

MapCoder: Multi-Agent Code Generation for Competitive Problem Solving

Md. Ashraful Islam¹, Mohammed Eunus Ali¹, Md Rizwan Parvez²

¹Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)

²Qatar Computing Research Institute (QCRI)
{mdashrafulpramanic, mohammed.eunus.ali}@gmail.com, mparvez@hbku.edu.qa

Abstract

Code synthesis, which requires a deep understanding of complex natural language (NL) problem descriptions, generation of code instructions for complex algorithms and data structures, and the successful execution of comprehensive unit tests, presents a significant challenge. Thus, while large language models (LLMs) demonstrate impressive proficiency in natural language processing (NLP), their performance in code generation tasks remains limited. In this paper, we introduce a new approach to code generation tasks leveraging the multi-agent prompting that uniquely replicates the full cycle of program synthesis as observed in human developers. Our framework, **MapCoder**, consists of four LLM agents specifically designed to emulate the stages of this cycle: recalling relevant examples, planning, code generation, and debugging. After conducting thorough experiments, with multiple LLMs ablations and analyses across eight challenging competitive problem-solving and program synthesis benchmarks—MapCoder showcases remarkable code generation capabilities, achieving their new state-of-the-art (pass@1) results—(HumanEval **93.9%**, MBPP **83.1%**, APPS **22.0%**, CodeContests **28.5%**, and xCodeEval **45.3%**). Moreover, our method consistently delivers superior performance across various programming languages and varying problem difficulties. We open-source our framework at <https://github.com/Md-Ashraful-Pramanik/MapCoder>.

1 Introduction

Computer Programming has emerged as an ubiquitous problem-solving tool that brings tremendous benefits to every aspects of our life (Li et al., 2022a; Parvez et al., 2018; Knuth, 1992). To maximize programmers’ productivity, and enhance accessibility, automation in program synthesis is paramount. With the growth of LLMs, significant

advancements have been made in program synthesis—driving us in an era where we can generate fully executable code, requiring no human intervention (Chowdhery et al., 2022; Nijkamp et al., 2022).

Despite LLMs’ initial success and the scaling up of model size and data, many of these models still struggle to perform well on complex problem-solving tasks, especially in competitive programming problems (Austin et al., 2021). To mitigate this gap, in this paper, we introduce **MapCoder**: a Multi-Agent Prompting Based Code Generation approach that can seamlessly synthesize solutions for competition-level programming problems.

Competitive programming or competition-level code generation, often regarded as the pinnacle of problem-solving, is an challenging task. It requires a deep comprehension of NL problem descriptions, multi-step complex reasoning beyond mere memorization, excellence in algorithms and data structures, and the capability to generate substantial code that produces desired outputs aligned with comprehensive test cases (Khan et al., 2023).

Early approaches utilizing LLMs for code generation employ a direct prompting approach, where LLMs generate code directly from problem descriptions and sample I/O (Chen et al., 2021a). Recent methods like chain-of-thought (Wei et al., 2022a) advocates modular or pseudo code-based generation to enhance planning and reduce errors, while retrieval-based approaches such as Parvez et al. (2021) leverage relevant problems and solutions to guide LLMs’ code generations. However, gains in such approaches remains limited in such a complex task like code generation where LLMs’ generated code often fails to pass the test cases and they do not feature bug-fixing schema (Ridnik et al., 2024).

A promising solution to the above challenge is self-reflection (Shinn et al., 2023; Chen et al., 2022), which iteratively evaluates the generated code against test cases, reflects on mistakes and

MapCoder：多智能体竞争问题求解的代码生成

马杜·阿什拉夫·伊斯拉姆，穆罕默德·尤努斯·阿里，马杜·里兹万·帕尔韦兹
计算机科学与工程学院 孟加拉国工程技术大学 (BUET)

卡塔尔计算研究学院 (QCRI) {mdashrafulpramanic,
mohammed.eunus.ali}@gmail.com,
mparvez@hbku.edu.qa

摘要

代码合成，需要深入理解复杂的自然语言 (NL) 问题描述，生成复杂算法和数据结构的代码指令，以及成功执行全面的单元测试，这提出了一个重大挑战。因此，尽管大型语言模型 (LLMs) 在自然语言处理 (NLP) 方面表现出令人印象深刻的熟练程度，但它们在代码生成任务中的表现仍然有限。在本文中，我们介绍了一种利用多智能体提示的新方法来处理代码生成任务，该方法独特地复制了人类开发者观察到的程序合成的完整周期。我们的框架 MapCoder 由四个LLM智能体组成，专门设计来模拟这个周期的各个阶段：回忆相关示例、规划、代码生成和调试。经过彻底的实验，包括在八个具有挑战性的竞争问题解决和程序合成基准上的多次LLMs消融和数据分析——MapCoder 展示了卓越的代码生成能力，实现了新的最先进 (pass@1) 结果——(HumanEval 93.9%，MBPP 83.1%，APPS 22.0%，CodeContests 28.5%，和 xCodeEval 45.3%)。此外，我们的方法在各种编程语言和不同问题难度上均能持续提供卓越的性能。我们开源

显著的进展已在程序综合领域取得——引领我们进入一个能够生成完全可执行代码的时代，无需人工干预 (Chowdhery 等, 2022; Nijkamp 等, 2022)。

尽管LLMs在初期取得了成功，并且模型规模和数据量得到了扩大，但这些模型仍然难以在复杂问题解决任务上表现出色，尤其是在竞技编程问题中 (Austin 等人, 2021 年)。为了弥合这一差距，本文介绍了 MapCoder：一种基于多智能体提示的代码生成方法，能够无缝合成竞技级编程问题的解决方案。

竞技编程或竞赛级代码生成，常被视为解决问题的顶峰，是一项具有挑战性的任务。它需要深入理解自然语言问题描述，进行多步骤的复杂推理，而不仅仅是记忆，还需要在算法和数据结构方面表现出色，以及生成大量能够产生符合全面测试用例所需输出的代码 (Khan 等, 2023)。

早期利用LLMs进行代码生成的方法采用直接提示方法，其中LLMs直接从问题描述和示例 I/O 生成代码 (Chen 等人, 2021a)。最近的方法，如思维链 (Wei 等人, 2022a) 提倡模块化或伪代码生成以增强规划和减少错误，而基于检索的方法，如 Parvez 等人 (2021) 利用相关问题和解决方案来指导LLMs的代码生成。然而，在如此复杂的任务如代码生成中，这些方法的收益仍然有限，因为LLMs生成的代码往往无法通过测试案例，并且它们不具备错误修复方案 (Ridnik 等人, 2024)。

一种应对上述挑战的有前景的解决方案是自我反思 (Shinn 等, 2023; Chen 等, 2022)，该方法迭代地评估生成的代码与测试用例的匹配度，反思错误。

我们的框架在 <https://github.com/Md-Ashraful-Pramanik/MapCoder>。

1 引言

计算机编程已成为一种无处不在的问题解决工具，为我们的生活的各个方面带来了巨大的益处 (Li 等, 2022a; Parvez 等, 2018; Knuth, 1992)。为了最大化程序员的效率并提高可访问性，程序综合中的自动化至关重要。随着LLMs的增长

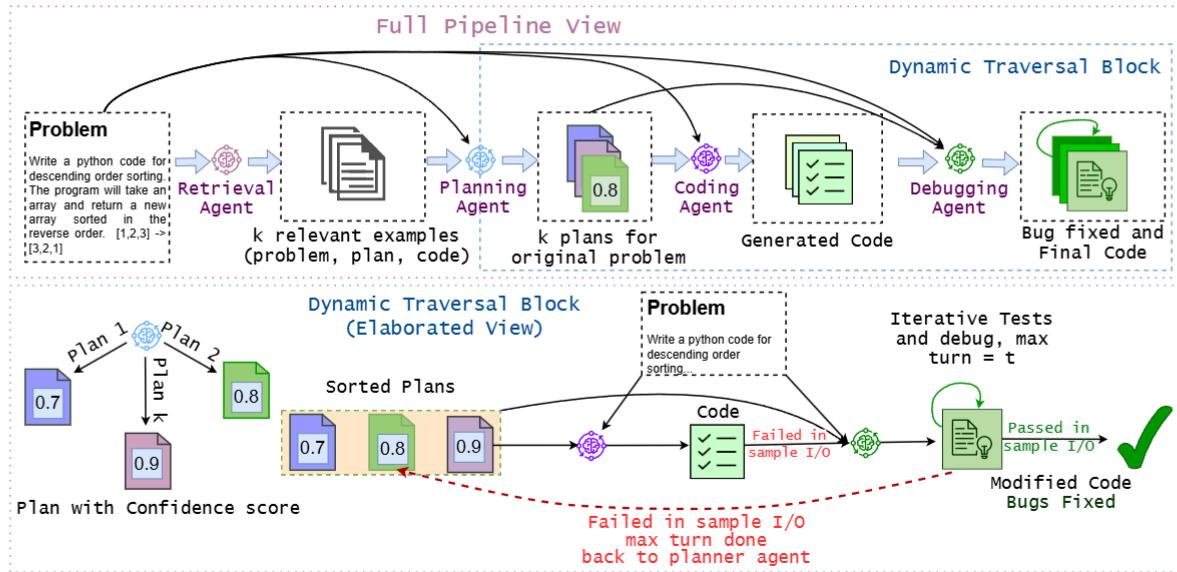


Figure 1: Overview of MapCoder (top). It starts with a retrieval agent that generates relevant examples itself, followed by planning, coding, and iterative debugging agents. Our dynamic traversal (bottom) considers the confidence of the generated plans as their reward scores and leverages them to guide the code generation accordingly.

modifies accordingly. However, such approaches have limitations too. Firstly, while previous studies indicate that superior problem-solving capabilities are attained when using in-context exemplars (Shum et al., 2023; Zhang et al., 2022; Wei et al., 2022a) or plans (Jiang et al., 2023b), these approaches, during both code generation and debugging, only leverage the problem description itself in a zero-shot manner. Consequently, their gains can be limited.

To confront the above challenge, we develop MapCoder augmenting the generation procedure with possible auxiliary supervision. We draw inspiration from human programmers, and how they use various signals/feedback while programming. The human problem-solving cycle involves recalling past solutions, planning, code writing, and debugging. MapCoder imitates these steps using LLM agents - retrieval, planning, coding, and debugging. In contrast to relying on human annotated examples, or external code retrieval models, we empower our retrieval agent to autonomously retrieve relevant problems itself (Yasunaga et al., 2023). Moreover, we design a novel structured pipeline schema that intelligently cascades the LLM agents and incorporates a dynamic iteration protocol to enhance the generation procedure at every step. Figure 1 shows an overview of our approach, MapCoder.

Additionally, existing iterative self-reflection methods rely on extra test cases generated by LLM agents (e.g., AgentCoder (Huang et al., 2023), LATS (Zhou et al., 2023), self-reflection (Shinn

et al., 2023)) or external tools, compounding the challenges. Test case generation is equally challenging as code generation (Pacheco et al., 2007), and incorrect test cases can lead to erroneous code. Blindly editing code based on these test cases can undermine problem-solving capabilities. For instance, while self-reflection boosts GPT-4’s performance on the HumanEval dataset, it drops by 3% on the MBPP dataset (Shinn et al., 2023). Upon identification, to validate this, on the HumanEval dataset itself, we replace their GPT-4 with ChatGPT, and see that model performance drops by 26.3%. Therefore, our debugging agent performs unit tests and bug fixing using only the sample I/O, without any artifact-more plausible for real-world widespread adoption.

We evaluate MapCoder on seven popular programming synthesis benchmarks including both basic programming like HumanEval, MBPP and challenging competitive program-solving benchmarks like APPS, CodeContests and xCodeEval. With multiple different LLMs including ChatGPT, GPT-4, and Gemini Pro, our approach significantly enhances their problem-solving capabilities - consistently achieving new SOTA performances, outperforming strong baselines like Reflexion (Shinn et al., 2023), and AlphaCodium (Ridnik et al., 2024). Moreover, our method consistently delivers superior performance across various programming languages and varying problem difficulties. Furthermore, with detailed ablation studies, we analyze MapCoder to provide more insights.

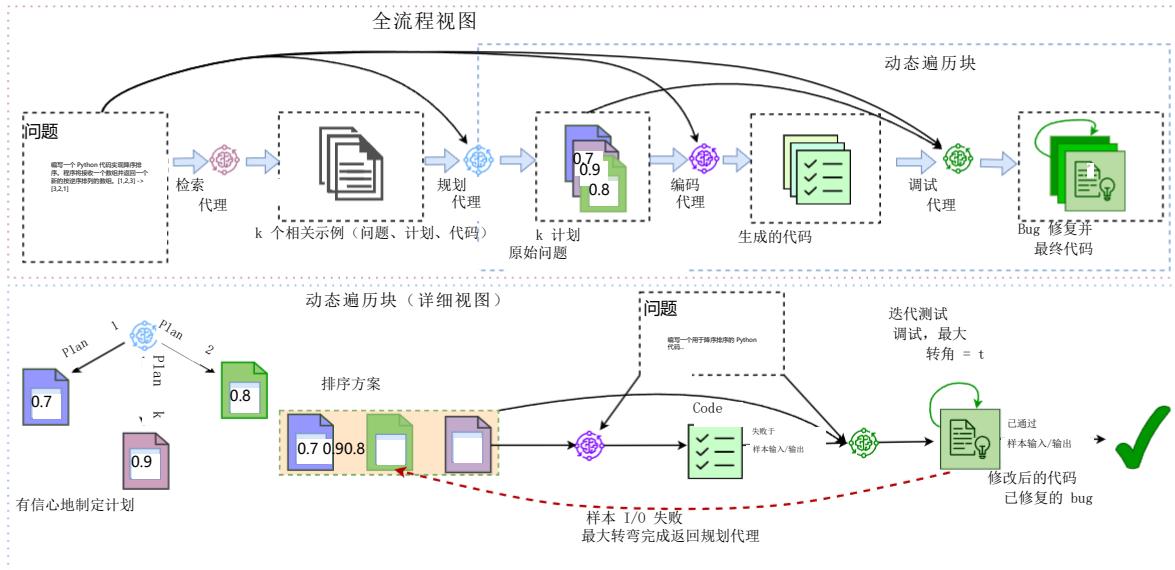


图1：MapCoder 概述（顶部）。它从生成相关示例的检索代理开始，接着是规划、编码和迭代调试代理。我们的动态遍历（底部）考虑生成计划的置信度作为其奖励分数，并据此引导代码生成。

相应地修改。然而，这种方法也存在局限性。首先，尽管先前的研究表明，在上下文中使用示例（Shum 等，2023；张等，2022；魏等，2022a）或计划（江等，2023b）可以获得更优越的问题解决能力，但这些方法在代码生成和调试过程中，仅以零样本的方式利用问题描述本身。因此，它们的收益可能有限。

面对上述挑战，我们开发了 MapCoder，通过增加可能的辅助监督来增强生成过程。我们从人类程序员那里汲取灵感，了解他们在编程时如何使用各种信号/反馈。人类问题解决周期包括回忆过去的解决方案、规划、编写代码和调试。MapCoder 通过使用LLM代理——检索、规划、编码和调试——来模仿这些步骤。与依赖于人类标注的示例或外部代码检索模型相比，我们赋予我们的检索代理自主检索相关问题的能力（Yasunaga 等人，2023）。此外，我们设计了一种新颖的结构化管道方案，智能地级联LLM代理，并引入动态迭代协议，以增强每一步的生成过程。图1展示了我们的方法，MapCoder 的概述。

此外，现有的迭代自反方法依赖于由LLM代理生成的额外测试用例（例如，AgentCoder（黄等，2023年），LATS（周等，2023年），自反 Shinn

等人，2023))或外部工具，增加了挑战。测试用例生成与代码生成同样具有挑战性（Pacheco 等人，2007），错误的测试用例可能导致错误代码。盲目地根据这些测试用例编辑代码会削弱解决问题的能力。例如，虽然自我反思提升了 GPT-4 在 HumanEval 数据集上的性能，但在 MBPP 数据集上下降了 3%（Shinn 等人，2023）。在识别出这一点后，为了验证这一点，我们在 HumanEval 数据集本身上用 ChatGPT 替换他们的 GPT-4，发现模型性能下降了 26.3%。因此，我们的调试代理仅使用样本 I/O 进行单元测试和错误修复，没有任何人工制品——更符合现实世界的广泛应用。

我们评估了 MapCoder 在包括 HumanEval、MBPP 等基本编程和 APPS、CodeContests 以及 xCodeEval 等具有挑战性的竞赛程序求解基准在内的七个流行的编程合成基准上。通过使用包括 ChatGPT、GPT-4 和 Gemini Pro 在内的多种不同的LLMs，我们的方法显著增强了它们的问题解决能力——持续实现新的 SOTA 性能，超越了如 Reflexion（Shinn 等，2023）和 AlphaCodium（Ridnik 等，2024）等强大的基线。此外，我们的方法在各种编程语言和不同难度的问题上持续提供优越的性能。此外，通过详细的消融研究，我们分析了 MapCoder 以提供更多见解。

2 Related Work

Program Synthesis: Program synthesis has a long standing history in AI systems (Manna and Waldinger, 1971). A large number of prior research attempted to address it via search/data flow approaches (Li et al., 2022a; Parisotto and Salakhutdinov, 2017; Polozov and Gulwani, 2015; Gulwani, 2011). LMs, prior to LLMs, attempt to generate code by fine-tuning (i.e., training) neural language models (Wang et al., 2021; Ahmad et al., 2021; Feng et al., 2020; Parvez et al., 2018; Yin and Neubig, 2017; Hellendoorn and Devanbu, 2017; Rabinovich et al., 2017; Hindle et al., 2016), conversational intents or data flow features (Andreas et al., 2020; Yu et al., 2019).

Large Language Models: Various LLMs have been developed for Code synthesis (Li et al., 2022b; Fried et al., 2022; Chen et al., 2021b; Austin et al., 2021; Nijkamp et al., 2022; Allal et al., 2023). Recent open source LLMs include Llama-2 (Touvron et al., 2023), CodeLlama-2 (Roziere et al., 2023), Mistral (Jiang et al., 2023a) Deepseek Coder (Guo et al., 2024), MoTCoder (Li et al., 2023) that are capable of solving many basic programming tasks.

Prompting LLMs: As indicated in Section 1, LLM prompting can be summarized into three categories: retrieval (Yasunaga et al., 2023; Parvez et al., 2023, 2021); planning (Wei et al., 2022b; Jiang et al., 2023b); debugging (Ridnik et al., 2024; Chen et al., 2023, 2022; Le et al., 2022) apart from the direct code generation approaches. In contrast, we combine all these paradigms and bridge their gaps (See Table 1). Among others, in different contexts of generic problem-solving, Tree-of-thoughts (Yao et al., 2023), and Cumulative reasoning (Zhang et al., 2023) approaches consider a tree traversal approach to explore different sub-steps towards a solution while our code generation approach mirrors the human programming cycle through various LLM agents. Notably, our traversal does not rely on sub-steps toward the solution but instead utilizes different forms of full solutions.

Approach	Self-retrieval	Planning	Additional test cases generation	Debugging
Reflexion	x	x	✓	✓
Self-planning	x	✓	x	x
Analogical	✓	✓	x	x
AlphaCodium	x	x	✓	✓
MapCoder	✓	✓	x	✓

Table 1: Features in code generation prompt techniques.

3 MapCoder

Our goal is to develop a multi-agent code generation approach for competitive problem-solving. In order to do so, our framework, MapCoder, replicates the human programming cycle through four LLM agents - retrieval, plan, code, and debug. We devise a pipeline sequence for MapCoder, intelligently cascading the agents in a structured way and enhancing each agent’s capability by augmenting in-context learning signals from previous agents in the pipeline. However, not all the agent responses/outputs are equally useful. Therefore, additionally, MapCoder features an adaptive agent traversal schema to interact among corresponding agents dynamically, iteratively enhancing the generated code by, for example, fixing bugs, while maximizing the usage of the LLM agents. In this section, we first discuss the agents (as per the pipeline), their prompts, and interactions, followed by the dynamic agent traversal protocol in MapCoder towards code generation for competitive problem-solving.

3.1 Retrieval Agent

Our first agent, the *Retrieval Agent*, recalls past relevant problem-solving instances, akin to human memory. It finds k (user-defined) similar problems without manual crafting or external retrieval models. Instead, we leverage the LLM agent itself, instructing it to generate such problems. Our prompt extends the analogical prompting principles (Yasunaga et al., 2023), generating examples and their solutions simultaneously, along with additional metadata (e.g., problem description, code, and plan) to provide the following agents as auxiliary data. We adopt a specific sequence of instructions, which is crucial for the prompt’s effectiveness. To achieve this, we instruct the LLM to produce similar and distinct problems and their solutions, facilitating problem planning through reverse-engineering. In particular, we prompt the LLM to generate solution code step-by-step, allowing for post-processing of thoughts sequences to form the corresponding plan. Finally, we direct the LLM to generate relevant algorithms and provide instructional tutorials, enabling the agent to reflect on underlying algorithms and generate algorithmically similar examples.

3.2 Planning Agent

The second agent, the *Planning Agent*, aims to create a step-by-step plan for the original problem.

2 相关工作

程序综合：程序综合在人工智能系统中有着悠久的历史 (Manna 和 Waldinger, 1971)。大量先前研究试图通过搜索/数据流方法来解决它 (Li 等人, 2022a; Parisotto 和 Salakhutdinov, 2017; Polozov 和 Gulwani, 2015; Gulwani, 2011)。在LLMs之前，语言模型 (LMs) 尝试通过微调 (即训练) 神经语言模型 (Wang 等人, 2021; Ahmad 等人, 2021; Feng 等人, 2020; Parvez 等人, 2018; Yin 和 Neubig, 2017; Hellendoorn 和 Devanbu, 2017; Rabinovich 等人, 2017; Hindle 等人, 2016) 或会话意图或数据流特征来生成代码 (Andreas 等人, 2020; Yu 等人, 2019)。

大型语言模型：各种LLMs有已开发用于代码综合 (Li 等, 2022b; Fried 等, 2022; Chen 等, 2021b; Austin 等, 2021; Nijkamp 等, 2022; Allal 等, 2023)。近期开源的LLMs包括 Llama-2 (Touvron 等, 2023)、CodeLlama-2 (Roziere 等, 2023)、Mistral (Jiang 等, 2023a)、Deepseek Coder (Guo 等, 2024)、MoTCoder (Li 等, 2023)，它们能够解决许多基本编程任务。提示LLMs：如第 1 节所示，LLM提示可以总结为三类：检索 (Yasunaga 等, 2023; Parvez 等, 2023, 2021)；规划 (Wei 等, 2022b; Jiang 等, 2023b)；调试 (Ridnik 等, 2024; Chen 等, 2023, 2022; Le 等, 2022)，除了直接的代码生成方法。相比之下，我们结合了所有这些范式并弥合了它们之间的差距 (见表 1)。在通用问题解决的多种不同情境中，Tree-ofthoughts (Yao 等, 2023) 和 Cumulative reasoning (Zhang 等, 2023) 方法考虑了树遍历方法来探索通向解决方案的不同子步骤，而我们的代码生成方法通过各种LLM代理反映了人类编程周期。显著的是，我们的遍历不依赖于解决问题的子步骤，而是利用不同的完整解决方案形式。

3 地图编码器

我们的目标是开发一种多智能体代码生成方法，用于竞争性问题解决。为此，我们的框架 MapCoder 通过四个LLM智能体——检索、计划、代码和调试——来复制人类编程周期。我们为 MapCoder 设计了一个管道序列，智能地以结构化方式级联智能体，并通过增强来自管道中先前智能体的上下文学习信号来提高每个智能体的能力。然而，并非所有智能体的响应/输出都同样有用。因此，MapCoder 还具备一个自适应智能体遍历方案，以动态地交互相应的智能体，通过例如修复错误等方式，迭代地增强生成的代码，同时最大化使用LLM智能体。在本节中，我们首先讨论智能体（按照管道顺序）、它们的提示和交互，然后讨论 MapCoder 中针对竞争性问题解决代码生成的动态智能体遍历协议。

3.1 检索代理

我们的第一个智能体，检索智能体，能够回忆过去相关的解决问题的实例，类似于人类的记忆。它能够找到 k (用户定义的) 相似问题，无需手动制作或使用外部检索模型。相反，我们利用LLM智能体本身，指示它生成这些问题。我们的提示扩展了类比提示原则 (Yasunaga 等人, 2023)，同时生成示例及其解决方案，以及额外的元数据（例如问题描述、代码和计划），为以下智能体提供辅助数据。我们采用特定的指令序列，这对于提示的有效性至关重要。为了实现这一点，我们指示LLM生成相似和不同的问题及其解决方案，通过逆向工程促进问题规划。特别是，我们提示LLM逐步生成解决方案代码，允许对思维序列进行后处理以形成相应的计划。最后，我们指导LLM生成相关算法并提供教学教程，使智能体能够反思底层算法并生成算法上相似的示例。

自检索规划方法			额外 测试用例 代	调试
反思 X X ✓ ✓	自我规划 X ✓ X X	类比 X X ✓ X	AlphaCoderium X X ✓ X	
地图编码器 ✓ ✓ X ✓				

表 1：代码生成提示技术中的特征。

3.2 规划代理

第二个智能体，即规划智能体，旨在为原始问题制定一个逐步的计划。

```


Planning Generation Prompt:
Given a competitive programming problem generate a concrete planning to solve the problem.
# Problem: {Description of a self-retrieved example problem}
# Planning: {Planning of that problem}
## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by the Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample I/Os}

Confidence Generation Prompt:
Given a competitive programming problem and a plan to solve the problem in {language} tell whether the plan is correct to solve this problem.
# Problem: {Original Problem}
# Planning: {Planning of our problem from previous step}


```

Figure 2: Prompt for *Planning Agent*.

Our *Planning Agent* uses examples and their plans obtained from the retrieval agent to generate plans for the original problem. A straightforward approach would be to utilize all examples collectively to generate a single target plan. However, not all retrieved examples hold equal utility. Concatenating examples in a random order may compromise the LLM’s ability to generate accurate planning. For instance, Xu et al. (2023) demonstrated that even repeating more relevant information (e.g., query) towards the end of the in-context input aids LLM reasoning more effectively than including relatively less relevant contexts. A similar conclusion of "separating noisy in-context data" can also be drawn from the state-of-the-art retrieval augmented generation approaches like Wang et al. (2023). Therefore, we generate a distinct target plan for each retrieved example. Additionally, multiple plans offer diverse pathways to success.

To help the generation steps in the following agents with the utility information for each plan, our designed prompt for the planning agent asks the LLM to generate both plans and a confidence score. Figure 2 shows our prompt got this agent.

3.3 Coding Agent

Next is the *Coding Agent*. It takes the problem description, and a plan from the *Planning Agent* as input and translates the corresponding planning into code to solve the problem. During the traversing of agents, *Coding Agent* takes the original problem and one particular plan from the *Planning Agent*, generates the code, and test on sample I/O. If the initial code fails, the agent transfers it to the next agent for debugging. Otherwise, predicts that as the final solution.

3.4 Debugging Agent

Finally, the *Debugging Agent* utilizes sample I/O from the problem description to rectify bugs in the generated code. Similar to humans cross-checking

their plan while fixing bugs, our pipeline supplements the *Debugging Agent* with plans from the *Planning Agent*. This plan-derived debugging significantly enhances bug fixing in MapCoder, underscoring the pivotal roles played by both the *Debugging Agent* and the *Planning Agent* in the generation process. We verify this in Section 6. For each plan, this process is repeated t times. The prompt for this step is illustrated in Figure 3. Note that, different from Reflexion (Shinn et al., 2023) and AlphaCodium (Ridnik et al., 2024), our *Debugging Agent* does not require any additional test case generation in the pipeline.

```


Debugging Agent
Given a competitive programming problem you have generated {language} code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving this problem.


```

Figure 3: Prompt for *Debugging Agent*.

3.5 Dynamic Agent Traversal

The dynamic traversal in MapCoder begins with the *Planning Agent*, which outputs the plans for the original problem with confidence scores. These plans are sorted, and the highest-scoring one is sent to the *Coding Agent*. The *Coding Agent* translates the plan into code, tested with sample I/Os. If all pass, the code is returned; otherwise, it’s passed to *Debugging Agent*. They attempt to rectify the code iteratively up to t times. If successful, the code is returned; otherwise, responsibility shifts back to the *Planning Agent* for the next highest confidence plan. This iterative process continues for k iterations, reflecting a programmer’s approach. We summarize our agent traversal in Algorithm A in Appendix. Our algorithm’s complexity is $O(kt)$. An example illustrating MapCoder’s problem-solving compared to Direct, Chain-of-thought, and Reflexion approaches is in Figure 4. All detailed prompts for each agent are in Appendix B.

4 Experimental Setup

4.1 Datasets

For extensive evaluation, we have used eight benchmark datasets: five from basic programming and three from complex competitive programming domains. Five basic programming datasets are:

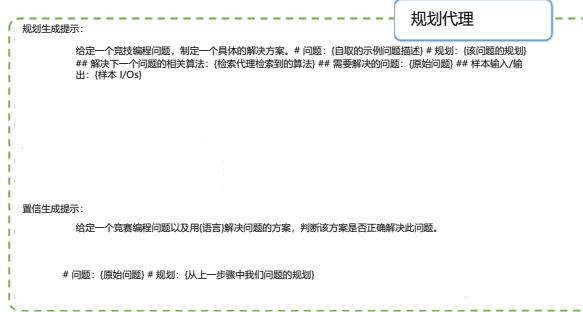


图 2：规划代理提示

我们的规划代理使用从检索代理获取的示例及其计划来生成原始问题的计划。一种直接的方法是利用所有示例共同生成一个目标计划。然而，并非所有检索到的示例都具有同等效用。随机顺序拼接示例可能会损害LLM生成准确规划的能力。例如，Xu 等人（2023）证明了即使在上下文输入的末尾重复更相关的信息（例如查询），也比包括相对不相关的上下文更能有效地帮助LLM推理。从 Wang 等人（2023）等最先进的检索增强生成方法中，也可以得出“分离噪声上下文数据”的类似结论。因此，我们为每个检索到的示例生成一个独特的目标计划。此外，多个计划提供了通往成功的不同路径。

为了帮助以下智能体的生成步骤，我们为规划智能体设计的提示要求LLM生成两个计划和置信度得分。图 2 展示了我们的提示如何作用于该智能体。

3.3 编码代理

接下来是编码代理。它接收问题描述和规划代理的规划作为输入，并将相应的规划转换为代码以解决问题。在代理遍历过程中，编码代理接收原始问题和规划代理的一个特定计划，生成代码并在样本输入输出上进行测试。如果初始代码失败，代理将其传递给下一个代理进行调试。否则，预测其为最终解决方案。

3.4 调试代理

最后，调试代理利用问题描述中的示例输入/输出以纠正生成的代码中的错误

与人类在修复错误时交叉检查计划类似，我们的流水线向调试代理补充来自规划代理的计划。这种基于计划的调试显著提高了 MapCoder 中的错误修复，突出了调试代理和规划代理在生成过程中的关键作用。我们在第 6 节中验证了这一点。对于每个计划，这个过程重复 t 次。这一步骤的提示如图 3 所示。请注意，与 Reflexion (Shinn 等, 2023) 和 AlphaCodium (Ridnik 等, 2024) 不同，我们的调试代理在流水线中不需要生成任何额外的测试用例。

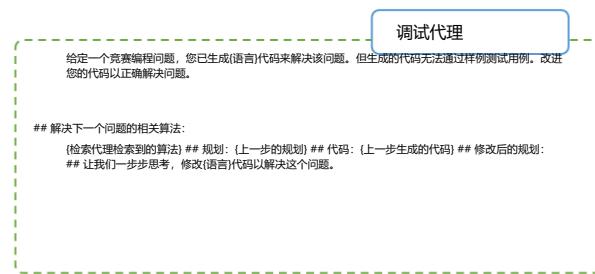


图 3：调试代理提示

3.5 动态代理遍历

动态遍历在 MapCoder 中从规划代理开始，该代理输出原始问题的计划并附带置信度分数。这些计划被排序，得分最高的计划被发送到编码代理。编码代理将计划转换为代码，并使用示例输入输出进行测试。如果全部通过，则返回代码；否则，将其传递给调试代理。他们尝试迭代性地修正代码，最多 t 次。如果成功，则返回代码；否则，责任转回到规划代理以获取下一个最高置信度的计划。此迭代过程持续 k 次迭代，反映了程序员的处理方式。我们在附录 A 中的算法 A 中总结了代理遍历。我们算法的复杂度是 $O(kt)$ 。图 4 展示了 MapCoder 的解决问题的示例，与直接、思维链和反思方法进行比较。每个代理的所有详细提示都在附录 B 中。

4 实验设置

4.1 数据集

为了进行广泛的评估，我们使用了八个基准数据集：其中五个来自基础编程领域，三个来自复杂竞技编程领域。五个基础编程数据集包括：

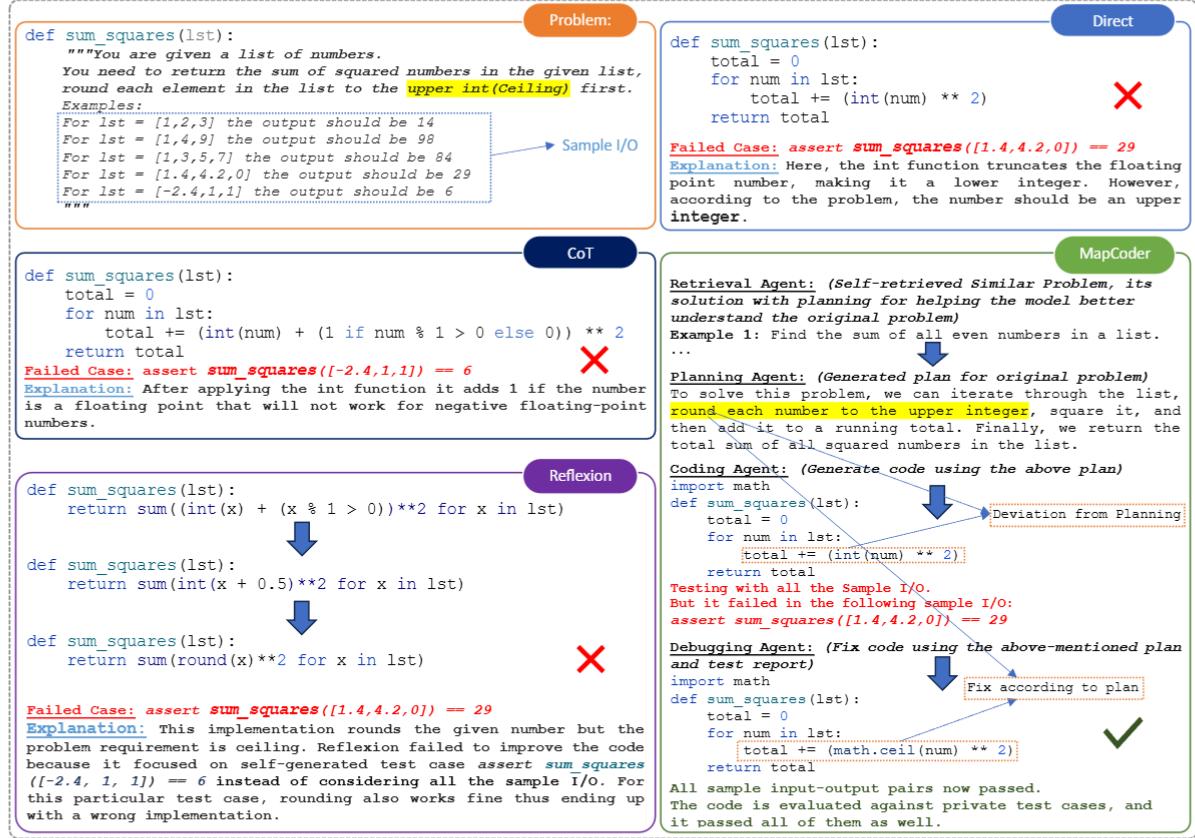


Figure 4: Example problem and solution generation using Direct, CoT, Reflexion, and MapCoder prompts. MapCoder explores high-utility plans first and uniquely features a plan-derived debugging for bug fixing.

HumanEval (Chen et al., 2021a), **HumanEval-ET** (Dong et al., 2023a), **EvalPlus** (Liu et al., 2023), **MBPP** (Austin et al., 2021), and **MBPP-ET** (Dong et al., 2023a). HumanEval-ET, EvalPlus extend HumanEval and MBPP-ET comprehends MBPP by incorporating more test cases. The problem set size of HumanEval and MBPP (and their extensions) are 164 and 397, respectively. Due to the absence of sample I/O in MBPP and MBPP-ET, our approach for code moderation involves randomly removing one test-case from MBPP-ET for each problem and provide this test-case as a sample I/O for the problem. Importantly, this removed test-case is carefully selected to ensure mutual exclusivity from the hidden test sets in MBPP and MBPP-ET. Three competitive programming datasets are: Automated Programming Progress Standard (**APPS**), **xCodeEval** (Khan et al., 2023), and **CodeContest**, where we have used 150 (randomly selected from test set), 106 (compact set), and 165 (test set) problems, respectively, in our experiments.

4.2 Baselines

We have compared MapCoder with several baselines and state-of-the-art approaches. **Direct**

Prompting instructs language models to generate code without explicit guidance, relying on their inherent capabilities of LLM. Chain of Thought Prompting (**CoT**) (Wei et al., 2022b) breaks down problems into step-by-step solutions, enabling effective tackling of complex tasks. **Self-Planning** Prompting (Jiang et al., 2023b) divides the code generation task into planning and implementation phases. **Analogical Reasoning** Prompting (Yasunaga et al., 2023) instructs models to recall relevant problems from training data. **Reflexion** (Shinn et al., 2023) provides verbal feedback to enhance solutions based on unit test results. **Self-collaboration** (Dong et al., 2023b) proposes a framework where different LLMs act as analyst, coder, and tester to cooperatively generate code for complex tasks, achieving better performance than directly using a single LLM. **AlphaCodium** (Ridnik et al., 2024) iteratively refines code based on AI-generated input-output tests.

4.3 Foundation Models, Evaluation Metric, k , and t

With $k = t = 5$ in HumanEval, and $k = t = 3$ for others, we evaluate all the datasets using **ChatGPT** (gpt-3.5-turbo-1106), **GPT-4** (gpt-4-1106-preview)

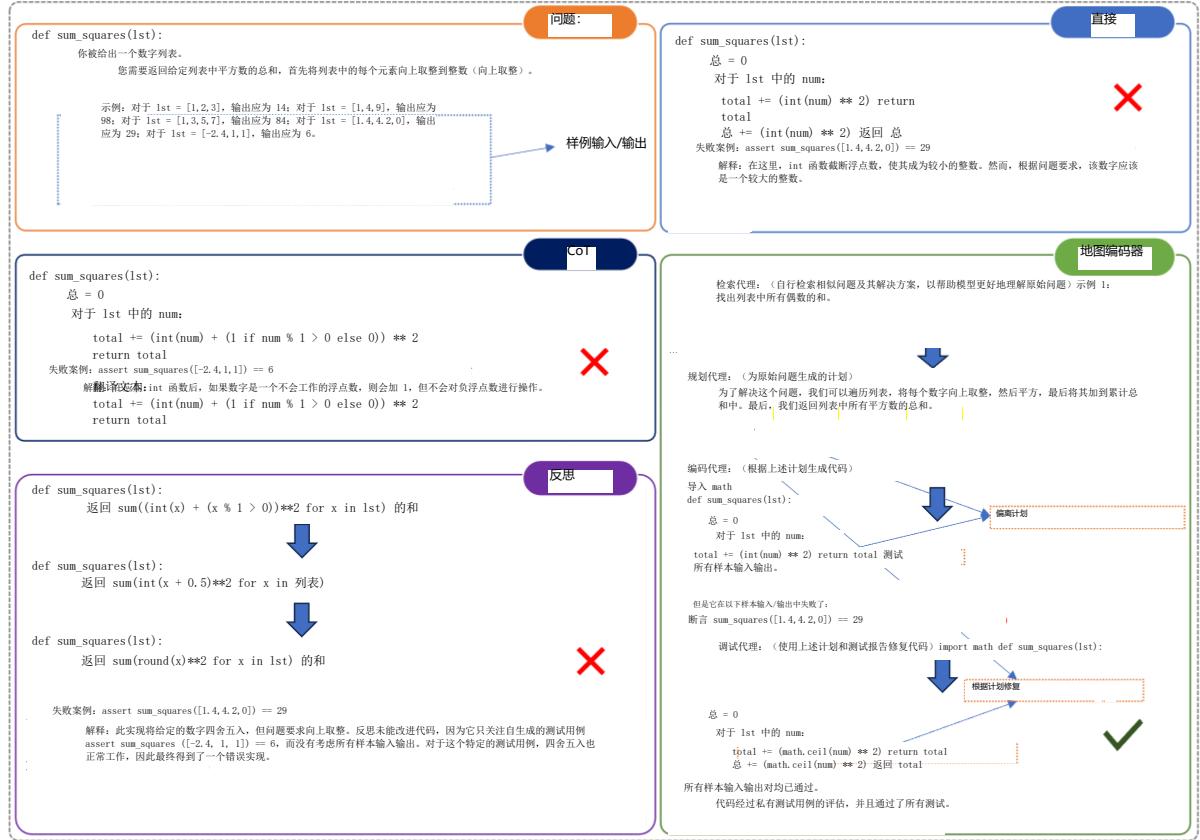


图 4: 使用 Direct、CoT、Reflexion 和 MapCoder 提示生成示例问题和解决方案。
MapCoder 首先探索高效用计划，并独特地提供基于计划的调试功能以修复错误。

人类评估（陈等, 2021a），人类评估 ET（董等, 2023a），评估增强（刘等, 2023），MBPP（奥斯汀等, 2021），以及 MBPPET（董等, 2023a）。人类评估-ET 和评估增强扩展了人类评估和 MBPP-ET，MBPP-ET 通过包含更多测试案例来包含 MBPP。人类评估和 MBPP（及其扩展）的问题集大小分别为 164 和 397。由于 MBPP 和 MBPPET 缺乏样本 I/O，我们的代码审查方法涉及从 MBPP-ET 中随机删除每个问题的一个测试案例，并将此测试案例作为问题的样本 I/O 提供。重要的是，这个被删除的测试案例被仔细选择，以确保与 MBPP 和 MBPP-ET 中的隐藏测试集相互排斥。三个竞技编程数据集是：自动化编程进展标准（APPS），xCodeEval（汗等, 2023），和 CodeContest，在我们的实验中分别使用了 150（从测试集中随机选择）、106（紧凑集）和 165（测试集）问题。

提示指导语言模型在不明确指导的情况下生成代码，依靠其固有的能力。思维链提示（CoT）（Wei 等, 2022b）将问题分解为逐步解决方案，从而有效地解决复杂任务。自我规划提示（Jiang 等, 2023b）将代码生成任务分为规划和实施阶段。类比推理提示（Yasunaga 等, 2023）指导模型回忆训练数据中的相关问题。反思（Shinn 等, 2023）根据单元测试结果提供口头反馈以增强解决方案。自我协作（Dong 等, 2023b）提出一个框架，其中不同的LLMs分别充当分析师、编码者和测试员，共同为复杂任务生成代码，其性能优于直接使用单个LLM。AlphaCodium（Ridnik 等, 2024）根据AI生成的输入输出测试迭代优化代码。

4.3 基础模型、评估指标、k 和 t

4.2 基准

我们比较了 MapCoder 与几个基线和最先进的方法。直接

在 HumanEval 中， $k = t = 5$ ，对于其他情况， $k = t = 3$ ，我们使用 ChatGPT (gpt-3.5-turbo-1106)、GPT-4 (gpt-4-1106-preview) 评估所

LLM	Approach	Simple Problems					Contest-Level Problems		
		HumanEval	HumanEval ET	EvalPlus	MBPP	MBPP ET	APPS	xCodeEval	CodeContest
ChatGPT	Direct	48.1%	37.2%	66.5%	49.8%	37.7%	8.0%	17.9%	5.5%
	CoT	68.9%	55.5%	65.2%	54.5%	39.6%	7.3%	23.6%	6.1%
	Self-Planning	60.3%	46.2%	-	55.7%	41.9%	9.3%	18.9%	6.1%
	Analogical	63.4%	50.6%	59.1%	70.5%	46.1%	6.7%	15.1%	7.3%
	Reflexion	67.1%	49.4%	62.2%	73.0%	47.4%	-	-	-
	Self-collaboration	74.4%	56.1%	-	68.2%	49.5%	-	-	-
	MapCoder	80.5% ↑ 67.3%	70.1% ↑ 88.5%	71.3% ↑ 7.3%	78.3% ↑ 57.3%	54.4% ↑ 44.3%	11.3% ↑ 41.3%	27.4% ↑ 52.6%	12.7% ↑ 132.8%
GPT4	Direct	80.1%	73.8%	81.7%	81.1%	54.7%	12.7%	32.1%	12.1%
	CoT	89.0%	61.6%	-	82.4%	56.2%	11.3%	36.8%	5.5%
	Self-Planning	85.4%	62.2%	-	75.8%	50.4%	14.7%	34.0%	10.9%
	Analogical	66.5%	48.8%	62.2%	58.4%	40.3%	12.0%	26.4%	10.9%
	Reflexion	91.0%	78.7%	81.7%	78.3%	51.9%	-	-	-
	MapCoder	93.9% ↑ 17.2%	82.9% ↑ 12.4%	83.5% ↑ 2.2%	83.1% ↑ 2.5%	57.7% ↑ 5.5%	22.0% ↑ 73.7%	45.3% ↑ 41.2%	28.5% ↑ 135.1%

Table 2: Pass@1 results for different approaches. The results of the yellow and blue colored cells are obtained from [Jiang et al. \(2023b\)](#) and [Shinn et al. \(2023\)](#), respectively. The results of the Self-collaboration [Dong et al. \(2023b\)](#) paper are collected from their paper. The green texts indicate the state-of-the-art results, and the red text is gain over Direct Prompting approach.

from OpenAI and [Gemini Pro](#) from Google. We have also evaluated our method using an open-source LLM, Mistral-7B-instruct. We have used the Pass@k evaluation metric, where the model is considered successful if at least one of the k generated solutions is correct.

5 Results

In this section, we evaluate the code generation capabilities of our framework, MapCoder, for competitive problem solving. Our experimental results are reported in Table 2. Overall, MapCoder shows a tremendous excellence in code generation, significantly outperforms all baselines, and achieves new state-of-the-art results in all benchmarks. In general the scales with GPT-4 are higher than ChatGPT.

5.1 Performance on basic code generation

The highest scale of performance (Pass@1) scores are observed in simple program synthesis tasks like HumanEval, MBPP in Table 2. Though with the simpler problem (non-contests) datasets such as HumanEval, HumanEval-ET, the current state-of-the-art method, Reflexion ([Shinn et al., 2023](#)) perform reasonably well, this approach does not generalize across varying datasets depicting a wide variety of problems. Self-reflection techniques enhance

GPT-4’s performance on HumanEval but result in a 3% decrease on the MBPP dataset. Similarly, with ChatGPT, there’s a notable 26.3% drop in performance where in several cases their AI generated test cases are incorrect. We observe that 8% of failures in HumanEval and 15% in MBPP is caused by their AI generates incorrect test cases while our approach is independent of AI test cases, and consistently improves code generations in general. Consequently, even in HumanEval, with GPT-4, our Pass@1 surpasses Reflexion by ~3%. On top, in all four simple programming datasets, MapCoder enhances the Direct prompting significantly with a maximum of 88% on HumanEvalET by ChatGPT.

5.2 Performance on competitive problem solving

The significance of MapCoder shines through clearly when evaluated in competitive problem-solving contexts. Across datasets such as APPS, xCodeEval, and CodeContests, MapCoder demonstrates substantial enhancements over Direct prompting methods, with improvements of 41.3%, 52.6%, and 132.8% for ChatGPT, and 73.7%, 41.2%, and 135.1% for GPT4, respectively. Notably, the most challenging datasets are APPS and CodeContest, where MapCoder’s performance stands out prominently. We deliberately com-

		简单问题竞赛级问题									
LLM 人类评估方法		人类评估 ET		评估加 MBPP		MBPP ET		应用 xCodeEval 编程竞赛			
ChatGPT	直接	48.1%	37.2%	66.5%	49.8%	37.7%	8.0%	17.9%	5.5%		
	CoT	68.9%	55.5%	65.2%	54.5%	39.6%	7.3%	23.6%	6.1%		
	自我规划	60.3%	46.2%	-	55.7%	41.9%	9.3%	18.9%	6.1%		
	类推	63.4%	50.6%	59.1%	70.5%	46.1%	6.7%	15.1%	7.3%		
	反刍	67.1%	49.4%	62.2%	73.0%	47.4%	--	自协作	74.4%	56.1%	-
	地图编码器	80.5%		↑ 67.3%	70.1%		↑ 7.3%	78.3%	54.4%	11.3%	27.4%
GPT4	直接	80.1%	73.8%	81.7%	81.1%	54.7%	12.7%	32.1%	12.1%		
	CoT	89.0%	61.6%	- 82.4%	56.2%	11.3%	36.8%	5.5%	自我规划	85.4%	62.2%
	类似性	66.5%	48.8%	62.2%	58.4%	40.3%	12.0%	26.4%	10.9%	-	75.8%
	反刍	91.0%	78.7%	81.7%	78.3%	51.9%	--	--	--	14.7%	34.0%
	地图编码器	93.9%		↑ 17.2%	82.9%		↑ 12.4%	83.5%	↑ 2.2%	83.1%	↑ 2.5%
								57.7%	↑ 5.5%	57.7%	↑ 73.7%
										22.0%	45.3%
										↑ 41.2%	↑ 135.1%
										28.5%	↑ 135.1%

表 2：不同方法的 Pass@1 结果。黄色和蓝色单元格的结果分别来自 Jiang 等 (2023b) 和 Shinn 等 (2023)。Self-collaboration Dong 等 (2023b) 论文的结果来自其论文。绿色文字表示最先进的结果，红色文字表示相对于 Direct Prompting 方法的增益。

来自 OpenAI 和 Google 的 Gemini Pro。我们还使用开源的 LLM、Mistral-7B-instruct 评估了我们的方法。我们使用了 Pass@k 评估指标，如果至少有一个生成的 k 个解决方案是正确的，则认为模型是成功的。

5 结果

在这一节中，我们评估了我们框架 MapCoder 在竞争性问题解决中的代码生成能力。我们的实验结果见表 2。总体而言，MapCoder 在代码生成方面表现出卓越的才能，显著优于所有基线，并在所有基准测试中实现了新的最先进成果。一般来说，与 GPT-4 相比，MapCoder 的规模更高。

自我反思技术提高了 GPT-4 在 HumanEval 上的表现，但在 MBPP 数据集上导致 3% 的下降。同样，对于 ChatGPT 来说，性能下降了 26.3%，在几个案例中，它们生成的 AI 测试用例是错误的。我们观察到，HumanEval 中有 8% 的失败是由 AI 生成错误的测试用例引起的，而我们的方法与 AI 测试用例无关，并且始终如一地提高了代码生成。因此，即使在 HumanEval 中，使用 GPT-4，我们的 Pass@1 也超过了 Reflexion 约 3%。此外，在所有四个简单的编程数据集中，MapCoder 通过 ChatGPT 显著增强了直接提示，在 HumanEval-ET 上的最大提升达到 88%。

5.2 竞技问题解决的表现

5.1 基本代码生成性能

在表 2 中观察到最高性能尺度 (Pass@1) 得分出现在像 HumanEval、MBPP 这样的简单程序综合任务中。尽管在像 HumanEval、HumanEval-ET 这样的简单问题 (非竞赛) 数据集上，当前最先进的方法 Reflexion (Shinn 等人，2023) 表现相当不错，但这种方法并不能推广到描述各种各样问题的不同数据集上。

当在竞争性问题解决环境中评估时，MapCoder 的重要性表现得非常明显。在 APPS、xCodeEval 和 CodeContests 等数据集中，MapCoder 相较于直接提示方法表现出显著的提升，ChatGPT 分别提高了 41.3%、52.6% 和 132.8%，GPT4 分别提高了 73.7%、41.2% 和 135.1%。值得注意的是，最具挑战性的数据集是 APPS 和 CodeContest，MapCoder 在这些数据集中的表现尤为突出。我们故意进行了——

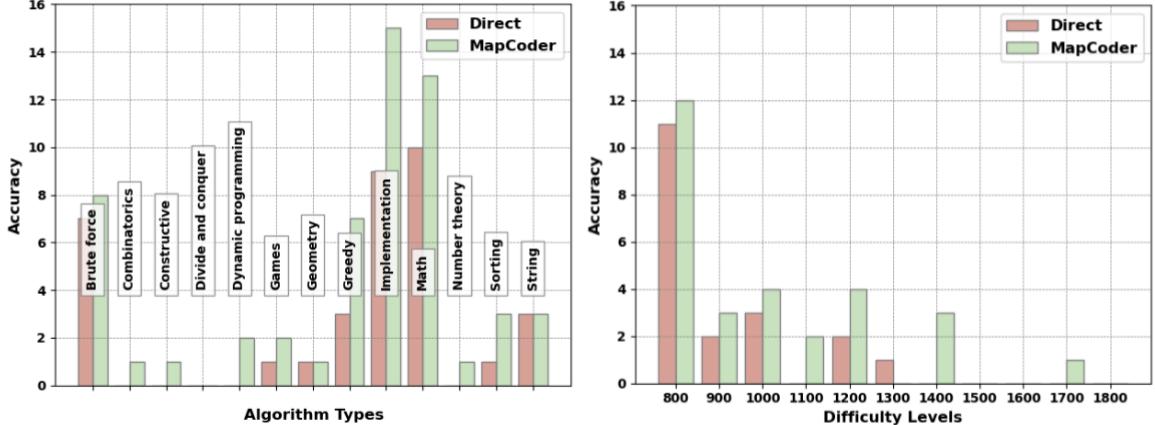


Figure 5: The number of correct answers wrt algorithm types (tags) and difficulty levels (xCodeEval dataset).

pare against strong baselines on these datasets, regardless of whether they are prompt-based or not. Importantly, on CodeContest our Pass@1 results match the Pass@5 scores of the concurrent state-of-the-art model AlphaCodium (Ridnik et al., 2024): 28.5% vs. their 29% (see Table 3). Furthermore, our Pass@5 results demonstrate an additional improvement of 12.8%. On APPS, MapCoder consistently surpasses the Pass@1 scores of all baseline prompts for both ChatGPT and GPT-4.

CodeContest (Pass@5)		
Approach	ChatGPT	GPT4
Direct	11.2%	18.8%
AlphaCodium	17.0%	29.0%
MapCoder	18.2% (↑ 63.1%)	35.2% (↑ 87.1%)

Table 3: Pass@5 results on CodeContest dataset. AlphaCodium result are from Ridnik et al. (2024). The green cells indicate the SoTA and the red text indicates improvement w.r.t Direct approach.

5.3 Performance with Varying Difficulty Levels

The APPS dataset comprises problems categorized into three difficulty levels: (i) Introductory, (ii) Interview, and (iii) Competition. Figure 6 illustrates the performance of various competitive approaches for these three categories. The results reveal that our MapCoder excels across all problem categories, with highest gain in competitive problem-solving indicating its superior code generation capabilities in general, and on top, remarkable effectiveness in competitive problem-solving. In order to gather more understanding on what algorithm problems it's capable of solving and in fact much difficulty level it can solve, we have also conducted a comparison between MapCoder and the Direct approach,

considering the difficulty levels¹ and tags² present in the xCodeEval dataset. The results of this comparison are depicted in Figure 5. This comparison showcases that MapCoder is effective across various algorithm types and exhibits superior performance even in higher difficulty levels, compared to the Direct approach. However, beyond (mid-level: difficulties>1000), its gains are still limited.

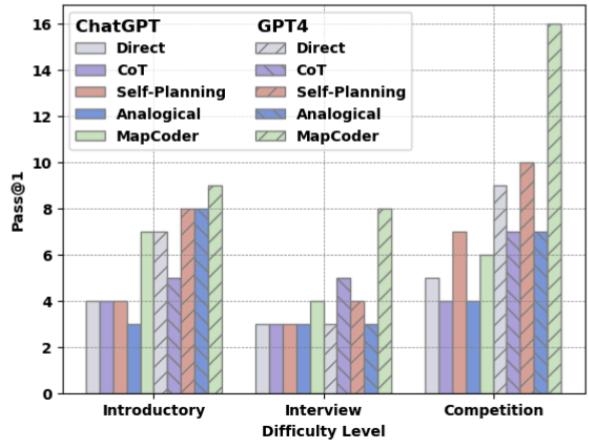


Figure 6: Performance vs problem types (APPS).

5.4 Performance Across Different LLMs

To show the robustness of MapCoder across various LLMs, we evaluate MapCoder using Gemini Pro, a different family of SoTA LLM in Table 4. We also evaluate MapCoder using an open-source LLM Mistral-7B instruct in Table 5. As expected, our method shows performance gains over other baseline approaches in equitable trends on both simple (HumanEval) and contest-level problems (CodeContest).

¹Difficulty levels in xCodeEval dataset represents an integer number, a higher value means more difficult problem

²Tags in xCodeEval dataset represents algorithm type that can be used to solve the problem i.e., greedy, dp, brute-force, constructive, and so on.

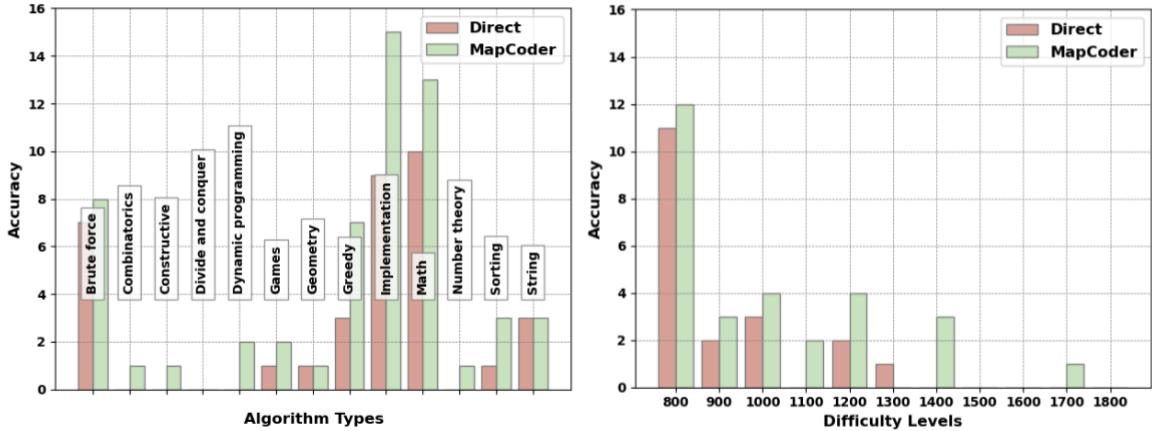


图 5：关于算法类型（标签）和难度级别（xCodeEval 数据集）的正确答案数量。

我们有意识地在这些数据集上与强大的基线进行比较，无论它们是否基于提示。重要的是，在 CodeContest 上，我们的 Pass@1 结果与同期最先进的模型 AlphaCodium (Ridnik 等人, 2024) 的 Pass@5 分数相当：28.5% 与他们的 29% (见表 3)。此外，我们的 Pass@5 结果还显示出额外的 12.8% 的改进。在 APPS 上，MapCoder 持续超越 ChatGPT 和 GPT-4 所有基线提示的 Pass@1 分数。

代码竞赛 (Pass@5) 方法 聊天机器人 GPT GPT4				
直接	11.2%	18.8%	AlphaCodium	17.0% 29.0%
地图编码器	18.2%	(↑ 63.1%)	35.2%	(↑ 87.1%)

表 3：CodeContest 数据集上的 Pass@5 结果。
AlphaCodium 结果来自 Ridnik 等人 (2024 年)。绿色单元格表示 SoTA，红色文字表示相对于直接方法的改进。

5.3 表现与不同难度级别

APPS 数据集包含分为三个难度级别的题目：(一) 入门级，(二) 面试级，(三) 竞赛级。图 6 展示了针对这三个类别的各种竞争方法的性能。结果显示，我们的 MapCoder 在所有问题类别中表现优异，在竞争性问题解决中的最高增益表明其在代码生成能力上具有优越性，并且，在竞争性问题解决方面表现出显著的有效性。为了更深入地了解它能解决哪些算法问题以及实际上能解决多少难度级别的问题，我们还对 MapCoder 和直接方法进行了比较。

考虑到 xCodeEval 数据集中存在的难度级别和标签。本比较的结果如图 5 所示。该比较表明，MapCoder 在各种算法类型中均有效，并且与直接方法相比，即使在更高的难度级别上，其性能也表现出优越性。然而，在 (中等难度：难度>1000) 之后，其收益仍然有限。

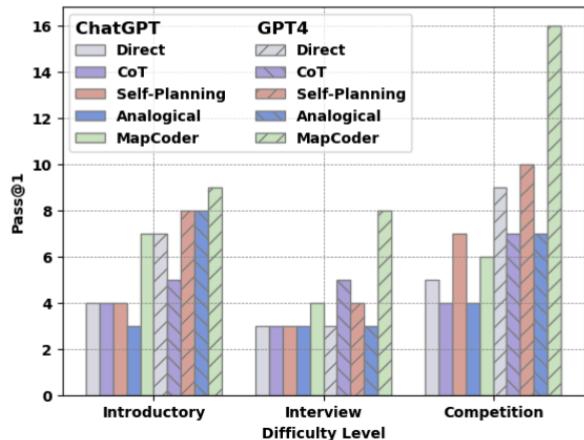


图 6：性能与问题类型 (APPS) 对比。

5.4 性能在不同LLMs上的表现

为了展示 MapCoder 在多种场景下的鲁棒性，我们使用 Gemini Pro 评估 MapCoder，它在表 4 中属于 SoTA 的不同系列。我们还使用开源的 Mistral-7B 指令在表 5 中评估 MapCoder。正如预期的那样，我们的方法在简单 (HumanEval) 和竞赛级问题 (CodeContest) 的公平趋势上，相较于其他基线方法均表现出性能提升。

xCodeEval 数据集中的难度级别表示为整数，数值越高表示问题越难
标签在 xCodeEval 数据集中代表可用于解决问题的算法类型，例如贪心、动态规划、暴力搜索、构造性等。

LLM	Approach	HumanEval	CodeContest
Gemini	Direct	64.6%	3.6%
	CoT	66.5%	4.8%
	MapCoder	69.5% (↑ 7.5%)	4.8% (↑ 32.0%)

Table 4: Pass@1 results with using Gemini Pro. The red text is gain over Direct Prompting approach.

LLM	Approach	HumanEval	HumanEval-ET
Mistral	Direct	27.3%	27.3%
	CoT	45.5%	42.4%
	MapCoder	57.6% (↑ 111.1%)	48.5% (↑ 77.8%)

Table 5: Pass@1 results with using Mistral-7B-instruct. The red text is gain over Direct Prompting approach.

5.5 Performance Across Different Programming Languages

Furthermore, we evaluate model performances using MapCoder across different programming languages. We utilize the xCodeEval dataset, which features multiple languages. Figure 7 shows that consistent proficiency across different programming languages is achieved by MapCoder with respect to baselines.

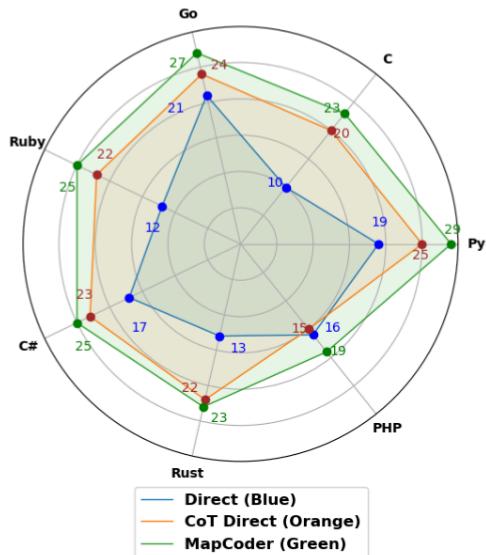


Figure 7: The number of correct answers wrt different programming languages (xCodeEval dataset).

6 Ablations Studies and Analyses

We present the ablation study of the MapCoder on HumanEval dataset as the problems are simpler and easy to diagnose by us humans.

6.1 Impact of Different Agents

We have also conducted a study by excluding certain agents from our MapCoder, which helps us investigate each agent’s impact in our whole pipeline.

Retrieval Agent	Planning Agent	Debugging Agent	Pass@1	Performance Drop
✗	✗	✓	68.0%	15.0%
✗	✓	✓	76.0%	5.0%
✗	✓	✗	52.0%	35.0%
✓	✗	✓	70.0%	12.5%
✓	✓	✗	66.0%	17.5%
✓	✗	✗	62.0%	22.5%
✓	✓	✓	80.0%	-

Table 6: Pass@1 results for different versions of MapCoder (by using ChatGPT on HumanEval dataset).

As expected, the results (Table 6) show that every agent has its role in the pipeline as turning off any agent decreases the performance of MapCoder. Furthermore, we observe that the Debugging Agent has the most significant impact on the pipeline, as evidenced by a performance drop of 17.5% when excluding this agent exclusively, and an avg performance drop of 24.83% in all cases. The *Planning agent* has the second best important with avg drop of 16.7% in all cases. In Table 6), we perform an ablation study of our multi-agent framework investigate each agent’s impact in our whole pipeline.

6.2 Qualitative Example

To verify the above numerical significance, and to understand how our method enhance the code generation, we have performed a qualitative analysis to find the underlying reason for the superior performance of MapCoder over other competitive prompting approaches. An example problem and the output with the explanation of Direct, CoT, Reflexion, and MapCoder prompting is shown in Figure 4. This example demonstrates how the *Debugging Agent* fixes the bugs leveraging the plan as a guide from the *Planning Agent*. This verifies the impact of these two most significant agents. We present more detailed examples in Appendix.

6.3 Impact of k and t

MapCoder involves two hyper-parameters: the number of self-retrieved exemplars, k , and the number of debugging attempts, t . Our findings (Table 7) reveal that higher k, t is proportionate performance gain at the expense of time.

Dataset Name	$\frac{t}{k}$	0	3	5
HumanEval	3	62.8%	76.8%	80.5%
	5	65.9%	79.9%	80.5%
HumanEval-ET	3	57.3%	61.0%	70.1%
	5	57.9%	67.1%	67.1%

Table 7: Pass@1 results by varying k and t .

方法	直接	CoT	MapCoder
双子	64.6% 3.6%		
	66.5% 4.8%		
		69.5% (17.5%) 4.8% (132.0%)	

表 4：使用 Gemini Pro 的 Pass@1 结果。红色文字表示相对于直接提示方法的增益。

LLM 方法	HumanEval	HumanEval-ET
迷斯特拉尔	直接 27.3%	27.3%
	CoT 45.5%	42.4%
	地图编码器	57.6% (\uparrow 111.1%) 48.5% (\uparrow 77.8%)

表 5：使用 Mistral-7B-instruct 的 Pass@1 结果
红色文本是相对于直接提示方法的增益。

5.5 跨不同编程语言的性能

此外，我们使用 MapCoder 在不同编程语言中评估模型性能。我们利用包含多种语言的 xCodeEval 数据集。图 7 显示，MapCoder 在基准测试中实现了在不同编程语言上的持续熟练度。

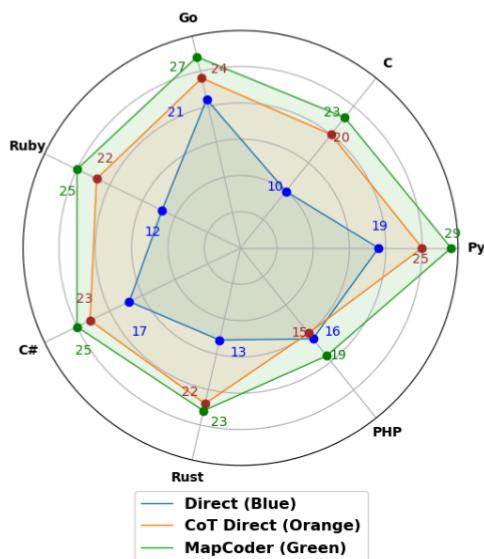


图 7：针对不同编程语言的正确答案数量（xCodeEval 数据集）。

6 消融研究与分析

我们展示了 MapCoder 在 HumanEval 数据集上的消融研究，因为问题对我们人类来说更简单且易于诊断。

6.1 不同因素的影响

我们还将某些代理从 MapCoder 中排除，进行了一项研究，这有助于我们调查每个代理在整个流程中的影响。

表 6: 不同版本的 MapCoder 在 HumanEval 数据集上使用 ChatGPT 的 Pass@1 结果。

如预期所示，结果（表 6）显示每个代理在流水线中都有其作用，关闭任何代理都会降低 MapCoder 的性能。此外，我们观察到调试代理对流水线的影响最为显著，如仅排除此代理时性能下降 17.5%，在所有情况下平均性能下降 24.83%。规划代理在所有情况下平均下降 16.7%，是第二重要的。在表 6 中，我们对我们多代理框架进行了消融研究，以调查每个代理在我们整个流水线中的影响。

6.2 定性示例

为验证上述数值意义，并了解我们的方法如何提升代码生成，我们进行了定性分析，以找出 MapCoder 相较于其他竞争性提示方法的优越性能背后的原因。图 4 展示了包含直接、CoT、Reflexion 和 MapCoder 提示的示例问题和输出，并对其进行了说明。此示例演示了调试代理如何利用规划代理的规划作为指南来修复错误。这验证了这两个最显著代理的影响。我们在附录中提供了更详细的示例。

6.3 影响

6.3 影响 k 很抱歉，您提供的源文本 "and t " 缺少足够的信息来进行准确的翻译。请提供完整的学术文本以 MapCoder 涉及两个参数 k （检索示例的数量）和 t （调试尝试的数量）。我们的发现（表 7）显示，较高的 k 、 t 与性能提升成正比，但以时间为代价。

数据集名称	0	3	5	t				
人类评估	3	62.8%	76.8%	80.5%	5	65.9%	79.9%	80.5%
人类评估-ET	3	57.3%	61.0%	70.1%	5	57.9%	67.1%	67.1%

表 7: 不同 k 和 t 值下的 Pass@1 结果。

LLM	Dataset	Average for MapCoder		Average for Direct Prompting		Accuracy Enhancement
		API Calls	Tokens (k)	API Calls	Tokens (k)	
ChatGPT	HumanEval	17	10.41	1	0.26	67.3%
	MBPP	12	4.84	1	0.29	57.3%
	APPS	21	26.57	1	0.66	41.3%
	xCodeEval	19	24.10	1	0.64	52.6%
	CodeContest	23	34.95	1	0.80	132.8%
GPT4	HumanEval	15	12.75	1	0.43	17.2%
	MBPP	8	4.96	1	0.57	2.5%
	APPS	19	31.80	1	0.82	73.7%
	xCodeEval	14	23.45	1	0.85	41.2%
	CodeContest	19	38.70	1	1.11	135.1%
Average		16.7	21.25	1	0.64	62.1%

Table 8: Average number of API calls, thousands of tokens used, required time in minutes to get the API response.

6.4 Impact of Number of Sample I/Os

Given the limited number of sample I/Os in the HumanEval dataset (average of 2.82 per problem), we supplemented it with an additional 5 sample I/Os from the HumanEval-ET dataset. Experiments with this augmented set showed an 1.5% performance gain.

6.5 Error Analysis and Challenges

Although MapCoder demonstrates strong performance compared to other methods, it faces challenges in certain algorithmic domains. For example, Figure 5 illustrates MapCoder’s reduced performance on more difficult problems requiring precise problem understanding and concrete planning—capabilities still lacking in LLMs. In the xCodeEval dataset (see Figure 5), it solves a limited number of problems in categories like Combinatorics, Constructive, Number Theory, Divide and Conquer, and Dynamic Programming (DP). Manual inspection of five DP category problems reveals occasional misinterpretation of problems, attempts to solve using greedy or brute-force approaches, and struggles with accurate DP table construction when recognizing the need for a DP solution.

7 Conclusion and Future Work

In this paper, we introduce MapCoder, a novel framework for effective code generation in complex problem-solving tasks, leveraging the multi-agent prompting capabilities of LLMs. MapCoder captures the complete problem-solving cycle by employing four agents - retrieval, planning, coding, and debugging - which dynamically interact to produce high-quality outputs. Evaluation across major benchmarks, including basic and competitive programming datasets, demonstrates MapCoder’s

consistent outperformance of well-established baselines and SoTA approaches across various metrics. Future work aims to extend this approach to other domains like question answering and mathematical reasoning, expanding its scope and impact.

8 Limitations

Among the limitations of our work, firstly, MapCoder generates a large number of tokens, which may pose challenges in resource-constrained environments. Table 8 shows the number of average API calls and token consumption with the default k and t (i.e., with respect to the reported performance) while Table 7 shows how k, t can be adjusted to proportionate the performance gain at the expense of time/token. We have not addressed the problem of minimizing tokens/API-calls in this paper and leave it for future works. Secondly, our method currently relies on sample input-output (I/O) pairs for bug fixing. Although sample I/Os provide valuable insights for LLMs’ code generation, their limited number may not always capture the full spectrum of possible test cases. Consequently, enhancing the quality of additional test case generation could reduce our reliance on sample I/Os and further improve the robustness of our approach. Additionally, future exploration of open-source code generation models, such as CodeLLaMa, LLaMa3, Mixtral 8x7B could offer valuable insights and potential enhancements to our approach. Another important concern is that while running machine-generated code, it is advisable to run it inside a sandbox to avoid any potential risks.

Acknowledgements

We thank Tanvir Parvez (Department of Statistics, Islamic University, Bangladesh) and QCRI for their efforts and resources in running our experiments.

LLM	数据集	API 调用令牌 (k)	平均编码器平均直接提示准确率			提升
			人类评估	MBPP	APPs	
ChatGPT	人类评估	17	10.41	1.0.26	67.3%	
	MBPP	12	4.84	1.0.29	57.3%	
	APPs	21	26.57	1.0.66	41.3%	
	xCodeEval	19	24.10	1.0.64	52.6%	
	代码竞赛	23	34.95	1.0.80	132.8%	
GPT4	人类评估	15	12.75	1.0.43	17.2%	
	MBPP	8	4.96	1.0.57	2.5%	
	APPs	19	31.80	1.0.82	73.7%	
	xCodeEval	14	23.45	1.0.85	41.2%	
	代码竞赛	19	38.70	1.1.11	135.1%	
平均			16.7	21.25	1.0.64	62.1%

表 8：平均 API 调用次数，使用的令牌数（千），获取 API 响应所需时间（分钟）。

6.4 样本 I/O 数量影响

考虑到 HumanEval 数据集中样本 I/O 数量有限（每个问题平均 2.82 个），我们从 HumanEval-ET 数据集中补充了 5 个样本 I/O。使用这个增强集进行的实验显示性能提升了 1.5%。

6.5 错误分析与挑战

尽管与其它方法相比 MapCoder 表现出强大的性能，但在某些算法领域它面临着挑战。例如，图 5 展示了 MapCoder 在需要精确问题理解和具体规划的更难问题上的性能降低——这些能力在LLMs中仍然缺乏。在 xCodeEval 数据集（见图 5）中，它仅解决了组合数学、构造性、数论、分而治之以及动态规划（DP）等类别中有限数量的问题。对五个 DP 类别问题的手动检查揭示了偶尔的问题误解、尝试使用贪婪或暴力方法解决问题，以及当需要 DP 解决方案时，在准确构建 DP 表时的困难。

展示了 MapCoder 在各种指标上持续优于已建立基线和 SoTA 方法的性能。未来的工作旨在将这种方法扩展到其他领域，如问答和数学推理，扩大其范围和影响。

8 局限性

在本文工作的局限性中，首先，MapCoder 生成了大量标记，这在资源受限的环境中可能带来挑战。表 8 显示了默认 k 和 t 值下的平均 API 调用次数和标记消耗量（即与报告的性能相关），而表 7 显示了如何调整 k、t 以在牺牲时间/标记的代价下成比例地提高性能。本文未解决最小化标记/API 调用次数的问题，将其留待未来工作。其次，我们目前的方法依赖于样本输入/输出（I/O）对进行错误修复。尽管样本 I/O 为LLMs代码生成提供了有价值的见解，但它们的数量有限，可能无法始终捕捉到所有可能的测试用例的全貌。因此，提高额外测试用例生成质量可以减少我们对样本 I/O 的依赖，并进一步提高我们方法的鲁棒性。此外，未来对开源代码生成模型（如 CodeLLaMa、LLaMa3、Mixtral 8x7B）的探索可能为我们提供有价值的见解和潜在的提升。另一个重要问题是，在运行机器生成的代码时，建议在沙盒中运行以避免任何潜在风险。

7 结论与未来工作

在这篇论文中，我们介绍了 MapCoder，这是一个用于复杂问题解决任务中有效代码生成的创新框架，利用了LLMs的多智能体提示能力。MapCoder 通过采用检索、规划、编码和调试四个智能体，动态交互以产生高质量的输出，从而捕捉完整的解决问题周期。在包括基础和竞技编程数据集在内的主要基准测试中进行了评估。

致谢

我们感谢 Tanvir Parvez（孟加拉国伊斯兰大学统计学系）和 QCRI 在运行我们的实验中所付出的努力和资源。

References

- Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.
- Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don’t reach for the stars! *arXiv preprint arXiv:2301.03988*.
- Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zотов. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.
- 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, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.
- Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023a. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.
- Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.
- Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 763–773, New York, NY, USA. ACM.
- Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.
- Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lampe, Lucile Saulnier, Lélio Renard Lavaud,

参考文献

- Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, 和 Kai-Wei Chang. 2021. 用于程序理解和生成的统一预训练。arXiv 预印本 arXiv:2103.06333。
- Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, 等人. 2023. Santacoder: 不要向星星伸手! arXiv 预印本 arXiv:2301.03988。
- 雅各布·安德烈亚斯, 约翰·布费, 大卫·伯克特, 查尔斯·陈, 乔希·克劳斯曼, 珍·克劳福德, 凯特·克里姆, 乔丹·德洛奇, 莉亚·多纳, 贾森·艾斯纳, 方浩, 郭艾伦, 大卫·霍尔, 克里斯汀·海斯, 凯莉·希尔, 霍莉·何, 温迪·伊瓦苏克, 斯里米蒂·贾, 丹·克莱因, 贾扬特·克里什纳穆尔蒂, 西奥·兰曼, 梁珀西, 林克里斯托弗·H, 伊利亚·林茨巴赫, 安迪·麦戈文, 亚历山大·尼森维奇, 亚当·保罗斯, 德米特里·彼得斯, 布伦特·里德, 丹·罗斯, 苏布罗·罗伊, 杰西·鲁萨克, 贝丝·肖特, 迪夫·斯洛明, 本·辛德, 斯蒂芬·Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. 任务导向对话作为数据流合成。计算语言学协会汇刊, 8: 556–571。
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan蔡凯丽, 迈克尔·特里, 吴国乐, 等。2021。使用大型语言模型的程序综合。arXiv 预印本 arXiv:2108.07732。
- 陈贝, 张峰吉, 阮安, 赞道光, 林泽琪, 楼建光, 陈伟竹。2022 年。
Codet: 代码生成与测试生成。arXiv 预印本 arXiv:2207.10397。
- 陈马克, 杰瑞特沃雷克, 崔慧宇, 袁奇明, 奥利维拉·平托·亨里克, 贾里德·卡普兰, 哈里·爱德华兹, 尤里·伯达, 尼古拉斯·约瑟夫, 格雷格·布罗克曼, 亚历克斯·雷, 劳尔·普里, 格雷琴·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, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebbgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. 在代码上训练的大型语言模型。arXiv 预印本 arXiv:2107.03374。
- 陈欣云, 林马克斯韦尔, 沙里尔·纳撒尼尔, 周登尼。2023。教会大型语言模型自我调试。arXiv 预印本 arXiv:2304.05128。
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, 等人. 2022. Palm: 通过路径扩展语言模型。arXiv 预印本 arXiv:2204.02311。
- 董一鸿, 丁家正, 蒋雪, 李卓, 李鸽, 金志。2023a. Codescore: 通过学习代码执行来评估代码生成。arXiv 预印本 arXiv:2301.09043。
- 董一鸿, 蒋雪, 金志, 李戈。2023b.
- 通过 ChatGPT 实现的自协作代码生成。
- 张银凤, 郭大亚, 唐杜宇, 端楠, 冯晓程, 龚明, 寿林君, 秦冰, 刘婷, 姜大新, 等。2020。Codebert: 一种用于编程和自然语言的预训练模型
《计算语言学协会发现: EMNLP 2020, 第 1536–1547 页》。
- 丹尼尔·弗里德, 阿门·阿加亚尼安, 杰西·林, 王思达, 埃里克·华莱士, 弗雷达·石, 钟瑞琪, 叶文涛
Luke Zettlemoyer, Mike Lewis. 2022. Incoder: 一种用于代码补全和合成的生成模型。arXiv 预印本 arXiv:2204.05999。
- Sumit Gulwani. 2011. 使用输入输出示例自动处理电子表格中的字符串。ACM Sigplan Notices, 46(1): 317–330。
- 郭垚, 朱启豪, 杨德健, 谢振达, 董凯, 张文涛, 陈冠廷, 毕晓, 吴 Y, 李 YK, 等。2024. Deepseek-coder: 大型语言模型遇见编程——代码智能的崛起。arXiv 预印本 arXiv:2401.14196。
- Vincent J. Hellendoorn 和 Premkumar Devanbu. 2017.
深度神经网络是否是最佳模型选择
源代码? 在 2017 年第 11 届软件工程基础联合会议论文集中, ESEC/FSE 2017, 第 763–773 页, 纽约, 纽约州, 美国。ACM。
- 阿布拉姆·欣德尔, 厄尔·T·巴, 马克·加贝尔, 孙振东, 普雷姆库马尔·德万布。2016。《论软件的自然性》。ACM 通讯, 59(5): 122–131。
- 董黄, 卜庆文, 张杰, 迈克尔·卢克, 崔鹤鸣。2023。Agentcoder: 基于多代理的代码生成, 具有迭代测试和优化。arXiv 预印本 arXiv:2312.13010。
- 江阿尔伯特·Q, 萨布莱罗莱斯·亚历山大, 门什·亚瑟, 班福德·克里斯, 查普洛特·德文德拉·辛格, 迪戈·德·卡萨斯, 弗洛里安·布雷斯南, 吉安娜·伦格耶尔, 纪尧姆·兰普勒, 吕西勒·索利尼尔, 莱利奥·雷纳德·拉沃德

- Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. *Mistral 7b*.
- Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023b. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.
- Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multi-lingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.
- Donald E Knuth. 1992. Literate programming. *CSLI Lecture Notes, Stanford, CA: Center for the Study of Language and Information (CSLI)*, 1992.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.
- Jingyao Li, Pengguang Chen, and Jiaya Jia. 2023. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022a. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.
- Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE’07)*, pages 75–84. IEEE.
- Emilio Parisotto and Ruslan Salakhutdinov. 2017. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*.
- Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.
- Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.
- Md Rizwan Parvez, Jianfeng Chi, Wasi Uddin Ahmad, Yuan Tian, and Kai-Wei Chang. 2023. Retrieval enhanced data augmentation for question answering on privacy policies. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 201–210, Dubrovnik, Croatia. Association for Computational Linguistics.
- Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126.
- Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *CoRR*, abs/1704.07535.
- Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Kashun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12113–12139, Singapore. Association for Computational Linguistics.

- 玛丽·安妮·拉肖, 皮埃尔·斯托克, 特文·勒斯考, 蒂博·拉维尔, 王涛, 蒂莫西·克拉克, 威廉·埃尔·赛义德。2023a. *Mistral 7b*.
- 薛江, 董一鸿, 王乐成, 商奇伟, 李戈。2023b. 基于大型语言模型的自我规划代码生成。arXiv 预印本 arXiv:2303.06689。
- 穆罕默德·阿卜杜拉·马汀·汗, M·萨iful·巴里, Xuan Long Do, 魏世王, Md Rizwan Parvez, 沙菲克·乔蒂。2023. xcodeeval: 一个大规模多语言多任务基准, 用于代码理解、生成、翻译和检索。arXiv 预印本 arXiv:2303.03004。
- 唐纳德·E·克努特。1992。《文献编程》。CSLI 讲义, 加利福尼亚州斯坦福: 语言与信息研究中心 (CSLI), 1992 年。
- 黄磊, 王越, 阿基什·迪帕克·戈特马尔, 西尔维奥 Savarese, Steven Chu Hong Hoi. 2022. CoderL: 通过预训练模型和深度强化学习掌握代码生成。神经信息处理系统进展, 35: 21314–21328。
- 李静瑶, 陈鹏光, 贾佳亚。2023. Motcoder: 通过模块化思维提升大型语言模型在具有挑战性的编程任务中的性能。arXiv 预印本 arXiv:2312.15960。
- 李宇佳, 蔡大卫, 郑俊勇, 纳特·库什曼, 朱利安·施里特维瑟, 雷米·勒布隆, 汤姆·埃克莱斯, 詹姆斯·基林, 费利克斯·吉门诺, 阿古斯丁·达·拉戈, 等。2022a. Alphacode 的竞争级代码生成。科学, 378(6624):1092–1097。
- 李宇佳, 蔡大卫, 郑俊勇, 内特·库什曼, 朱利安·施里特维瑟, 雷米·勒布朗, 汤姆·埃克利斯, 詹姆斯·基林, 费利克斯·吉门诺, 奥古斯丁·达·拉戈, 托马斯·休伯特, 彼得·阜宜, 西普里恩·德·马松·德·奥图姆, 伊戈尔·巴布什金, 陈欣云, 黄普森, 约翰内斯·韦尔布, 斯文·戈瓦尔, 阿列克谢·切列帕诺夫, 詹姆斯·莫洛伊, 丹尼尔·曼科夫茨, 艾斯梅
- Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, 和 Oriol Vinyals. 2022b. Alphacode 的竞争级代码生成。
- 刘佳伟, 夏春秋, 王宇瑶, 张凌明。2023。你的代码真的是由 ChatGPT 生成的吗? 对代码生成大型语言模型的严格评估。在第三十七届 神经信息处理系统大会
- 索哈尔·马纳和理查德·J·沃尔丁格。1971。《迈向自动程序综合》。ACM 通讯, 14(3): 151–165。
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, 和 Caiming Xiong. 2022. Codegen: 一个开源的大语言模型, 用于代码的多轮程序合成。arXiv 预印本 arXiv:2203.13474。
- 卡洛斯·帕切科, 舒文杜·K·拉里, 迈克尔·D·恩斯特, 托马斯·鲍尔。2007. 基于反馈的随机测试生成。在第 29 届国际软件工程会议 (ICSE' 07), 第 75–84 页。IEEE
- 埃米利奥·帕里索托和鲁斯兰·萨拉克图丁诺夫。2017。《神经网络: 深度强化学习的结构化记忆》。arXiv 预印本 arXiv:1702.08360。
- Parvez, Md Rizwan; Ahmad, Wasi Uddin; Chakraborty, Saikat; Ray, Baishakhi; Chang, Kai-Wei. 2021. 基于检索的代码生成与摘要。arXiv 预印本 arXiv:2108.11601.
- Parvez, Md Rizwan; Chakraborty, Saikat; Ray, Baishakhi; Chang, Kai-Wei. 2018. 带有命名实体的文本语言模型构建。在《会议论文集》中。第 56 届计算语言学协会年会 (第 1 卷: 长篇论文), 第 2373–2383 页, 澳大利亚墨尔本。计算语言学协会。
- Parvez, Md Rizwan, Chi, Jianfeng, Ahmad, Wasi Uddin, Tian, Yuan, Chang, Kai-Wei. 2023. 隐私政策问答中的检索增强数据增强。第 17 届会议论文集 欧洲计算语言学协会分会会议, 第 201–210 页, 克罗地亚杜布罗夫尼克。计算语言学协会。
- Oleksandr Polozov 和 Sumit Gulwani. 2015. Flashmeta: 一种归纳程序综合框架。在 2015 年 ACM SIGPLAN 国际面向对象编程、系统、语言和应用会议论文集, 第 107–126 页。
- 拉宾诺维奇, 米切尔·斯特恩, 丹·克莱因。2017. 代码生成和语义解析的抽象语法网络。CoRR, abs/1704.07535.
- 塔勒·里德尼克, 德迪·克雷多, 伊塔马尔·弗里德曼。2024 年。 代码生成与 Alphacodium: 从提示工程到流程工程。arXiv 预印本 arXiv:2401.08500.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, 等人。2023. 代码羊: 开源代码基础模型。arXiv 预印本 arXiv:2308.12950.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, 和 Shunyu Yao. 2023. 反思: 具有言语强化的语言代理 在第三十七届神经信息处理系统会议上学习。
- 卡顺·舒姆, 刁世哲, 张彤。2023 年。 自动提示增强与选择: 基于标记数据的思维链。在《计算语言学协会发现: EMNLP 2023》, 第 12113–12139 页, 新加坡。计算语言学协会。

- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. [Llama 2: Open foundation and fine-tuned chat models](#). *arXiv preprint arXiv:2307.09288*.
- Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708.
- Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022a. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.
- Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, and Jian guang Lou. 2023. [Re-reading improves reasoning in language models](#).
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.
- Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.
- Pengcheng Yin and Graham Neubig. 2017. [A syntactic neural model for general-purpose code generation](#). *CoRR*, abs/1704.01696.
- Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019. [CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases](#). In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.
- Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. 2023. Cumulative reasoning with large language models. *arXiv preprint arXiv:2308.04371*.
- Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.
- 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. *arXiv preprint arXiv:2310.04406*.

Appendix

A Algorithm of MapCoder

Algorithm 1 shows the pseudo-code of our prompting technique.

Algorithm 1 MapCoder

```

1:  $k \leftarrow$  number of self-retrieved exemplars
2:  $t \leftarrow$  number of debugging attempts
3:
4:  $\text{exemplars} \leftarrow \text{RetrivalAgent}(k)$ 
5:
6:  $\text{plans} \leftarrow$  empty array of size  $k$ 
7: for  $\text{example}$  in  $\text{exemplars}$  do
8:    $\text{plans}[i] \leftarrow \text{PlanningAgent}(\text{example})$ 
9: end for
10:
11:  $\text{plans} \leftarrow \text{SortByConfidence}(\text{plans})$ 
12:
13: for  $i \leftarrow 1$  to  $k$  do
14:    $\text{code} \leftarrow \text{CodingAgent}(\text{code}, \text{plan}[i])$ 
15:    $\text{passed}, \log \leftarrow \text{test}(\text{code}, \text{sample\_io})$ 
16:   if  $\text{passed}$  then
17:     Return  $\text{code}$ 
18:   else
19:     for  $j \leftarrow 1$  to  $t$  do
20:        $\text{code} \leftarrow \text{DebuggingAgent}(\text{code}, \log)$ 
21:        $\text{passed}, \log \leftarrow \text{test}(\text{code}, \text{sample\_io})$ 
22:       if  $\text{passed}$  then
23:         Return  $\text{code}$ 
24:       end if
25:     end for
26:   end if
27: end for
28: Return  $\text{code}$ 
```

B Details Promptings of MapCoder

The detailed prompting of the Retrieval Agent, Planning Agent, Coding Agent, and Debugging Agent are shown in Figure 8, 9, and 10 respectively. Note that we adopt a specific sequence of instructions in the prompt for Retrieval Agent which is a crucial design choice.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale 等人. 2023. Llama 2: 开放基础和微调聊天模型。arXiv 预印本 arXiv:2307.09288。

王越, 王威诗, 沙菲克·乔蒂, 史蒂文·C·H·霍伊。2021. Codet5: 用于代码理解和生成的标识符感知统一预训练编码器-解码器模型。在 EMNLP, 第 8696-8708 页。

王志若, 荒木淳, 姜正宝, Rizwan Parvez, 格雷厄姆·纽比格。2023. 学习过滤上下文以增强检索生成。arXiv 预印本 arXiv:2311.08377。

贾森·魏, 王雪志, 戴尔·舒尔曼斯, 马腾·博斯马, 夏斐, 艾德·奇, 吴国越·莱, 周登尼, 等。2022a。思维链提示引发大型语言模型的推理。神经信息处理系统进展, 35: 24824–24837。

贾森·魏, 王雪志, 戴尔·舒尔曼斯, 马腾·博斯马, 夏斐, 艾德·奇, 吴国越·莱, 周登尼, 等。2022b。思维链提示引发大型语言模型的推理。神经信息处理系统进展, 35: 24824–24837。

许晓涵, 陶长阳, 沈涛, 许灿, 徐宏波, 龙国栋, 楼建光。

2023. 重新阅读提高语言模型中的推理能力。

姚顺宇, 余电, 赵杰夫, 伊扎克·沙法兰, 托马斯·L·格里菲斯, 曹媛, 纳拉西哈南·卡蒂克。2023. 思维之树: 大型语言模型辅助的深思熟虑问题解决。arXiv 预印本 arXiv:2305.10601。

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chien. 2023. 大型语言模型作为类比推理论者。arXiv 预印本 arXiv:2310.01714.

殷鹏程, 内布格·格雷厄姆。2017。《通用代码生成的句法神经网络模型》。
CoRR, abs/1704.01696.

陶宇, 张瑞, 尔海扬, 李思怡, 薛艾瑞克, 庞博, 林西维多利亚, 陈逸然, 石天泽, 李紫涵, 姜有轩, 山内道宏, 沈圣洛, 陈涛, 亚历山大·法布里, 紫凡

李, 吕瑶 陈, 余文 张, 斯里雅·迪希特, 张文森, 熊才明, 索 cher, 拉斯基, 德拉戈米尔·拉代夫。2019。CoSQL: 面向跨领域自然语言数据库界面的对话式文本到 SQL 挑战。载于 2019 年实证自然语言处理会议和第 9 届国际自然语言处理联合会议 (EMNLP-IJCNLP) 论文集, 第 1962-1979 页, 中国香港。计算语言学协会。

张一帆, 杨静琴, 袁洋, 姚期智。2023. 大型语言模型的累积推理。arXiv 预印本 arXiv:2308.04371。

张卓胜, 张昀, 李穆, 亚历克斯·斯莫拉。2022. 大型语言模型中的自动思维链提示。arXiv 预印本 arXiv:2210.03493。

周安迪, 严凯, 米哈尔·夏普恩托克-罗思曼, 王浩瀚, 王宇翔。2023. 语言代理树搜索统一了语言模型中的推理、行动和规划。arXiv 预印本 arXiv:2310.04406。

附录

A 算法 地图编码器

算法 1 展示了我们的提示技术的伪代码。

算法 1 MapCoder

```
1: k ← 自检索样本数量
2: t ← 调试尝试次数 3: 4: 示例
   ← 检索代理(k)
5:
6: 计划 ← 空数组, 大小为 k
示例在示例中 8: 计划[i] ← 规划代理 (示例)
9: 结束循环
10:
11: 计划 ← 按置信度排序 (plans)
12:
13: 对于 i ← 1 到 k 循环
14: 代码 ← 编码代理 (code, plan[i])
15: 通过, 日志 ← 测试 (code, sample_io)
16: 如果通过则
17:   返回码
18: 否则
19:   对于 j ← 1 到 t 循环
20:     code ← 调试代理 (code, log)
21:     通过, 日志 ← 测试 (code, sample_io)
22:     如果通过则
23:       返回代码
24:   结束 if
25: 结束循环
26: 结束 if
27: 结束循环
28: 返回码
```

B 地图编码器的提示细节

图 8、9 和 10 分别展示了检索代理、规划代理、编码代理和调试代理的详细提示。请注意, 我们在检索代理的提示中采用了特定的指令顺序, 这是一个关键的设计选择。

Retrieval Agent

Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

```

# Problem:
{Problem Description will be added here}

# Exemplars:
Recall k relevant and distinct problems (different from problem mentioned above). For each problem,
1. describe it
2. generate {language} code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:
-----
Important:
Your response must follow the following xml format-
<root>
  <problem>
    # Recall k relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.
    <description> # Describe the problem. </description>
    <code> # Let's think step by step to solve this problem in {language} programming language. </code>
    <planning> # Planning to solve this problem. </planning>
  </problem>
  # similarly add more problems here...
</algorithm>
</root>
```

Figure 8: Prompt for self-retrieval Agent.

Planning Agent

Planning Generation Prompt:

Given a competitive programming problem generate a concrete planning to solve the problem.

```

# Problem: {Description of the example problem}
# Planning: {Planning of the example problem}
## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample I/Os}
```

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

Confidence Generation Prompt:

Given a competitive programming problem and a plan to solve the problem in {language} tell whether the plan is correct to solve this problem.

```

# Problem: {Original Problem}
# Planning: {Planning of our problem from previous step}
-----
Important: Your response must follow the following xml format-
<root>
  <explanation> Discuss whether the given competitive programming problem is solvable by using the above mentioned planning. </explanation>
  <confidence> Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
  </confidence>
</root>
```

Figure 9: Prompt for Planning Agent. The example problems that are mentioned in this figure will come from the Retrieval Agent.

Coding Agent

Given a competitive programming problem generate Python3 code to solve the problem.

```

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved:
{Our Problem Description will be added here}
## Planning: {Planning from the Planning Agent}
## Sample Input/Outputs: {Sample I/Os}
## Let's think step by step.
-----
Important:
## Your response must contain only the {language} code to solve this problem. Do not add extra explanation or words.
```

Debugging Agent

Given a competitive programming problem you have generated {language} code to solve the problem. But the generated code cannot pass sample test cases. Improve your code to solve the problem correctly.

```

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving this problem.
-----
Important:
## Your response must contain the modified planning and then the {language} code inside ``` block to solve this problem.
```

Figure 10: Prompt for Coding and Debugging Agent.

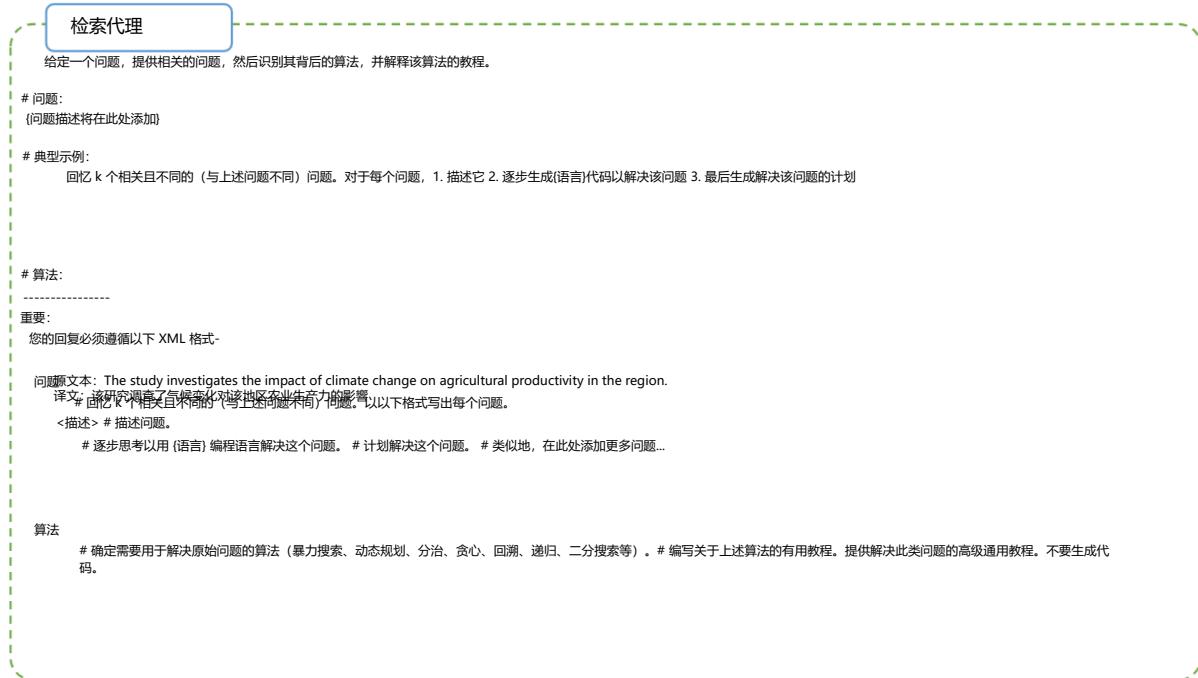


图 8：自我检索代理提示

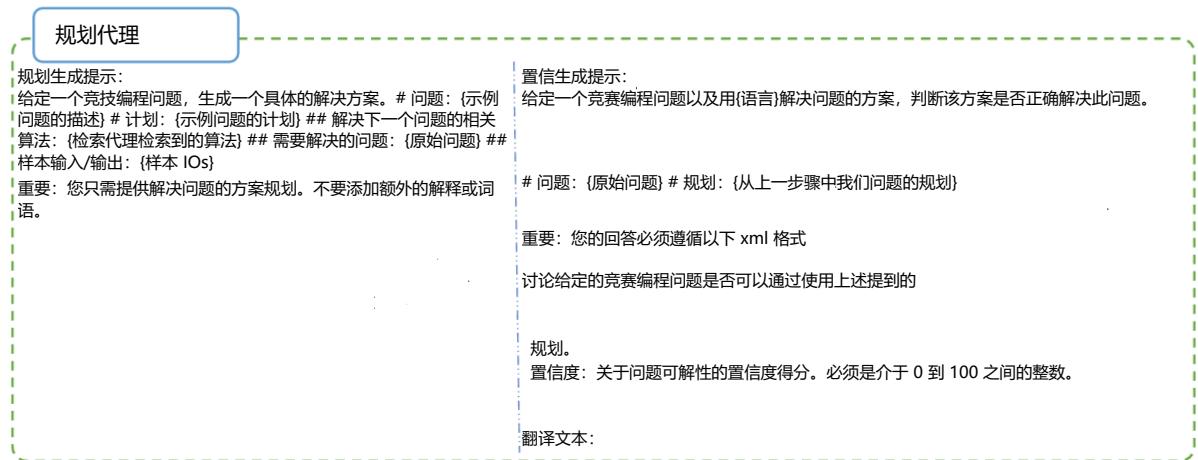


图 9：规划代理的提示。图中提到的示例问题将来自检索代理。

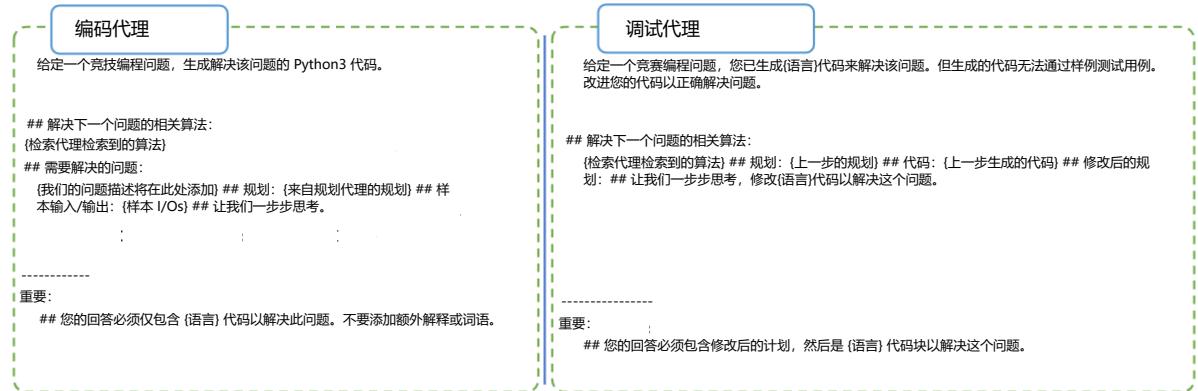


图 10：编码和调试代理提示

C Example Problem

Two complete examples of how MapCoder works by showing all the prompts and responses for all four agents is given below:

C.1 An example containing problem from HumanEval Dataset (k=5, t=5)

Prompt for Self-retrieval Agent:

Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

We will refer this as **original problem** from the next time

Problem:

```
def sum_squares(lst):
    """You are given a list of numbers.
    You need to return the sum of squared numbers in the given list,
    round each element in the list to the upper int(Ceiling) first.
    Examples:
        For lst = [1,2,3] the output should be 14
        For lst = [1,4,9] the output should be 98
        For lst = [1,3,5,7] the output should be 84
        For lst = [1.4,4.2,0] the output should be 29
        For lst = [-2.4,1,1] the output should be 6
    """
```

Exemplars:

Recall five (05) relevant and distinct problems (different from problem mentioned above). For each problem,

1. describe it
2. generate Python3 code step by step to solve that problem
3. finally generate a planning to solve that problem

Algorithm:

Important:

Your response must follow the following xml format-

```
<root>
    <problem>
        # Recall five (05) relevant and distinct problems (different from problem mentioned above). Write each
        # problem in the following format.
        <description> # Describe the problem. </description>
        <code> # Let's think step by step to solve this problem in Python3 programming language. </code>
        <planning> # Planning to solve this problem. </planning>
    </problem>
```

similarly add more problems here...

```
<algorithm>
```

```
    # Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking,
    # Recursive, Binary search, and so on) that needs to be used to solve the original problem.
    # Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for
    # solving this types of problem. Do not generate code.
```

```
</algorithm>
```

```
</root>
```

C 语言示例问题

以下给出了两个完整的 MapCoder 工作示例，展示了所有四个代理的全部提示和响应：

C.1 包含来自 HumanEval 数据集 ($k=5, t=5$) 问题的示例

自我检索代理提示：

给定一个问题，提供相关的问题，然后识别其背后的算法，并解释该算法的教程。

我们将从下次开始称之为原始问题

问题：

```
def sum_squares(lst):
```

 你被给出一个数字列表。

 你需要返回给定列表中平方数的总和，首先将列表中的每个元素向上取整到整数（向上取整）。示例：对于 $lst = [1,2,3]$ ，输出应为 14；对于 $lst = [1,4,9]$ ，输出应为 98。

 对于 $lst = [1,3,5,7]$ ，输出应为 84

 对于 $lst = [1.4,4.2,0]$ ，输出应为 29

 对于 $lst = [-2.4,1,1]$ ，输出应为 6

典型示例：

回忆五个 (05) 相关且独特的问题（与上述问题不同）。对于每个问题，1. 描述它 2. 逐步生成 Python3 代码以解决问题 3. 最后制定解决问题的计划

算法：

重要：

您的回复必须遵循以下 XML 格式-

原文问题 The study investigates the impact of climate change on agricultural productivity in the context of global food security.
回忆五个 (05) 相关且独特的问题（与上述问题不同）。以以下格式写出每个问题。
译文：该研究探讨了气候变化对全球粮食安全背景下农业生产力的影响

<描述> # 描述问题。

逐步思考以使用 Python3 编程语言解决这个问题。 # 计划解决这个问题。

同样，在此处添加更多问题...

算法

确定需要用于解决原始问题的算法（暴力搜索、动态规划、分治、贪心、回溯、递归、二分搜索等）。# 编写关于上述算法的有用教程。提供解决此类问题的高级通用教程。不要生成代码。

Response from Self-retrieval Agent:

```
<root>
<problem>
<description> 1. Problem: Find the sum of all even numbers in a list. </description>
<code>
def sum_even(lst):
    total = 0
    for num in lst:
        if num % 2 == 0:
            total += num
    return total
</code>
<planning>
    To solve this problem, we can iterate through the list and check if each number is even. If it is, we add it
    to a running total. Finally, we return the total sum of all even numbers in the list.
</planning>
</problem>

<problem>
<description> 2. Problem: Find the product of all numbers in a list. </description>
<code>
def product_of_numbers(lst):
    product = 1
    for num in lst:
        product *= num
    return product
</code>
<planning>
    To solve this problem, we can iterate through the list and multiply each number with a running product.
    Finally, we return the total product of all numbers in the list.
</planning>
</problem>
```

We will refer this as **Retrieved algorithm**
from the next time

Three more plans are generated in similar way...

```
<algorithm>
    The algorithm behind the original problem of summing squared numbers in a list is a simple iteration
    through the list and performing a calculation on each element. This can be solved using a brute-force
    algorithm.
    Tutorial:
    Brute-force algorithm involves iterating through the input and checking all possible solutions to find the
    best one. In the case of summing squared numbers in a list, we iterate through the list, square each
    number, round it to the upper integer, and then sum them up. This algorithm is straightforward and easy to
    implement, but it may not be the most efficient for large inputs.
</algorithm>
</root>
```

响应来自自检索代理：

源文本：The study investigates the impact of climate change on agricultural productivity in the region.
译文：该研究调查了气候变化对该地区农业生产力的影响。

```
<code>
Source Text: The study investigates the impact of climate change on agricultural productivity in the region.
Translated Text: 该研究调查了气候变化对该地区农业生产力的影响</code>
```

对于 lst 中的 num：
如果 $num \% 2 == 0$:
 总 += 数
 返回总计
</code>

规划

为了解决这个问题，我们可以遍历列表并检查每个数字是否为偶数。如果是，我们将其添加到累计总和中。最后，我们返回列表中所有偶数的总和。

问题

2. 问题：求一个列表中所有数字的乘积。

```
<code>
Source Text: The study investigates the impact of climate change on agricultural productivity in the region.
Translated Text: 该研究调查了气候变化对该地区农业生产力的影响</code>
```

``` 对于 lst 中的 num:  
    产品 \*= 数  
    返回产品  
</code>

规划

为了解决这个问题，我们可以遍历列表并将每个数字与一个运行乘积相乘。

最后，我们返回列表中所有数字的总乘积。

我们将从下次开始称之为检索算法

三种更多计划以类似方式生成.....

算法

该算法背后的原始问题是求一个数列中平方数的和，它通过遍历数列并对每个元素进行计算来实现。这可以通过暴力算法来解决。教程：暴力算法涉及遍历输入并检查所有可能的解决方案以找到最佳方案。在求一个数列中平方数的和的情况下，我们遍历数列，对每个数进行平方，然后向上取整，最后将它们相加。此算法简单易实现，但对于大量输入可能不是最高效的。

翻译文本：

### Prompt for Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

**# Problem:** 1. Problem: Find the sum of all even numbers in a list.

**# Planning:** To solve this problem, we can iterate through the list and check if each number is even. If it is, we add it to a running total. Finally, we return the total sum of all even numbers in the list.

**## Relevant Algorithm to solve the next problem:** {Retrieved algorithm}

**## Problem to be solved:** {Original problem}

**## Sample Test cases:**

```
assert sum_squares([1,2,3])==14
assert sum_squares([1,4,9])==98
assert sum_squares([1,3,5,7])==84
assert sum_squares([1.4,4.2,0])==29
assert sum_squares([-2.4,1,1])==-6
```

We will refer this as **Sample IO** from the next time

**## Planning:**

**Important:** You should give only the planning to solve the problem. Do not add extra explanation or words.

### Response from Planning Agent (Plan Generation):

To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

---

### Prompt for Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

**# Problem:** {Original problem}

**# Planning:**

To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

**Important:** Your response must follow the following xml format-

<root>

  <explanation>

    Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.

  </explanation>

  <confidence>

    Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.

  </confidence>

</root>

## 规划代理提示（计划生成）：

给定一个竞技编程问题，生成一个具体的解决方案。

问题：1. 问题：求一个列表中所有偶数的和。

# 规划：为了解决这个问题，我们可以遍历列表并检查每个数字是否为偶数。如果是，我们将其添加到累计总和中。最后，我们返回列表中所有偶数的总和。

## 解决下一个问题的相关算法：{检索算法} ## 需要解决的问题：{原始问题} ## 样本测试用例：

```
断言 sum_squares([1,2,3]) == 14 断言
sum_squares([1,4,9]) == 98 断言
sum_squares([1,3,5,7]) == 84 断言
sum_squares([1.4,4.2,0]) == 29 断言
sum_squares([-2.4,1,1]) == 6
```

我们将从下次开始称之为样本  
IO

## 规划：

重要：您只需提供解决问题的方案规划。不要添加额外的解释或词语。

## 响应来自规划代理（计划生成）：

为了解决这个问题，我们可以遍历列表，将每个数字向上取整，然后平方，最后将其加到累计总和中。最后，我们返回列表中所有平方数的总和。

---

## 提示规划代理（置信度生成）：

给定一个竞赛编程问题以及一个用 Python3 解决问题的方案，判断该方案是否正确解决此问题。

# 问题：{原始问题}

# 规划：

为了解决这个问题，我们可以遍历列表，将每个数字向上取整，然后平方，最后将其加到累计总和中。最后，我们返回列表中所有平方数的总和。

重要：您的回答必须遵循以下 XML 格式

```
<explanation>
```

讨论上述提到的规划是否可以解决给定的竞赛编程问题。

置信度

置信度关于问题可解性的评分。必须是 0 到 100 之间的整数。

### Response from Planning Agent (Confidence Generation):

```
<root>
<explanation>
The given competitive programming problem is solvable by using the above mentioned planning. The plan correctly outlines the steps to iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, returning the total sum of all squared numbers in the list.
</explanation>
<confidence> 100 </confidence>
</root>
```

*Similarly generate planning and their confidence using other examples....*

**Decision:** As First Planning gets 100 as confidence score Coding Agent will work with first Planning.

---

### Prompt for Coding Agent:

Given a competitive programming problem generate Python3 code to solve the problem.

**## Relevant Algorithm to solve the next problem:** {Retrieved algorithm}

**## Problem to be solved:** {Original problem}

**## Planning:**

To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

**## Sample Test cases:** {Sample I/O}

**## Let's think step by step.**

---

**Important:**

## Your response must contain only the Python3 code to solve this problem. Do not add extra explanation or words.

### Response from Coding Agent:

```
def sum_squares(lst):
 total = 0
 for num in lst:
 total += (int(num) ** 2)
 return total
```

**Decision:** Testing using Sample I/O will fail because this is wrong implementation. So, our approach will go to Debugging Agent.

---

响应来自规划代理（置信度生成）：

源文本: The study investigates the impact of climate change on agricultural productivity in the region.  
译文：该研究调查了气候变化对该地区农业生产的影响。该规划正确地概述了遍历列表、将每个数字向上取整、平方，并将其加到运行总和中的一系列步骤。最后，返回列表中所有平方数的总和。

<置信度> 100

同样，使用其他例子生成计划和它们的置信度.....

**决策：当第一规划获得 100 的置信度分数时，编码代理将与第一规划一起工作。**

---

编码代理提示：

给定一个竞技编程问题，生成解决该问题的 Python3 代码。

## 解决下一个问题的相关算法：{检索到的算法}

## 需要解决的问题：{原始问题}

## 规划：

为了解决这个问题，我们可以遍历列表，将每个数字向上取整，然后平方，最后将其加到累计总和中。最后，我们返回列表中所有平方数的总和。

## 样本测试用例：{样本输入/输出}

## 一步一步来思考。

-----  
重要：

## 您的回答必须仅包含解决此问题的 Python3 代码。不要添加额外解释或文字。

响应来自编码代理：

```
def sum_squares(lst):
 总 = 0
 对于 lst 中的 num:
 total += (int(num) ** 2)
 return total
 总 += (int(num) ** 2)
 返回 总
```

**决策：使用样本 I/O 进行测试将失败，因为这属于错误实现。因此，我们的方法将转向调试代理。**

---

### **Prompt for Debugging Agent: (1<sup>st</sup> iteration)**

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

```
Relevant Algorithm to solve the next problem: {Retrieved algorithm}
Problem to be solved: {Original problem}
Planning: {Current Planning}
Code: {Current Implementation}
Test Report:
passed in test case: assert sum_squares([1,2,3]) == 14
passed in test case: assert sum_squares([1,4,9]) == 98
passed in test case: assert sum_squares([1,3,5,7]) == 84
failed in test case: assert sum_squares([1.4,4.2,0]) == 29
passed in test case: assert sum_squares([-2.4,1,1]) == 6
Modified Planning:
Let's think step by step to modify Python3 Code for solving this problem.
```

---

#### **Important:**

## Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

### **Response from Debugging Agent: (1<sup>st</sup> iteration)**

#### **## Modified Planning:**

To modify the code, we need to round each number to the upper integer before squaring it. We can use the math.ceil() function to achieve this. After rounding each number, we can then square it and add it to the total sum.

```
Code:
```Python3
import math

def sum_squares(lst):
    total = 0
    for num in lst:
        total += (math.ceil(num) ** 2)
    return total
```

```

*Decision: This time all Sample I/O passed. We return this implementation as the Final Code for final evaluation. Finally, we evaluated this code with private test case and our implementation passed the evaluation.*

调试代理提示：（1 次迭代）

给定一个竞赛编程问题，你已经生成了用于解决问题的 Python3 代码。但是生成的代码无法通过样例测试用例。改进你的代码以正确解决问题。

```
解决下一个问题的相关算法: {检索到的算法} ## 需要解决的问题: {原
始问题} ## 计划: {当前计划} ## 代码: {当前实现} ## 测试报告: 在测试用
例 assert sum_squares([1,2,3]) == 14 中通过, 在测试用例 assert
sum_squares([1,4,9]) == 98 中通过
```

通过测试用例: assert sum\_squares([1,3,5,7]) == 84 失  
败在测试用例: assert sum\_squares([1.4,4.2,0]) == 29 通  
过测试用例: assert sum\_squares([-2.4,1,1]) == 6

## 修改规划: ## 让我们一步步思考修改 Python3 代码以解决这个问题。

-----  
**重要:**

## 您的回答必须包含修改后的计划和随后在 `` 块内的 Python3 代码以解决这个问题。

响应来自调试代理：（1 次迭代）

## 修改后的规划:

在修改代码时，我们需要将每个数字向上取整后再进行平方。我们可以使用 math.ceil() 函数来实现这一点。在将每个数字取整后，我们可以将其平方并加到总和中。

```
代码:
```Python3
# 导入 math 模块
from math import ceil
```

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += ceil(num) ** 2
    return total
```

决策：这次所有样本输入/输出均通过。我们将此实现作为最终代码提交进行最终评估。最后，我们使用私有测试案例评估了此代码，我们的实现通过了评估。

C.2 An example containing problem from CodeContest Dataset (k=3, t=5)

We will refer this as **original problem** from the next time

Prompt for Self-retrieval Agent:

Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

Problem:

Luntik has decided to try singing. He has a one-minute songs, b two-minute songs and c three-minute songs. He wants to distribute all songs into two concerts such that every song should be included to exactly one concert.

He wants to make the absolute difference of durations of the concerts as small as possible. The duration of the concert is the sum of durations of all songs in that concert.

Please help Luntik and find the minimal possible difference in minutes between the concerts durations.

Input

The first line contains a single integer t ($1 \leq t \leq 1000$) – the number of test cases.

Each test case consists of one line containing three integers a, b, c ($1 \leq a, b, c \leq 10^9$) – the number of one-minute, two-minute and three-minute songs.

Output

For each test case print the minimal possible difference in minutes between the concerts durations.

Example

Input

4

1 1 1

2 1 3

5 5 5

1 1 2

Output

0

1

0

1

Note

In the first test case, Luntik can include a one-minute song and a two-minute song into the first concert, and a three-minute song into the second concert. Then the difference will be equal to 0.

In the second test case, Luntik can include two one-minute songs and a two-minute song and a three-minute song into the first concert, and two three-minute songs into the second concert. The duration of the first concert will be $1 + 1 + 2 + 3 = 7$, the duration of the second concert will be 6. The difference of them is $|7 - 6| = 1$.

Important Note: You must follow the input output format. Input must be taken from standard input and output must be given to standard output. The code will be tested against multiple test cases and all the test cases must be passed.

C.2 一个包含来自 CodeContest 数据集 ($k=3$, $t=5$) 问题的示例

我们将从下次开始称之为原始问题

自我检索代理提示：

给定一个问题，提供相关的问题，然后识别其背后的算法，并解释该算法的教程。

问题：

Luntik 决定尝试唱歌。他有一分钟的歌曲， b 两分钟的歌曲， c 三分钟的歌曲。他想将所有歌曲分配到两场音乐会中，使得每首歌曲恰好被包含在一场音乐会中。他希望使两场音乐会持续时间的绝对差值尽可能小。音乐会的持续时间是那场音乐会中所有歌曲持续时间的总和。请帮助 Luntik 找到音乐会持续时间之间可能的最小差值（以分钟为单位）。

输入：第一行包含一个整数 t ($1 \leq t \leq 1000$) —— 测试用例的数量。每个测试用例由一行包含三个整数 a 、 b 、 c ($1 \leq a, b, c \leq 10^9$) 组成——一分钟、两分钟和三分钟歌曲的数量。输出：对于每个测试用例，打印音乐会持续时间之间的最小可能分钟差。

示例

输入

```
4
1 1 1
2 1 3
5 5 5
1 1 2
输出
0
1
0
1
```

注意：在第一个测试案例中，Luntik 可以将一首一分钟的歌曲和一首两分钟的歌曲纳入第一场音乐会，以及一首三分钟的歌曲纳入第二场音乐会。那么差异将等于 0。在第二个测试案例中，Luntik 可以将两首一分钟的歌曲、一首两分钟的歌曲和一首三分钟的歌曲纳入第一场音乐会，以及两首三分钟的歌曲纳入第二场音乐会。第一场音乐会的时长将是 $1 + 1 + 2 + 3 = 7$ ，第二场音乐会的时长将是 6。它们的差异是 $|7-6| = 1$ 。

-----重要提示：您必须遵循输入输出格式。输入必须来自标准输入，输出必须输出到标准输出。代码将针对多个测试用例进行测试，并且所有测试用例都必须通过。

Exemplars:

Recall three (03) relevant and distinct problems (different from problem mentioned above). For each problem,

1. describe it
2. generate Python3 code step by step to solve that problem
3. finally generate a planning to solve that problem

Algorithm:**Important:**

Your response must follow the following xml format-

```
<root>
  <problem>
    # Recall three (03) relevant and distinct problems (different from problem mentioned above). Write
    each problem in the following format.
    <description> # Describe the problem. </description>
    <code> # Let's think step by step to solve this problem in Python3 programming language. </code>
    <planning> # Planning to solve this problem. </planning>
  </problem>

  # similarly add more problems here...

  <algorithm>
    # Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking,
    Recursive, Binary search, and so on) that needs to be used to solve the original problem.
    # Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for
    solving this types of problem. Do not generate code.
  </algorithm>
</root>
```

Response from Self-retrieval Agent:

```
<root>
  <problem>
    <description>
      Problem 1:
      You are given an array of integers, find the maximum sum of a contiguous subarray within the array.
    </description>
    <code>
      def maxSubArray(nums):
        max_sum = nums[0]
        current_sum = nums[0]

        for i in range(1, len(nums)):
          current_sum = max(nums[i], current_sum + nums[i])
          max_sum = max(max_sum, current_sum)

      return max_sum
    </code>
    <planning>
      - Initialize two variables max_sum and current_sum to the first element of the array.
    </planning>
  </problem>
</root>
```

典型示例:

回忆三个 (03) 相关且独特的问题 (与上述问题不同)。对于每个问题, 1. 描述它 2. 逐步生成 Python3 代码以解决问题 3. 最后制定解决问题的计划

算法:

重要:

您的回复必须遵循以下 XML 格式-

源文**问题** The study investigates the impact of climate change on agricultural productivity in the context of global food

security # 回忆三个 (03) 相关且不同的 (与上述问题不同) 问题。以下格式书写每个问题。

译文: 该研究探讨了气候变化对全球粮食安全背景下农业生产力的影响

<描述> # 描述问题。

逐步思考以使用 Python3 编程语言解决这个问题。# 计划解决这个问题。

同样, 在此处添加更多问题...

算法

确定需要用于解决原始问题的算法 (暴力搜索、动态规划、分治、贪心、回溯、递归、二分搜索等)。# 编写关于上述算法的有用教程。提供解决此类问题的高级通用教程。不要生成代码。

响应来自自检索代理:

源**题本**: The study investigates the impact of climate change on agricultural productivity in the context of global food

security **description**>

译文: 该研究探讨了气候变化对全球粮食安全背景下农业生产力的影响

问题 1: 给定一个整数数组, 找出数组中连续子数组的最大和。<code></code>

```
def maxSubArray(nums):
    max_sum = nums[0]
    current_sum = nums[0]
    最大和 = nums[0] 当前和 =
    nums[0]
    for i in range(1, len(nums)):
        for i 在前到 len(nums) 的范围内: 前和 + nums[i]) 最大和 = max(最大和, 当
        前和)
```

返回最大和

</code>

规划

- 将两个变量 max_sum 和 current_sum 初始化为数组的第一个元素。

```

        - Iterate through the array and at each index, update the current_sum to be the maximum of the
        current element and the sum of the current element and the previous current_sum.
        - Update max_sum to be the maximum of max_sum and current_sum.
        - Return max_sum at the end.

    </planning>
</problem>

<problem>
    <description>
        Problem 2:
        You are given a list of tasks with their start and end times. Find the minimum number of meeting rooms
        required to schedule all the tasks.
    </description>
    <code>
        import heapq

        def minMeetingRooms(intervals):
            if not intervals:
                return 0

            intervals.sort(key=lambda x: x[0])

            rooms = []
            heapq.heappush(rooms, intervals[0][1])

            for i in intervals[1:]:
                if rooms[0] <= i[0]:
                    heapq.heappop(rooms)
                heapq.heappush(rooms, i[1])

            return len(rooms)
    </code>
    <planning>
        - Sort the intervals based on their start times.
        - Create a min heap to store the end times of the meetings.
        - Push the end time of the first meeting into the heap.
        - Iterate through the intervals, if the start time of the current interval is greater than or equal to the end
        time at the top of the heap, pop the end time from the heap. Push the end time of the current interval
        into the heap.
        - Return the length of the heap at the end.
    </planning>
</problem>

```

```

<problem>
    <description>
        Problem 3:
        You are given a list of intervals representing the start and end times of meetings. Find the maximum
        number of overlapping intervals.
    </description>
    <code>

```

- 遍历数组，在每一个索引处，将 current_sum 更新为当前元素与当前元素和前一个 current_sum 之和的最大值。
- 更新 max_sum 为 max_sum 和 current_sum 中的最大值。
- 最后返回 max_sum。

问题

<description>

问题 2：给定一个包含任务开始和结束时间的列表。找出安排所有任务所需的最小会议室数量。<code></code>

导入 heapq

```
def minMeetingRooms(intervals):
    if len(intervals) == 0:
        return 0

    intervals.sort(key=lambda x: x[0]) # 将 intervals 列表按照 key 指定的第一个元素进行排序

    rooms = [] heapq.heappush(rooms, intervals[0])
    room_count = 1
    for i in intervals[1:]:
        if rooms[0][0] <= i[0]:
            heapq.heappop(rooms)
            heapq.heappush(rooms, i)
        else:
            heapq.heappush(rooms, i)
    return room_count
</code>
```

规划

- 根据区间的起始时间进行排序。
- 创建一个最小堆以存储会议的结束时间。
- 将第一次会议的结束时间推入堆中。
- 遍历区间，如果当前区间的开始时间大于或等于堆顶的结束时间，则从堆中弹出结束时间。将当前区间的结束时间推入堆中。

- 返回堆的长度。

问题

<description>

问题 3：给定一个表示会议开始和结束时间的区间列表。找出最大重叠区间的数量。<code></code>

```

def maxOverlap(intervals):
    if not intervals:
        return 0

    start_times = sorted([i[0] for i in intervals])
    end_times = sorted([i[1] for i in intervals])

    max_overlap = 0
    current_overlap = 0
    start_ptr = 0
    end_ptr = 0

    while start_ptr < len(intervals):
        if start_times[start_ptr] < end_times[end_ptr]:
            current_overlap += 1
            max_overlap = max(max_overlap, current_overlap)
            start_ptr += 1
        else:
            current_overlap -= 1
            end_ptr += 1

    return max_overlap

```

</code>

<planning>

- Sort the start times and end times of the intervals separately.
- Initialize max_overlap and current_overlap to 0, and start and end pointers to 0.
- Iterate through the start times, if the current start time is less than the current end time, increment current_overlap and update max_overlap. Increment the start pointer.
- If the current start time is greater than or equal to the current end time, decrement current_overlap and increment the end pointer.
- Return max_overlap at the end.

</planning>

</problem>

<algorithm>

The algorithm that can be used to solve the original problem of distributing songs into two concerts with minimal possible difference in durations is the Greedy algorithm. The Greedy algorithm makes the locally optimal choice at each stage with the hope of finding a global optimum.

To solve the original problem, we can use the following steps:

1. Sort the songs by their durations in non-increasing order.
2. Start with two empty concerts.
3. Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
4. Calculate the absolute difference in durations between the two concerts at the end.

This approach ensures that we are making the locally optimal choice at each stage by assigning the current song to the concert with the smaller total duration, and it leads to the minimal possible difference in durations between the concerts.

</ algorithm>

</root>

We will refer this as **Retrieved algorithm** from the next time

```

def maxOverlap(intervals):
    如果不为区间:
        返回 0

    start_times = sorted([i[0] for i in intervals]) end_times =
    sorted([i[1] for i in intervals])
    开始时间 = 对[intervals][0]进行排序的列表[] end 时间 = 对
    最大重叠们[]进行排序的列表[]
    当前重叠 = 0
    start_ptr = 0
    end_ptr = 0
    Translated Text: end_ptr = 0
    当 start_ptr < len(intervals):
        如果 start_times[start_ptr] < end_times[end_ptr]:
            当前重叠 += 1
            最大重叠 = max(最大重叠, 当前重叠)
            起始指针 += 1
        else:
            当前重叠 -= 1 结束指针 +=
            1

    返回最大重叠
</code>

```

规划

- 分别对区间的起始时间和结束时间进行排序。
- 初始化 `max_overlap` 和 `current_overlap` 为 0，并将起始和结束指针设置为 0。 - 遍历起始时间，如果当前起始时间小于当前结束时间，则增加 `current_overlap` 并更新 `max_overlap`。增加起始指针。 - 如果当前起始时间大于或等于当前结束时间，则减少 `current_overlap` 并增加结束指针。

- 返回最后的最大重叠。

算法

我们将从下次开始称之为检索算法

该算法可用于解决将歌曲分配到两场音乐会中，使持续时间差异尽可能小的原始问题，即为贪心算法。贪心算法在每个阶段都做出局部最优的选择，以期找到全局最优解。

为了解决原始问题，我们可以采取以下步骤：1. 按歌曲时长降序排列歌曲。

2. 从两个空音乐会开始。
3. 对排序后的歌曲进行迭代，在每一阶段，将当前歌曲分配给总时长较短的演唱会。
4. 计算两个音乐会持续时间之间的绝对差值。

这种方法确保我们在每个阶段都做出局部最优的选择，通过将当前歌曲分配给总时长较短的演唱会，从而使得演唱会之间的时长差异尽可能小。

翻译文本：

Prompt for First Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

Problem:

Problem 1:

You are given an array of integers, find the maximum sum of a contiguous subarray within the array.

Planning:

- Initialize two variables max_sum and current_sum to the first element of the array.
- Iterate through the array and at each index, update the current_sum to be the maximum of the current element and the sum of the current element and the previous current_sum.
- Update max_sum to be the maximum of max_sum and current_sum.
- Return max_sum at the end.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}

Problem to be solved: {Original problem}

Sample Test cases:

Input:

```
4  
1 1 1  
2 1 3  
5 5 5  
1 1 2
```

Expected output:

```
0 1 0 1
```

We will refer this as **Sample IO**
from the next time

Planning:

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

Response from First Planning Agent (Plan Generation):

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Prompt for First Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

Problem: {Original problem}

Planning:

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.

提示：首次规划代理（计划生成）：

给定一个竞技编程问题，生成一个具体的解决方案。

问题：

问题 1：给定一个整数数组，找出数组中连续子数组的最大和。

规划：

- 将两个变量 `max_sum` 和 `current_sum` 初始化为数组的第一个元素。 - 遍历数组，在每一个索引处，将 `current_sum` 更新为当前元素与当前元素和前一个 `current_sum` 之和中的最大值。

- 更新 `max_sum` 为 `max_sum` 和 `current_sum` 中的最大值。

- 最后返回 `max_sum`。

解决下一个问题的相关算法：{检索算法} ## 需要解决的问题：{原始问题} ## 样本测试用例：

Input:
输入：
1 1 1
Translated Text:
输入5
1 1 2
Expected output:
预期输出：

我们将从下次开始称之为样本
IO

规划：

重要：您只需提供解决问题的方案规划。不要添加额外的解释或词语。

响应来自第一个规划代理（计划生成）：

- 按歌曲时长降序排列。
- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。

计算两次音乐会持续时间之间的绝对差异。

提示：首次规划代理（置信度生成）：

给定一个竞赛编程问题以及一个用 Python3 解决问题的方案，判断该方案是否正确解决此问题。

问题：{原始问题}

规划：

- 按歌曲时长降序排列。
- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。

- Calculate the absolute difference in durations between the two concerts at the end.
-

Important: Your response must follow the following xml format-

```
<root>
  <explanation>
    Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.
  </explanation>
  <confidence>
    Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
  </confidence>
</root>
```

Response from First Planning Agent (Confidence Generation):

```
<root>
  <explanation>
    The given competitive programming problem is solvable by using the above mentioned planning. The plan involves sorting the songs by duration and then assigning them to two concerts in a way that minimizes the absolute difference in durations between the concerts.
  </explanation>
  <confidence> 100 </confidence>
</root>
```

Prompt for Second Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

Problem:

Problem 2:

You are given a list of tasks with their start and end times. Find the minimum number of meeting rooms required to schedule all the tasks.

Planning:

- Sort the intervals based on their start times.
- Create a min heap to store the end times of the meetings.
- Push the end time of the first meeting into the heap.
- Iterate through the intervals, if the start time of the current interval is greater than or equal to the end time at the top of the heap, pop the end time from the heap. Push the end time of the current interval into the heap.
- Return the length of the heap at the end.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}

Problem to be solved: {Original problem}

Sample Test cases: {Sample I/O}

Planning:

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

计算两次音乐会持续时间之间的绝对差异。

重要：您的回答必须遵循以下 XML 格式

```
<explanation>
```

讨论上述提到的规划是否可以解决给定的竞赛编程问题。

置信度

置信度关于问题可解性的评分。必须是 0 到 100 之间的整数。

响应来自第一个规划代理（置信度生成）：

源文本 ~~The study~~ investigates the impact of climate change on agricultural productivity in the context of global food security. The study finds that climate change will have a significant impact on agricultural productivity, particularly in developing countries. The study also finds that adaptation measures can help mitigate the impact of climate change on agriculture.

提示：第二规划代理（计划生成）：

给定一个竞技编程问题，生成一个具体的解决方案。

问题：

问题 2：给定一个包含任务开始和结束时间的列表。找出安排所有任务所需的最小会议室数量。

规划：

- 根据区间的起始时间进行排序。
- 创建一个最小堆以存储会议的结束时间。

将第一次会议的结束时间推入堆中。

- 遍历区间，如果当前区间的开始时间大于或等于堆顶的结束时间，则从堆中弹出结束时间。将当前区间的结束时间推入堆中。

- 返回堆的长度。

```
## 解决下一个问题的相关算法: {检索到的算法} ## 需要解决的问题: {原  
始问题} ## 样本测试用例: {样本输入输出} ## 计划:
```

重要：您只需提供解决问题的方案规划。不要添加额外的解释或词语。

Response from Second Planning Agent (Plan Generation):

- Sort the songs by their durations in non-increasing order.
 - Start with two empty concerts.
 - Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
 - Calculate the absolute difference in durations between the two concerts at the end.
-

Prompt for Second Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

Problem: {Original problem}

Planning:

- Sort the songs by their durations in non-increasing order.
 - Start with two empty concerts.
 - Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
 - Calculate the absolute difference in durations between the two concerts at the end.
-

Important: Your response must follow the following xml format-

```
<root>
  <explanation>
    Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.
  </explanation>
  <confidence>
    Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
  </confidence>
</root>
```

Response from Second Planning Agent (Confidence Generation):

```
<root>
  <explanation>
    The given competitive programming problem is solvable by using the above mentioned planning. The plan involves sorting the songs by duration, then assigning the songs to two concerts in a way that minimizes the absolute difference in durations between the concerts.
  </explanation>
  <confidence> 100 </confidence>
</root>
```

Similarly generate the planning and the confidence of last example....

Decision: As First Planning gets 100 as confidence score Coding Agent will work with first Planning.

回复来自第二个规划代理（计划生成）：

- 按歌曲时长降序排列。
- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。

计算两次音乐会持续时间之间的绝对差异。

提示：第二规划代理（置信度生成）：

给定一个竞赛编程问题以及一个用 Python3 解决问题的方案，判断该方案是否正确解决此问题。

```
# 问题: {原始问题}
# 规划:
- 按歌曲时长降序排列。
- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。
```

计算两次音乐会持续时间之间的绝对差异。

重要：您的回答必须遵循以下 XML 格式

```
<explanation>
    讨论上述提到的规划是否可以解决给定的竞赛编程问题。
```

置信度

置信度关于问题可解性的评分。必须是 0 到 100 之间的整数。

回复来自第二个规划代理（置信度生成）：

原文本 The study investigates the impact of climate change on agricultural productivity in the context of global food
所给的竞赛编程问题可以通过上述提到的规划来解决。该规划包括按时长对歌曲进行排序，然后以最小化两
场比赛音乐会对气候变化对全球粮食安全背景下农业生产力的影响
100

同样生成上一个例子中的计划和信心.....

决策：当第一规划获得 100 的置信度分数时，编码代理将与第一规划一起工作。

Prompt for Coding Agent:

Given a competitive programming problem generate Python3 code to solve the problem.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}

Problem to be solved: {Original problem}

Planning:

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Sample Test cases: {Sample I/O}

Let's think step by step.

Important:

Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will fail the test cases.

Your response must contain only the Python3 code to solve this problem. Do not add extra explanation or words.

Response from Coding Agent:

```
def min_difference_in_concerts(t, test_cases):  
    for case in test_cases:  
        a, b, c = case  
        songs = [(1, a), (2, b), (3, c)]  
        songs.sort(key=lambda x: x[0], reverse=True)  
        concert1 = 0  
        concert2 = 0  
        for song in songs:  
            if concert1 <= concert2:  
                concert1 += song[0] * song[1]  
            else:  
                concert2 += song[0] * song[1]  
    print(abs(concert1 - concert2))  
  
t = 4  
test_cases = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]  
min_difference_in_concerts(t, test_cases)
```

Decision: Testing using Sample I/O will fail because of incorrect implementation. The main problem with this code is that it does not take input from the console. It writes a function and just calls it with fixed value. So, our approach will go to the Debugging Agent.

编码代理提示：

给定一个竞技编程问题，生成解决该问题的 Python3 代码。

解决下一个问题的相关算法：{检索到的算法} ## 需要解决的问题：{原始问题} ## 计划： - 按歌曲时长降序排列歌曲。

- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。

计算两次音乐会持续时间之间的绝对差异。

样本测试用例：{样本输入/输出}
让我们一步步思考。

重要：

注意：严格遵循输入和输出格式。输入应从标准输入获取，输出应输出到标准输出。如果你正在编写函数，则在函数定义之后，使用`input()`函数获取输入，然后使用指定参数调用函数，最后打印函数的输出。不要添加额外的打印语句，否则将无法通过测试用例。## 你的回答必须只包含解决此问题的 Python3 代码，不要添加额外的解释或文字。

响应来自编码代理：

```
def min_difference_in_concerts(t, test_cases):
```

对于测试用例中的每个案例：

```
a, b, c = case  
songs = [(1, a), (2, b), (3, c)]  
songs.sort(key=lambda x: x[0], reverse=True)  
concert1 = 0  
concert2 = 0
```

对于歌曲中的每首歌曲：

```
    如果音乐会 1 <= 音乐会 2:  
        concert1 += song[0] * song[1] elif:  
  
            concert2 += song[0] * song[1]  
print(abs(concert1 - concert2))
```

```
t = 4 测试用例 = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]  
min_difference_in_concerts(t, 测试用例)
```

决策：使用样本 I/O 进行测试将因实现错误而失败。此代码的主要问题是它没有从控制台获取输入。它定义了一个函数，只是用固定值调用它。因此，我们的方法将转向调试代理。

Prompt for Debugging Agent: (1st iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}
Problem to be solved: {Original problem}

Planning:

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Code:

```
def min_difference_in_concerts(t, test_cases):  
    for case in test_cases:  
        a, b, c = case  
        songs = [(1, a), (2, b), (3, c)]  
        songs.sort(key=lambda x: x[0], reverse=True)  
        concert1 = 0  
        concert2 = 0  
        for song in songs:  
            if concert1 <= concert2:  
                concert1 += song[0] * song[1]  
            else:  
                concert2 += song[0] * song[1]  
    print(abs(concert1 - concert2))  
  
t = 4  
test_cases = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]  
min_difference_in_concerts(t, test_cases)
```

Test Report:

Tested passed:

No test cases passed.

Tests failed:

Input:

4

1 1 1

2 1 3

5 5 5

1 1 2

Expected Output:

0 1 0 1

Your Output:

0 5 0 3

Modified Planning:

调试代理提示： (1 次迭代) —

给定一个竞赛编程问题，你已经生成了用于解决问题的 Python3 代码。但是生成的代码无法通过样例测试用例。改进你的代码以正确解决问题。

```
## 解决下一个问题的相关算法: {检索到的算法} ## 需要解决的问题: {原始问题}
```

规划:

- 按歌曲时长降序排列。
- 从两个空荡荡的音乐会开始。
- 遍历排序后的歌曲，在每个阶段，将当前歌曲分配给总时长较短的演唱会。

计算两次音乐会持续时间之间的绝对差异。

代码:

```
def min_difference_in_concerts(t, test_cases):
```

对于测试用例中的每个案例:

```
        a, b, c = case
        songs = [(1, a), (2, b), (3, c)]
        songs.sort(key=lambda x: x[0], reverse=True)
        concert1 = 0
        concert2 = 0
```

对于歌曲中的每首歌曲:

```
        如果音乐会 1 <= 音乐会 2:
            concert1 += song[0] * song[1]
        elif:
            concert2 += song[0] * song[1]
        print(abs(concert1 - concert2))
```

```
t = 4 测试用例 = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]
min_difference_in_concerts(t, 测试用例)
```

测试报告:

测试未通过: 没有

测试用例通过。

测试失败:

Input:

输入:

```
1 1 1
Translated Text:
輸入:
```

5 5 5

1 1 2

Expected Output:

Translated Text: 预期输出:

Source Text: The study investigates the impact of climate change on agricultural productivity in the region.
Translated Text: 该研究调查了气候变化对该地区农业生产力的影响

修改后的规划:

Let's think step by step to modify Python3 Code for solving this problem.

Important:

Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will fail the test cases.

Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (1st iteration)

Modified Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
 - Read the integers a, b, and c from input.
 - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
 - Calculate the total duration of all songs.
 - Calculate the minimum possible difference in minutes between the concert durations.
 - Print the minimum possible difference in minutes between the concert durations.
4. Call the function min_difference_in_concerts with t and test_cases as input parameters.

Code:

```
'''Python3
def min_difference_in_concerts(t, test_cases):
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            print(total_duration // 2)
        else:
            print((total_duration - 1) // 2)

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

min_difference_in_concerts(t, test_cases)
'''
```

Decision: This time, it takes input from the console and gives output to the console. It correct that part. But again, it failed because of incorrect implementation. So, the Debugging Agent tries to improve the code again.

```
## 逐步思考修改 Python3 代码以解决此问题
```

重要:

注意: 严格遵循输入和输出格式。输入应从标准输入获取, 输出应输出到标准输出。如果你正在编写函数, 则在函数定义之后, 使用`input()`函数获取输入, 然后使用指定参数调用函数, 最后打印函数的输出。不要添加额外的打印语句, 否则将无法通过测试用例。

```
## 您的回答必须包含修改后的计划和随后在```块内的 Python3 代码以解决这个问题。
```

响应来自调试代理: (1 次迭代) _

修改后的规划:

1. 从输入中读取整数 t , 以确定测试用例的数量。
2. 定义一个函数 `min_difference_in_concerts(t , test_cases)`, 该函数接受 t 和 `test_cases` 作为输入参数。
3. 对于 `test_cases` 中的每个测试用例, 执行以下操作:
 - 从输入中读取整数 a 、 b 和 c 。
 - 计算每种类型歌曲的总时长 (1 分钟、2 分钟、3 分钟)。
 - 计算所有歌曲的总时长。
 - 计算音乐会时长之间可能的最小分钟差。
 - 打印音乐会时长之间可能的最小差异 (分钟)。
4. 调用函数 `min_difference_in_concerts`, 输入参数为 t 和 `test_cases`。

代码:

```
'''Python3 def min_difference_in_concerts( $t$ ,  
测试用例):  
    对于测试用例中的每个 case:  $a$ ,  
         $b, c = case$  total_duration =  $a +$   
         $2*b + 3*c$   
        if 总时长 % 2 == 0: print(总时长  
        // 2) else: print((总时长 - 1) // 2)
```

```
t = int(input())  
测试用例列表 = []  
for _ in range(t): a, b, c =  
    map(int, input().split())  
    test_cases.append((a, b, c))  
  
min_difference_in_concerts(t, 测试用例)  
'''
```

决策: 这次它从控制台获取输入并将输出发送到控制台。那部分是正确的。但是, 由于实现错误, 它再次失败了。因此, 调试代理再次尝试改进代码。

Prompt for Debugging Agent: (2nd iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}

Problem to be solved: {Original problem}

Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
 - Read the integers a, b, and c from input.
 - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
 - Calculate the total duration of all songs.
 - Calculate the minimum possible difference in minutes between the concerts durations.
 - Print the minimum possible difference in minutes between the concerts durations.
4. Call the function min_difference_in_concerts with t and test_cases as input parameters.

Code:

```
def min_difference_in_concerts(t, test_cases):
```

```
    for case in test_cases:
```

```
        a, b, c = case
```

```
        total_duration = a + 2*b + 3*c
```

```
        if total_duration % 2 == 0:
```

```
            print(total_duration // 2)
```

```
        else:
```

```
            print((total_duration - 1) // 2)
```

```
t = int(input())
```

```
test_cases = []
```

```
for _ in range(t):
```

```
    a, b, c = map(int, input().split())
```

```
    test_cases.append((a, b, c))
```

```
min_difference_in_concerts(t, test_cases)
```

Test Report:

Tested passed:

No test cases passed.

Tests failed:

Input:

4

1 1 1

2 1 3

5 5 5

1 1 2

Expected Output:

0 1 0 1

Your Output:

3 6 15 4

调试代理提示：(2 次迭代) —

给定一个竞赛编程问题，你已经生成了用于解决问题的 Python3 代码。但是生成的代码无法通过样例测试用例。改进你的代码以正确解决问题。

```
## 解决下一个问题的相关算法: {检索到的算法} ## 需要解决的问题: {原  
始问题} ## 计划:
```

1. 从输入中读取整数 t ，以确定测试用例的数量。
2. 定义一个函数 `min_difference_in_concerts(t, test_cases)`，该函数接受 t 和 `test_cases` 作为输入参数。
3. 对于 `test_cases` 中的每个测试用例，执行以下操作：
从输入中读取整数 a 、 b 和 c 。
 - 计算每种类型歌曲的总时长 (1 分钟、2 分钟、3 分钟)。
 - 计算所有歌曲的总时长。
 - 计算音乐会时长之间可能的最小分钟差。
 - 打印音乐会时长之间可能的最小差异 (分钟)。
4. 调用函数 `min_difference_in_concerts`，输入参数为 t 和 `test_cases`。

```
## 代码:  
def min_difference_in_concerts(t, test_cases):  
    对于测试用例中的每个 case: a,  
    b, c = case total_duration = a +  
    2*b + 3*c  
    if 总时长 % 2 == 0: print(总时长  
    // 2) else: print((总时长 - 1) // 2)
```

```
t = int(input())  
测试用例列表 = []  
for _ in range(t): a, b, c =  
map(int, input().split())  
test_cases.append((a, b, c))
```

```
min_difference_in_concerts(t, 测试用例)
```

```
## 测试报告:  
## 测试未通过: 没有  
测试用例通过。
```

```
## 测试失败:
```

```
Input:  
输入:  
1 1 1  
Translated Text:  
输入  
5 5 5  
1 1 2
```

```
Expected Output:  
Translated Text: 预期输出:
```

Source Text: The study investigates the impact of climate change on agricultural productivity in the region.
Translated Text: 该研究调查了气候变化对该地区农业生产力的影响

Modified Planning:
Let's think step by step to modify Python3 Code for solving this problem.

Important:

Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will fail the test cases.

Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (2nd iteration)

Modified Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function $\text{min_difference_in_concerts}(t, \text{test_cases})$ that takes t and test_cases as input parameters.
3. For each test case in test_cases , do the following:
 - Read the integers a , b , and c from input.
 - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
 - Calculate the total duration of all songs.
 - Calculate the minimum possible difference in minutes between the concert durations.
 - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

Code:

```
```Python3
def min_difference_in_concerts(t, test_cases):
 result = []
 for case in test_cases:
 a, b, c = case
 total_duration = a + 2*b + 3*c
 if total_duration % 2 == 0:
 result.append(total_duration // 2)
 else:
 result.append((total_duration + 1) // 2)
 return result

t = int(input())
test_cases = []
for _ in range(t):
 a, b, c = map(int, input().split())
 test_cases.append((a, b, c))

output = min_difference_in_concerts(t, test_cases)
for res in output:
 print(res)
```

```

```
## 修改规划：## 让我们一步步思考修改 Python3 代码以解决这个问题。
```

重要：

注意：严格遵循输入和输出格式。输入应从标准输入获取，输出应输出到标准输出。如果你正在编写函数，则在函数定义之后，使用`input()`函数获取输入，然后使用指定参数调用函数，最后打印函数的输出。不要添加额外的打印语句，否则将无法通过测试用例。

```
## 您的回答必须包含修改后的计划和随后在 `` 块内的 Python3 代码以解决这个问题。
```

响应来自调试代理：(2 次迭代)

修改后的规划：

1. 从输入中读取整数 t ，以确定测试用例的数量。
2. 定义一个函数 $\text{min_difference_in_concerts}(t, \text{test_cases})$ ，该函数接受 t 和 test_cases 作为输入参数。
3. 对于 test_cases 中的每个测试用例，执行以下操作：
从输入中读取整数 a 、 b 和 c 。
 - 计算每种类型歌曲的总时长（1分钟、2分钟、3分钟）。
 - 计算所有歌曲的总时长。
 - 计算音乐会时长之间可能的最小分钟差。
 - 将可能的最小差值添加到列表中。
4. 返回最小可能差异的列表。

代码：

```
```Python3 def min_difference_in_concerts(t,  
test_cases): result = []
```Python3 def 最小差异演唱会(t, 测试用例): 结  
-   对于测试用例中的每个 case: a,  
b, c = case total_duration = a +  
2*b + 3*c  
if 总时长 % 2 == 0: 结果.append(总时长  
// 2) else: 结果.append((总时长 + 1) // 2)  
return 结果
```

```
t = int(input())  
测试用例列表 = []  
for _ in range(t): a, b, c =  
map(int, input().split())  
test_cases.append((a, b, c))
```

```
输出 = min_difference_in_concerts(t, 测试用例) for  
输出 in 输出: 打印(输出)
```

Decision: Again failed because of incorrect implementation. So, the Debugging Agent tries to improve the code again.

Prompt for Debugging Agent: (3rd iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

Relevant Algorithm to solve the next problem: {Retrieved algorithm}

Problem to be solved: {Original problem}

Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
 - Read the integers a, b, and c from input.
 - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
 - Calculate the total duration of all songs.
 - Calculate the minimum possible difference in minutes between the concerts durations.
 - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

Code:

```
def min_difference_in_concerts(t, test_cases):  
    result = []  
    for case in test_cases:  
        a, b, c = case  
        total_duration = a + 2*b + 3*c  
        if total_duration % 2 == 0:  
            result.append(total_duration // 2)  
        else:  
            result.append((total_duration + 1) // 2)  
    return result  
  
t = int(input())  
test_cases = []  
for _ in range(t):  
    a, b, c = map(int, input().split())  
    test_cases.append((a, b, c))  
  
output = min_difference_in_concerts(t, test_cases)  
for res in output:  
    print(res)
```

Test Report:

Tested passed:

No test cases passed.

Tests failed:

Input:

4

决策：再次因实现错误而失败。因此，调试代理再次尝试改进口代码。

调试代理提示：（3 次迭代）

给定一个竞赛编程问题，你已经生成了用于解决问题的 Python3 代码。但是生成的代码无法通过样例测试用例。改进你的代码以正确解决问题。

```
## 解决下一个问题的相关算法: {检索到的算法} ## 需要解决的问题: {原  
始问题}
```

规划:

1. 从输入中读取整数 t ，以确定测试用例的数量。
2. 定义一个函数 `min_difference_in_concerts(t , test_cases)`，该函数接受 t 和 `test_cases` 作为输入参数。
3. 对于 `test_cases` 中的每个测试用例，执行以下操作：
 从输入中读取整数 a 、 b 和 c 。
 - 计算每种类型歌曲的总时长（1分钟、2分钟、3分钟）。
 计算所有歌曲的总时长。
 计算音乐会时长之间可能的最小分钟差。
 - 将可能的最小差值添加到列表中。
4. 返回最小可能差异的列表。

代码:

```
def min_difference_in_concerts( $t$ , test_cases):  
    result = []  
  
    对于测试用例中的每个 case: a,  
        b, c = case  
        total_duration = a +  
        2*b + 3*c  
  
        if 总时长 % 2 == 0: 结果.append(总时长  
        // 2) else: 结果.append((总时长 + 1) // 2)  
    return 结果
```

```
t = int(input())  
测试用例列表 = []  
for _ in range(t): a, b, c =  
    map(int, input().split())  
test_cases.append((a, b, c))
```

```
输出 = min_difference_in_concerts(t, 测试用例) for  
输出 in 输出: 打印(输出)
```

```
## 测试报告:  
## 测试未通过: 没有  
测试用例通过。
```

```
## 测试失败:  
Input:  
输入:
```

Translated Text:
输入:

```
1 1 1  
2 1 3  
5 5 5  
1 1 2
```

Expected Output:

```
0 1 0 1
```

Your Output:

```
3 7 15 5
```

Modified Planning:

Let's think step by step to modify Python3 Code for solving this problem.

Important:

Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.

Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (3rd iteration)

Modified Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
 - Read the integers a, b, and c from input.
 - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
 - Calculate the total duration of all songs.
 - Calculate the minimum possible difference in minutes between the concerts durations.
 - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

Code:

```
'''Python3
def min_difference_in_concerts(t, test_cases):
    result = []
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            result.append(0)
        else:
            result.append(1)
    return result

t = int(input())
```

```
1 1 1  
2 1 3  
5 5 5  
1 1 2
```

Expected Output:

Translated Text: 预期输出:

Source Text: The study investigates the impact of climate change on agricultural productivity in the region.

Translated Text: 该研究调查了气候变化对该地区农业生产力的影响

修改规划: ## 让我们一步步思考修改 Python3 代码以解决这个问题。

重要:

注意: 严格遵循输入和输出格式。输入应从标准输入获取, 输出应输出到标准输出。如果你正在编写函数, 则在函数定义之后, 使用`input()`函数获取输入, 然后使用指定参数调用函数, 最后打印函数的输出。不要添加额外的打印语句, 否则将无法通过测试用例。

您的回答必须包含修改后的计划和随后在 `` 块内的 Python3 代码以解决这个问题。

响应来自调试代理: (3 次迭代) _____

修改后的规划:

1. 从输入中读取整数 t, 以确定测试用例的数量。
2. 定义一个函数 `min_difference_in_concerts(t, test_cases)`, 该函数接受 t 和 test_cases 作为输入参数。
3. 对于 test_cases 中的每个测试用例, 执行以下操作:
从输入中读取整数 a、b 和 c。
 - 计算每种类型歌曲的总时长 (1 分钟、2 分钟、3 分钟)。
 - 计算所有歌曲的总时长。
 - 计算音乐会时长之间可能的最小分钟差。
 - 将可能的最小差值添加到列表中。
4. 返回最小可能差异的列表。

代码:

```
'''Python3 def min_difference_in_concerts(t,
test_cases): result = []'''
'''Python3 def 最小差异演唱会(t, 测试用例): 结
    对于测试用例中的每个 case: a,
    b, c = case total_duration = a +
    2*b + 3*c
    if total_duration % 2 ==
    0: result.append(0) else:
    result.append(1) return
result
如果 total_duration % 2 等
于 0: result.append(0) 否

t = int(input())
```

```
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

output = min_difference_in_concerts(t, test_cases)
for res in output:
    print(res)
...
```

Decision: This time all Sample I/O passed. We return this implementation as the Final Code for final evaluation. Finally, we evaluated this code with a private test case, and our implementation passed the evaluation. This way Debugging Agents debug the code like a human.

```
测试用例列表 = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

输出 = min_difference_in_concerts(t, 测试用例)
for 输出 in 输出:
    打印(输出)
```

决策：这次所有样本输入/输出都通过了。我们将此实现作为最终代码提交进行最终评估。最后，我们使用一个私有测试用例评估了此代码，我们的实现通过了评估。这样调试代理就像人类一样调试代码。