
        level = 0
        max_level = 0
        for c in s:
            if c == '(':
                level += 1
                max_level = max(max_level, level)
            elif c == ')':
                level -= 1
        return max_level

    return [parse_parens(s) for s in paren_string.split()]
```

Listing 13: The code above is summarized by 2-stage summarization, and the one below is summarized by 2-stage concrete.

7.2.5. Planning : In most cases, it is observed that the model fails to accurately capture the requirements in natural language.

```python
from typing import List

def rescale_to_unit(numbers: List[float]) -> List[float]:
    """
    Apply a linear transformation to a list of numbers so that the smallest number
    becomes 0 and the largest becomes 1.

    Args:
    numbers (List[float]): A list of floating point numbers, with at least two
    elements.

    Returns:
    List[float]: The list of numbers after applying the linear transformation.

    Examples:
    >>> rescale_to_unit([1.0, 2.0, 3.0, 4.0, 5.0])
    [0.0, 0.25, 0.5, 0.75, 1.0]
    """
```

Listing 14: Python instructions to rescale a list of numbers to unit range

Algorithm Steps:

```
    1. Sort the input list of numbers in ascending order.
    2. Find the minimum and maximum values in the sorted list.
    3. Calculate the range of the data (max - min).
    4. Create a new list to store the rescaled values.
    5. Iterate over the original list of numbers and for each 6. number, calculate
    its rescaled value using the following formula:
    7. Add the rescaled value to the new list.
    8. Return the new list of rescaled values.
```

Listing 15: Planning for the implementation of the above function. It claims to provide a formula but fails to do so.

7.2.6.1. Planning to implement : correctly solve difficult problems that the baseline got wrong.

```python
def intersperse(numbers: List[int], delimeter: int) -> List[int]:
    return [x for pair in zip(numbers[::2], numbers[1::2]) for x in pair] + numbers
    [-1:]
def intersperse(numbers: List[int], delimeter: int) -> List[int]:
    result = []
    for i in range(len(numbers)):
        result.append(numbers[i])
        if i < len(numbers) - 1:
            result.append(delimeter)
    return result
```

Listing 16: The above is a problem the baseline got wrong, but the code below correctly solves it by implementing planning.

7.2.6.2. Planning to implementation : Generates code that completely fails to meet the requirements of the Plan.

Algorithm Steps:

```
    1. Tokenize the input string into individual parentheses and spaces using the \
    texttt{split()} method.
    2. Create an empty list to store the maximum nesting levels of each group.
    3. Iterate over each token, keeping track of the current nesting level and the
    maximum nesting level for each group.
    4. If a closing parenthesis is encountered, decrement the nesting level.
    5. If a space is encountered, add the current nesting level to the list of
    maximum nesting levels.
    6. Return the list of maximum nesting levels.
```

Listing 17: Algorithm steps.

```python
def parse_nested_parens(paren_string: str) -> List[int]:
    tokens = paren_string.split()
    max_levels = []
    current_level = 0
    for token in tokens:
        if token == '(':
            current_level += 1
        elif token == ')':
            current_level -= 1
        else:
            max_levels.append(current_level)
    return max_levels
```

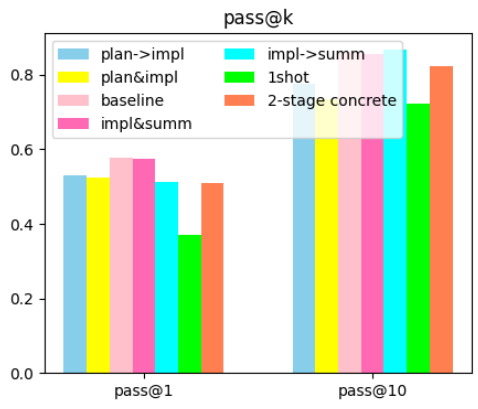
Listing 18: Spaces need to be included, but using split removes them.

7.2.7. Planning and implementation : It's unclear what was conducted during the planning phase, and the results are strange

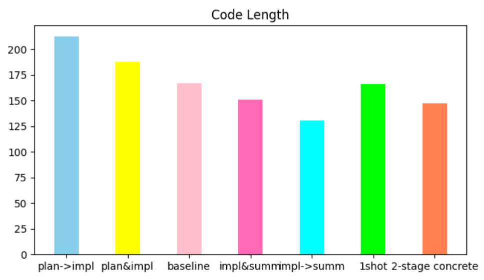
```python
def string_xor(a: str, b: str) -> str:
    """
    Input are two strings a and b consisting only of 1s and 0s. Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
    result = ''
    for i in range(len(a)):
        if a[i] == b[i]:
            result += '0'
        else:
            result += '1'
    return result
def string_xor(a, b):
    return bin(int(a, 2) ^ int(b, 2))[2:]
```

Listing 19: The above code is the baseline, and the code below is the result of combining planning and implementation.

C Quantative Results (Graph)



(a)



(b)

Figure 2: (a) pass@k (b) code length

D CodeLLaMA Inference Parameters

Parameter	**value**
top k	50
top p	1.0
max tokens	300
temperature	0
repeat penalty	1.1
presence penalty	0
frequency penalty	0
system prompt	Responses should be written in Python.
system prompt$_{planning}$	Responses should be written in natural language.

Table 4: CodeLLaMA Inference parameter.