`MapCoder`: Multi-Agent Code Generation for Competitive Problem Solving

Anonymous ACL submission

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, w/ multiple LLMs ablations, and analyses across seven 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 & varying problem difficulties.

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. W/ 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 **M**ulti-**A**gent **P**rompting 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 & 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, which iteratively evaluates the generated code against test cases, reflects on mistakes, and modifies accordingly (Shinn et al., 2023; Chen et al., 2022). However, such approaches have limitations too. Firstly, while previous studies indicate that superior problem-solving capabilities are

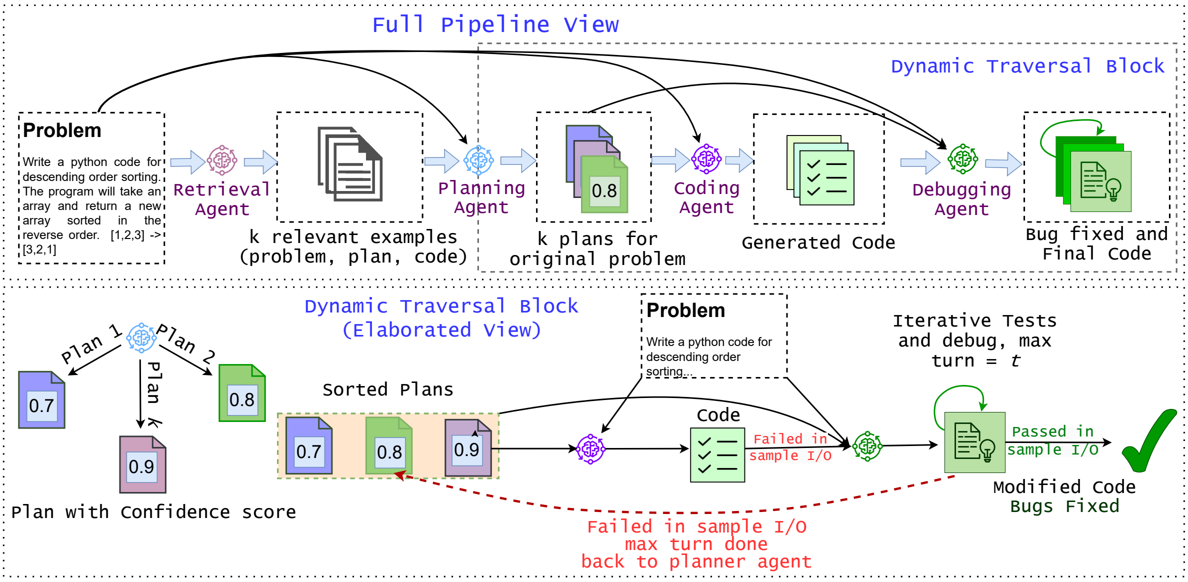

Figure 1: Overview of MapCoder (top). It starts with a retrieval agent that generates relevant examples itself, followed by planning, coding, & iterative debugging agents. Our dynamic traversal (bottom) considers the confidence of the generated plans as their reward/utility scores & leverage them to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

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 0-shot manner. Consequently, their gains can be limited.

To confront the above challenge, we develop MapCoder augmenting the generation procedure w/ possible auxiliary supervisions. We draw inspiration from the cognitive processes of real programmers and their adept utilization of various learning signals throughout the procedure. 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 self-reflection methods rely on extra test cases generated by LLM agents 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, w/o 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 & varying problem difficulties. Furthermore, with detailed ablation studies, we analyze MapCoder to provide more insights.

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 ap-

Approach	Self-retrieval	Planning	Additional test cases generation	Debugging
Reflexion	✗	✗	✔	✔
Self-planning	✗	✔	✗	✗
Analogical	✔	✔	✗	✗
AlphaCodium	✗	✗	✔	✔
MapCoder	✔	✔	✗	✔

Table 1: Features in code generation prompt techniques.

proaches (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., 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 complete solutions.

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, & plan) to provide the following agents as auxiliary data. We adopt a specific sequence of instructions, which is crucial for the prompt's effectiveness. In particular, initially, we instruct the LLM to produce similar and distinct problems and their solutions, facilitating problem planning reverse-engineering. Then, we prompt the LLM to generate solution code step-by-step, allowing post-processing 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. 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

```
Planning Agent

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*.

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 inevitable *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

Datasets: For extensive evaluation, we have used seven benchmark datasets: four from basic programming and three from complex competitive programming domains. Four basic programming datasets are: **HumanEval** (Chen et al., 2021a), **HumanEval-ET** (Dong et al., 2023), **MBPP**) (Austin et al., 2021), and **MBPP-ET** (Dong et al., 2023). HumanEval-ET and MBPP-ET extend HumanEval and MBPP, respectively 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

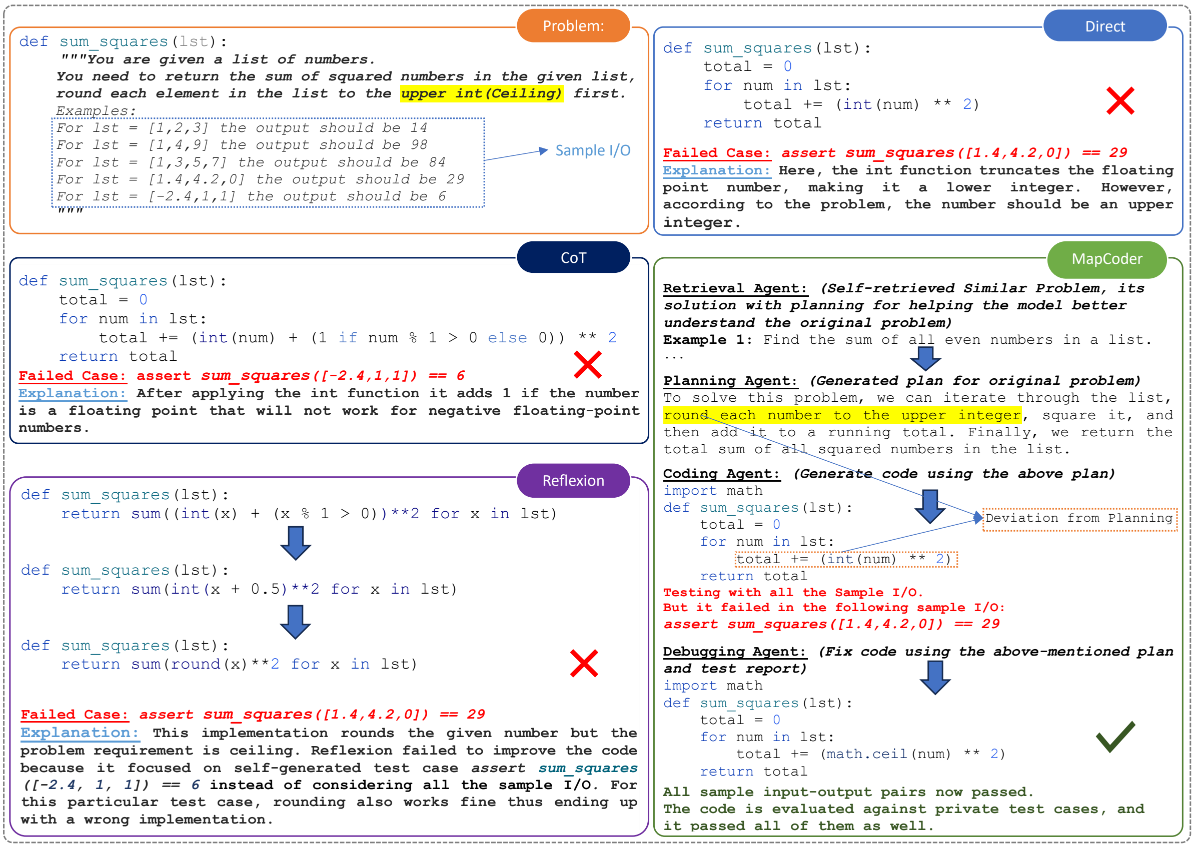

Figure 4: Example problem & solution generation using Direct, CoT, Reflexion, & MapCoder prompts. MapCoder explores high-utility plans first and uniquely features a plan-derived debugging for enhanced bug fixing.

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, 106, and 156 problems, respectively, in our experiments.

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. **AlphaCodium** (Ridnik et al., 2024) iteratively refines code based on AI generated input-output tests.

Foundation Models, Evaluation Metric, k, t W/ $k = t = 5$ in HumanEval, and $k = t = 3$ for others, we evaluate all competitive methods using ChatGPT (gpt-3.5-turbo-1106), GPT-4 (gpt-4-1106-preview) from OpenAI and Gemini Pro from Google. We have used the Pass@k evaluation metric, where the model is considered successful if at least one of 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 w/ GPT-4 are higher than ChatGPT.

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 Hu-

LLM	Approach	Simple Problems				Contest-Level Problems		
		HumanEval	HumanEval ET	MBPP	MBPP ET	APPS	xCodeEval	CodeContest
ChatGPT	Direct	48.1%	37.2%	49.8%	37.7%	8.0%	17.9%	5.5%
	CoT	53.9%	45.5%	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%	57.9%	70.5%	46.1%	6.7%	15.1%	7.3%
	Reflexion	67.1%	49.4%	73.0%	47.4%	-	-	-
	MapCoder	**80.5%** ↑ 67.3%	**70.1%** ↑ 88.5%	**78.3%** ↑ 57.3%	**57.3%** ↑ 44.3%	**11.3%** ↑ 41.3%	**27.4%** ↑ 52.6%	**12.7%** ↑ 132.8%
GPT4	Direct	80.1%	73.8%	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%	58.4%	40.3%	12.0%	26.4%	10.9%
	Reflexion	91.0%	78.7%	78.3%	51.9%	-	-	-
	MapCoder	**93.9%** ↑ 17.2%	**82.9%** ↑ 12.4%	**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 green texts indicate SoTA results, and the red text is gain over Direct Prompting approach..

Approach	CodeContest (Pass@5)	
	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. AlphCodium result are from Ridnik et al. (2024). The green cells indicate the SoTA and the red text indicates improvement w.r.t Direct approach.

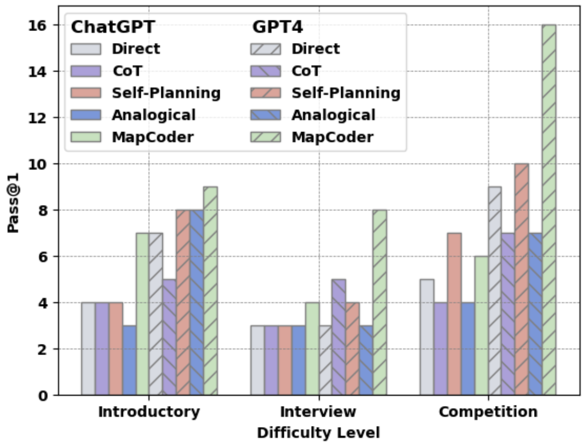

Figure 5: Performance vs problem types (APPS).

manEval, 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, w/ GPT-4, our Pass@1 surpasses Reflexion by ~3%. On top, in all four simple programming datasets, `MapCoder` enhances the Direct prompting significantly w/ a maximum of 88% on HumanEvalET by ChatGPT.

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 compare 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, even out-

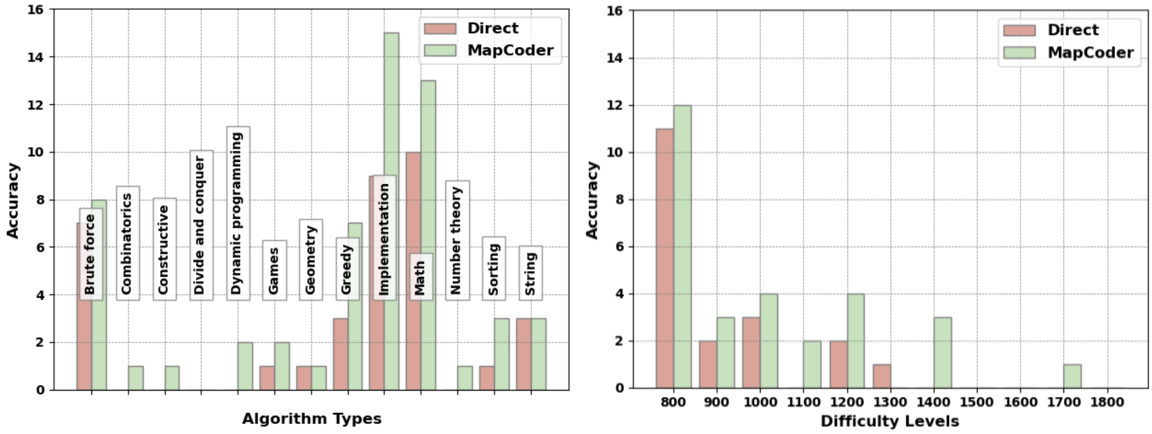

Figure 6: The number of correct answers wrt algorithm types (tags) and difficulty levels (xCodeEval dataset).

performing the concurrent state-of-the-art model MoTCoder (Li et al., 2023) by 1.2% Pass@1.

Performance with Varying Difficulty Levels: The APPS dataset comprises problems categorized into three difficulty levels: (i) Introductory, (ii) Interview, and (iii) Competition. Figure 5 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 levels[1] and tags[2] present in the xCodeEval dataset. The results of this comparison are depicted in Figure 6. 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.

Performance Across Different Programming Languages and LLMs To show the robustness of `MapCoder` across various LLMs, we evaluate `MapCoder` using Gemini Pro, a non OpenAI SoTA LLM in Table 4. As expected, our method shows perfromance gains over other baseline approaches in equitable trend on both simple (HumanEval) and contest-level problems (CodeContest). 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` w.r.t baselines.

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

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

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 w/ using Gemini Pro. The red text is gain over Direct Prompting approach.

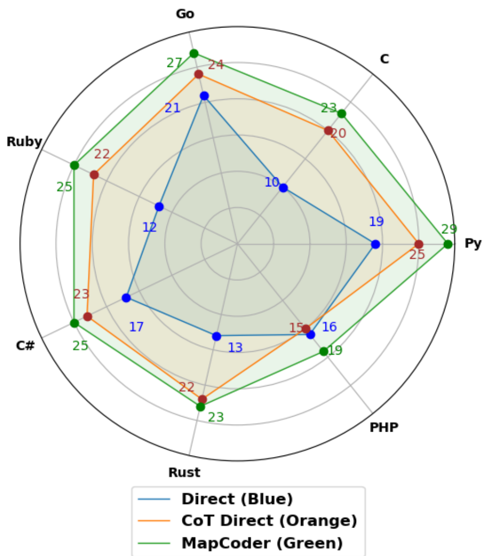

Figure 7: The number of correct answers wrt different programming languages (xCodeEval dataset).

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.

Impact of Different Agents We have also conducted a study by excluding certain agents from our

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

MapCoder, which helps us investigate each agent's impact in our whole pipeline. As expected, the results (Table 5) 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 5), we perform an ablation study of our multi-agent framework investigate each agent's impact in our whole pipeline.

Qualitative Example To verify the above numerical significance, and to understand how infact, 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.

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 6) reveal that higher k, t is proportionate performance gain at the expense of time.

Dataset Name	k \ t	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 6: Pass@1 results by varying k and t.

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 a 1.5% gain.

Time and Token Analysis Avg. number of API calls, tokens (thousands), and time required for solving a problem in Table 7 to estimate the LLM usage cost:

				Average		
LLM	Dataset	k	t	API Calls	Tokens (k)	Time (min)
ChatGPT	HumanEval	5	5	17	10	0.5
	MBPP	3	5	12	5	0.2
	APPS	3	5	21	27	1.3
	xCodeEval	3	5	19	24	1.2
	CodeContest	3	5	23	35	1.7
GPT4	HumanEval	5	5	15	13	5.9
	MBPP	3	5	8	5	2.3
	APPS	3	5	19	32	14.8
	xCodeEval	3	5	14	23	10.9
	CodeContest	3	5	19	39	18.1

Table 7: Avg. #API calls, tokens used, required time.

Error Analysis and Challenges Although MapCoder demonstrates strong performance compared to other methods, it faces challenges in certain algorithmic domains. For example, Figure 6 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 6), it solves a limited number of problems in categories like Combinatorics, Constructive, Number Theory, Divide & 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

We have identified several limitations of our work. Firstly, MapCoder generates a large number of tokens 7, which may pose challenges in resource-constrained environments. 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 (Roziere et al., 2023), 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 aboid potential risk.

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 Zotov. 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. 2023. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

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.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, 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 multilingual 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 Schrittwieser, 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 Schrittwieser, 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.

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.

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, Jérémy 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.

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 Fabbri, 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*.

Appendix

A Algorithm of `MapCoder`

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

Algorithm 1 `MapCoder`

```
k ← number of self-retrieved exemplars
t ← number of debugging attempts

exemplars ← RetrivalAgent(k)

plans ← empty array of size k
for example in exemplars do
    plans[i] ← PlanningAgent(example)
end for

plans ← SortByConfidence(plans)

for i ← 1 to k do
    code ← CodingAgent(code, plan[i])
    passed, log ← test(code, sample_io)
    if passed then
        Return code
    else
        for j ← 1 to t do
            code ← DebuggingAgent(code, log)
            passed, log ← test(code, sample_io)
            if passed then
                Return code
            end if
        end for
    end if
end for
Return code
```

B Details Promptings of `MapCoder`

The detailed prompting of the Retrieval Agent, Planning Agent, Coding Agent, and Debugging Agent are shown in Figures 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.

C Example Problem

Two complete examples of how `MapCoder` works by showing all the prompts and responses for all four agents is given in the following pages.

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>
    # 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>
```

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 IOs}

----------------
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.

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>

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>

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>

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.*

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: {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: (1st 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.

An example containing problem from CodeContest Dataset (k=3, 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.

 # 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 ≤ t ≤ 1000) — the number of test
cases.
Each test case consists of one line containing three integers a, b, c (1 ≤ a, b, c
≤ 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.
```

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

 # 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.

- 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>

```
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>

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

<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>

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.

- 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.

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.

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 failed 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.*

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:
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: (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 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:
```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.*

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

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: (2nd 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(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)
```

***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
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())
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.