The Dissertation Committee for Jierui Li
certifies that this is the approved version of the following dissertation:

Enhancing Competitive-level Code Generation by Utilizing Natural Language Reasoning

Committee:

Raymond Mooney, Supervisor

Jessy Li

Milos Gligoric

Huan Sun

Enhancing Competitive-level Code Generation by Utilizing Natural Language Reasoning

by

Jierui Li

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2026

Dedication

To all future Longhorn graduate students.

Epigraph

What starts here changes the world.

—University of Texas at Austin

Acknowledgments

I could not have reached any of my goals or completed the work mentioned in this thesis without the help of many people throughout the five years. I could not thank them enough.

First, I'd like to express my sincere thanks to my great advisor, Raymond Mooney, who has always been a supportive, kind, knowledgeable, visionary, and wise supervisor for my research and study at UT, who is always encouraging and insightful. He has encouraged me to think like an ML person and an NLP person, and to focus on the niche of direction I want to pursue. He has taught me many things that would benefit both my research and my life, and I know I'll carry them for many years.

I want to thank my committee members, Jessy Li, Milos Gligoric, and Huan Sun, who have been providing me with enlightening ideas and guidance through discussions in our meetings. Their expertise and perspectives in code generation have been insightful to my research, and I'm honored to have them in my committee.

Thanks to all of my co-authors who worked with me, especially my mentors from my internships, including Wei Lu, Vipul Raheja, Dhruv Kumar, Hung Le and Jifan Chen. I have learned a lot from you. I also want to thank Zhanming Jie, Szymon Tworkowski, Yingying Wu, and Shanshan Xiao, whom I worked closely with; you are great collaborators and friends.

I also want to thank my lab mates, Albert Yu, Vanya Cohen, Jordan Voas, Jialin Wu, Sheena Panthaplackel, and Prasoon Goyal, for the discussions and your valuable feedback for my papers, posters, and talks. Other colleagues in the UT community who either provide me with valuable suggestions for my research and career or great discussions during the NLL or NLSE meetings: Greg Durrett, Eunsol Choi, Elias Stengel-Eskin, Etienne Vouga, David Harwath, Leqi Liu, Amy Pavel, Kyle Mahowald, Pengyu Nie, Jiyang Zhang, Aditya Thimmaiah, Liyan Tang, Juan Diego

Rodriguez, Linghan Zhong, Anirudh Khatry, Fanqi Yan, Anuj Diwan, Zayne Sprague, Zeyu Leo Liu, Fangcong Yin, Yating Wu, Hongli Zhan, Mina Huh, Manya Wadhwa, Puyuan Peng, Xi Ye, Layne Berry, Kaj Bostrom, Jifan Chen, and all other members. I would like to thank great staff and faculty at UT for their support and lectures.

I would like to thank my research mentors and colleagues in my undergrad; without you, I would never have stepped on this path. Lemao Liu, Zenglin Xu, Lei Wang, Jipeng Zhang, Guanlin Li, Huayang li and other co-workers. I want to express my special thanks to Chris Manning and Richard Socher; my first impression of NLP is CS 224N.

I would like to thank my friends and family for their support. Thanks to my friends, Yuejia, Yijun, Jialing, Qingtong, Shuhuai, and other friends, for being there for me when I'm pursuing my PhD. Thanks to my grandparents for advocating education from my childhood. Thanks to my cousin Yunqi, who makes me happy daily. Thanks to my guinea pig Piggy for being my noisy company.

I express my deep thanks to my parents: Thanks to my mom, Jie He, who always supports me in getting higher education and teaches me to be optimistic about life. Thanks to my dad, Ze Li, who always shows me encouragement and lifts me when I feel emotionally down.

Part of the research conducted is funded by The University of Texas at Austin and Air Force Research Laboratory(AFRL) and DARPA for the KAIROS program under agreement number FA8750-19-2-1003.

Preface

Readers, this document represents a great deal of time and effort. Enjoy.

Cheers, Jierui Li.

Abstract

Enhancing Competitive-level Code Generation by Utilizing Natural Language Reasoning

Jierui Li, PhD
The University of Texas at Austin, 2026

SUPERVISOR: Raymond Mooney

Large language models have substantially advanced automatic code generation, but competitive programming remains a demanding test of their reasoning abilities. Solving these problems requires more than producing syntactically valid code: a model must abstract a narrative specification into its underlying mathematical structure, discover an efficient algorithm, develop the algorithm into a complete plan that accounts for correctness, complexity, and edge cases, and finally translate the plan into an executable program that succeeds on unseen tests under strict resource constraints. This complex process requires strategy exploration and problem-solving skills, and is beyond implementing straightforward functionalities. Human competitors typically develop these skills through algorithmic principles, worked examples, and expert-written editorials rather than through problem–program pairs alone. This thesis proposes to utilize natural language as an intermediate representation between problem understanding and reliable program synthesis.

This thesis investigates how to leverage and improve LLMs' abilities in natural-language reasoning, specifically in the context of solving competitive-level programming problems. It first decomposes competitive programming problem-solving into

strategy discovery, specific verbal solution details, and implementation. Across the evaluated settings, large language models can often explain verified human solutions and implement programs from detailed verbal descriptions even when they struggle to solve the same problems directly. These results suggest that the main bottleneck lies less in translating a complete solution into code than in discovering the correct algorithmic strategy and specifying it with sufficient precision.

Building on this diagnosis, the thesis develops an explanation-based distillation framework that first automatically generates editorial-style reasoning from verified human programs and uses these explanations to teach a separate reasoning model to guide code generation on new problems. The findings show that semantically rich natural-language supervision transfers algorithmic knowledge more effectively than directly training in the problem-to-code setting and encourages the use of more efficient strategies rather than superficial or brute-force implementations.

The thesis then introduces CodeTree, which operationalizes the separation of reasoning and implementation through an agent-guided search process: distinct roles propose strategies, implement programs, diagnose failures, and evaluate candidate solutions, while execution evidence and model-generated critique determine whether a search path should be expanded, revised, discarded, or accepted. This study finds that exploring diverse strategies and making informed search decisions are generally more effective than repeatedly refining a single initial solution, although successful role decomposition depends on the underlying model's ability to follow specialized instructions.

The thesis next examines whether stronger problem-solving performance implies transferable algorithmic understanding. AlgoSimBench evaluates whether models can recognize problems that share an underlying solution method despite differences in wording and narrative context, and demonstrates that solving individual programming tasks and recognizing reusable algorithmic structure are distinct capabilities. The results also show that comparing generated solution attempts can

expose algorithmic structure more clearly than comparing raw problem statements, even when the attempted solutions are not fully correct.

Finally, the thesis extends this investigation to code retrieval through InstEmbed, an instruction-sensitive representation-learning framework in which the desired target, definition of similarity, and relevant portion of the query jointly determine what should be retrieved. By contrasting documents that are related to the same query but valid under different instructions, the framework learns to distinguish general semantic relevance from task-specific usefulness and improves both target-aware and noise-robust retrieval.

Together, these studies establish natural language as a practical intermediate representation for learning, exploring, evaluating, and retrieving algorithmic knowledge, bridging the gap between problem and code, while showing that reliable code generation requires explicit attention to both the structure of the reasoning process and the generalizability of the representations it produces.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Programming has become a central tool for solving problems across science, engineering, and everyday applications. With the rise of large language models (LLMs), code generation has advanced rapidly, making it possible to automate tasks that once required expert knowledge. Yet competitive programming remains a particularly challenging domain: problems are complex, solutions demand precise reasoning, and small errors often lead to failure. Tang et al. (2023b) argued that LLMs rely heavily on semantic associations rather than formal symbolic reasoning, making them stronger at NL-style reasoning but poor at true symbolic logic. Understanding how LLMs can bridge the gap between natural language reasoning and executable code is therefore both a practical and scientific question, and it forms the focus of this thesis.

1.1 Background and Motivation

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022; Lyu et al., 2023), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Chen et al., 2022b; Zhou et al., 2023). Chain-of-thought (CoT) prompting (Wei et al., 2022b) addresses limitations of previous LLMs by instructing them to generate intermediate steps towards the final answer, thereby decomposing complex problems step-by-step.

LLMs trained on large corpora of natural language and code demonstrate strong performance across a range of software engineering tasks, from writing utility scripts to solving introductory programming challenges (Chen et al., 2021; Hendrycks et al., 2021). These advances highlight the potential of LLMs to serve as general assistants for reasoning about and generating functionally correct code.

However, challenges remain, particularly in complex reasoning tasks like algo-

rithmic programming. For example, the majority of human competitors still outperform advanced models like GPT-4 in Codeforces contests (OpenAI, 2023b). Complex programming problems have stringent time and space complexity constraints, where straightforward implementation methods often yield time-consuming brute-force solutions.

This gap suggests that current LLMs, while highly capable in many natural language tasks, continue to face bottlenecks in structured algorithmic reasoning and reliable program synthesis. This thesis explores a central thesis: the path to superhuman performance in competitive programming lies in explicitly structuring and leveraging the natural language reasoning capabilities of LLMs as a scaffold for code generation. The research background is in Chapter 2.

1.2 Research Questions

To investigate this thesis, my research follows a deliberate methodological arc, where the findings from each stage motivate the inquiry of the next. This progression is guided by six core research questions.

1.2.1 Diagnose the Bottleneck

To generate the correct solution for a complex problem, we propose three stages: 1)Find the appropriate strategy and solution to tackle the programming challenge. 2)Apply the strategy and finalize details of the exact method. 3)Implement the solution as an executable program.

In Li et al. (2023b), we decompose LLMs' capabilities in problem-solving code generation to perform these 3 stages sequentially, and evaluate them separately. This helps identify the bottleneck of solving complex competitive-level programming problems (CP). We want to answer the following research question: **RQ1: In the process of solving a complex programming problem, where is the primary bottleneck for LLMs?** Relevant work is introduced in Chapter 3.

1.2.2 Utilizing Natural Language Reasoning as a Scaffold

The diagnostic results from Section 3 show that LLMs are good at explaining code, and can faithfully implement natural-language-described verbal solutions into executable programs, despite their poor abilities to directly solve them. This answers to **RQ1** and directly leads to **RQ2: Can we utilize LLMs' strength in explaining and NL reasoning to improve their problem-solving ability?**

Human-written rationales for solving algorithmic reasoning problems, known as *editorials*, are hard to collect as they are often posted on personal blogs or as tutorial videos. An alternative is to distill such natural-language-described problem-solving strategies from larger models. Distilling explicit chain-of-thoughts (CoT) reasoning processes has been shown to be an effective method to learn multi-step reasoning from larger models (Hsieh et al., 2023; Yue et al., 2023). Most reasoning-distillation pipelines obtain these traces by asking a teacher model to solve each training problem and explain its answer. However, when facing challenging tasks where state-of-the-art models struggle to generate effective solutions, it becomes infeasible to gather reasoning processes at scale.

We found from our previous research (Li et al., 2023b) that LLMs are good at explaining code, and Tang et al. (2023b) found that LLMs can learn Natural language (NL) reasoning abilities more easily than they learn symbolic language (e.g., programming language) reasoning.

In our work (Li and Mooney, 2024), we synthesize explanations of human-written code and utilize them as the chain-of-thoughts for a student model to learn. In our experiments, we found that it's more effective for LLMs to learn from natural language descriptions than to directly learn from code, which answers **RQ2**.

1.2.3 Structuring NL Reasoning Within an Agentic Framework

Having shown that separating reasoning from implementation can be useful in a reason-then-implement pipeline, the next question is whether this separation can

support autonomous search and revision. And how can we build a robust system that could perform this process?

Modern code generation pipelines (Shinn et al., 2023; Wang et al., 2023a) have adapted a general "generate → execute → refine" framework towards more robust code generation. In previous works, we separated steps for complex CP problem-solving code generation. In addition, we are interested in: **RQ3: Can we always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline?** and **RQ4: How can we build an agent that systematically explores this decomposed reasoning space?**

1.2.4 Probing the Generalization of Algorithmic Reasoning

Improved solve rates do not by themselves reveal whether a model has learned reusable algorithmic structure or has merely become better at generating, testing, and selecting candidate programs. Although CP is a demanding benchmark (Li et al., 2022; Shi et al., 2024), end-to-end program correctness evaluates only one manifestation of algorithmic reasoning. It is not clear if the skills learned generalize beyond code generation.

This raises our next question, **RQ5: Does a model's ability to solve specific problems reflect a deeper, generalizable understanding of the underlying algorithms?** In our work AlgoSimBench (Li and Mooney, 2025), 402 multiple-choice questions were curated in an adversarial setting: each given reference problem is paired with one algorithmically similar problem and three distractors that are semantically close but algorithmically dissimilar. This design forces models to rely on algorithmic reasoning rather than superficial textual cues. Our evaluation shows that LLMs consistently struggle under this setting.

To address this gap, we propose Attempted Solution Matching (ASM), which compares model-generated first-attempt solutions rather than raw problem state-

ments. ASM-NL and ASM-PL improve average selection accuracy by 9.7 and 8.3 percentage points, respectively, and remain useful even when many generated programs are not fully correct.

1.2.5 Learning Instruction-sensitive Representations for Code

AlgoSimBench further shows that generic dense embedding models perform poorly when relevance is defined by algorithmic structure rather than broad semantic similarity. More generally, the same code-related query may require different targets depending on whether the instruction asks for equivalent functionality, a shared algorithm, documentation, tests, or another artifact.

This leads to **RQ6: Can we leverage the instruction-following abilities of modern large language models to learn a task-aware and instruction-sensitive embedding model that can retrieve different targets following specific instructions?** We propose InstEmbed, a contrastive training framework that preserves the instruction-following capabilities of base LLMs and enforces instruction-sensitivity in the embedding space. InstEmbed makes the retrieval instruction part of the relevance definition rather than treating it only as a task prefix. We model instructions across three critical dimensions: similarity definition, target object, and contextual focus. We construct hard negative documents semantically similar but violate the specific instruction. InstEmbed successfully follows fine-grained instructions, achieving 24.45% absolute precision@1 improvement on average in our task-aware diagnostic tests while maintaining on-par state-of-the-art performance on standard benchmarks like MTEB-code.

1.3 Thesis Outline

The remainder of this thesis details the completed and future work that addresses these research questions.

- Chapter 2 provides background on large language models for code generation and frames the core tasks.
- Chapter 3 presents the first work, "Explaining CP Solutions using LLMs", which aims to answer **RQ1**.
- Following the findings from Chapter 3, Chapter 4 presents "Distilling Algorithmic Reasoning from LLMs via Explaining Solution Programs", which leverages LLMs' abilities to explain human-written code and fine-tune a model to generate explanations as chain-of-thought. This chapter aims to answer **RQ2**
- Chapter 5 introduces CodeTree: Agent-guided Tree Search for Code Generation with LLMs, which answers **RQ3** and **RQ4**. It separates strategy generation (i.e., natural language solution), code implementation, debugging, and evaluation within an agent-guided tree search framework.
- Chapter 6 presents "AlgoSimBench: Identifying Algorithmically Similar Problems for Competitive Programming." It addresses **RQ5** by testing whether models can recognize shared solution structure independently of surface-level semantic similarity.
- Chapter 7 follows the findings from Chapter 6, which proposes Instruction-sensitive code embedding that can understand intent and target in the instruction. It gives an answer to **RQ6**.
- Chapter 8 outlines future work on integrating instruction-sensitive embeddings into agentic software-engineering systems, a new architecture to learn embeddings, and jointly optimizing retrievers and generators for retrieval-augmented code generation.
- Chapter 9 is a conclusion for this thesis. It summarizes the main contributions and findings of existing works, and future directions.

Chapter 2: Research Background and Related Works

2.1 Emergence of Large Language Models

Vaswani et al. (2017) proposed the Transformer architecture to only use the self-attention mechanism to obtain context-rich representations of tokens while encoding all tokens simultaneously. This design has opened the doors for efficient training and inference for LLMs. Efforts have been made to improve the structure in terms of attention mechanisms (Dao et al., 2022; Choromanski et al., 2022), more expressive position embeddings (Shaw et al., 2018; Dai et al., 2019; Su et al., 2023b), and efficiency (Shoeybi et al., 2020; Kwon et al., 2023).

Devlin et al. (2019) follows the Transformer architecture and learns language representation through pre-training on a large corpus. Other masked language models have been developed to improve upon BERT (Liu et al., 2019; Beltagy et al., 2020; Sanh et al., 2020; Clark et al., 2020).

In parallel, a lineage of causal language models, pre-trained with an autoregressive objective, gained prominence. This began with the Generative Pre-trained Transformer (GPT) (Radford et al., 2018) and evolved with GPT-2, which demonstrated that a sufficiently scaled model could perform a variety of tasks without explicit supervision, acting as an unsupervised multitask learner (Radford et al., 2019). This trajectory culminated in models like GPT-3(Brown et al., 2020), where immense scale unlocked the emergent ability of in-context learning (ICL), allowing the model to perform tasks given only a few demonstrations in the prompt, without any gradient updates (Brown et al., 2020; Gao et al., 2021a). Further research revealed that complex reasoning could be elicited from these large models through Chain-of-Thought (CoT) prompting, which encourages the model to generate intermediate reasoning steps before providing a final answer (Wei et al., 2022b,a).

The rapid growth in model size and capability was systematized by the discovery of neural scaling laws, which established that model performance, measured by cross-entropy loss, scales predictably as a power-law with model size, dataset size, and the compute used for training (Kaplan et al., 2020; Hoffmann et al., 2022). These findings provided a principled framework for efficiently allocating computational budgets, demonstrating that larger models are more sample-efficient and that the optimal strategy involves training very large models on relatively modest amounts of data.

To adapt these powerful pre-trained models for specific downstream applications, various fine-tuning methodologies were developed. The Text-to-Text Transfer Transformer (T5) introduced a unified framework that treats every NLP task as a text-to-text problem, enabling a single model to be fine-tuned on a diverse mixture of tasks (Raffel et al., 2020). Building on this, instruction tuning was proposed to enhance zero-shot generalization by fine-tuning models on a collection of tasks described via natural language instructions (Wei et al., 2022a; Chung et al., 2022; Sanh et al., 2022; Wang et al., 2023b). Concurrently, knowledge distillation emerged as a key technique for model compression, enabling the transfer of capabilities from a large "teacher" model to a smaller "student" model to create more efficient deployments (Hinton et al., 2015; Sun et al., 2019; Xu et al., 2024).

As LLMs became more capable, aligning their behavior with human intent became a critical challenge. Reinforcement Learning from Human Feedback (RLHF) was established as a powerful, multi-stage process to steer models toward generating outputs that are helpful, honest, and harmless. This process involves supervised fine-tuning (SFT), training a reward model on human preference data, and then optimizing the language model policy using reinforcement learning (Ouyang et al., 2022; Stiennon et al., 2020). To simplify this complex and often unstable pipeline, subsequent work introduced more direct alignment methods. Constitutional AI proposed using AI-generated feedback (RLAIF) based on a set of principles to automate the preference labeling process (Bai et al., 2022). More recently, Direct Preference Optimization (DPO) and its variants have emerged as a stable and computationally lightweight

alternative that bypasses the explicit reward modeling step, optimizing the policy directly on preference data through a simple classification loss (Rafailov et al., 2023; Ethayarajh et al., 2024; Shao et al., 2024).

Finally, to address the static nature of parametric knowledge and the tendency for models to hallucinate, Retrieval-Augmented Generation (RAG) was developed. This framework grounds LLMs in external, verifiable knowledge by combining a pre-trained parametric model with a non-parametric memory, such as a dense vector index of a text corpus. By retrieving relevant documents and providing them as context in the prompt, RAG improves factual accuracy, allows for knowledge updates without retraining, and enhances the interpretability of model outputs (Lewis et al., 2020b; Guu et al., 2020; Karpukhin et al., 2020).

2.2 Symbolic Reasoning with Large Language Models

"*Symbolic reasoning represents concepts or objects as symbols instead of numbers and manipulates them according to logical rules. Neuro-symbolic AI combines the deep learning capabilities of neural networks with symbolic reasoning for more robust decision-making. This is a fairly recent advancement and is still an emerging area of research.*" (Caballar and Stryker, 2025)

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022; Lyu et al., 2023), commonsense reasoning (Zelikman et al., 2022), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Zhou et al., 2023; Chen et al., 2022b).

Gendron et al. (2024); Tang et al. (2023b) argue that LLMs rely heavily on semantics in tokens and contexts, and struggle more when semantics are inconsistent or when symbolic/counter-commonsense reasoning is needed. Besides training LLMs on symbolic reasoning tasks (MA et al., 2024), many methods are proposed to enhance LLMs' capabilities to do symbolic reasoning.

The chain-of-thought prompting method (Wei et al., 2022b) explicitly instructs LLMs to generate intermediate steps until the final answer is reached, enabling the model to decompose problems and solve them step by step. This method, along with majority voting, has led to notable advancements in solving high-school-level mathematical problems (Lewkowycz et al., 2022). Kojima et al. (2022) generalize the idea of CoT to zero-shot learning. Another technique that builds upon CoT is the self-consistency decoding strategy (Wang et al., 2023a). This approach samples diverse reasoning paths and selects the most consistent answer, which has been shown to improve LLMs' performance on complex reasoning tasks by embracing multiple ways of thinking. Zhou et al. (2023); Yao et al. (2024) approach from a different angle of making use of sampling, and propose approaching the correct answer through multiple paths. They leverage multiple generations on the same question to find the correct answer. Yao et al. (2023); Schick et al. (2023) decompose the reasoning process into a chain of actions and utilize tools to perform actions. Similarly, Chen et al. (2022b); Zhu et al. (2023) shift from textual steps to executable programs, leveraging interpreters for exactness.

Disentangling semantic understanding from logical or formal/symbolic reasoning has been explored across a range of tasks and domains. Zhong et al. (2023) separated semantic representation from multi-step reasoning by decomposing general QA into two dedicated modules. Hua et al. (2024) introduced ContextHub to analyze the role of context in logical problems, showing that reasoning performance is strongly affected by natural-language–rich context. (Valentino et al., 2024) examined this issue in syllogistic reasoning by injecting content bias, benchmarking whether LLMs can judge an argument's formal validity independently of its plausibility. Related efforts in math word problem solving similarly aim to remove linguistic confounds by transforming problems into a more purely logical form (Calais et al., 2025). To the best of our knowledge, AlgoSimBench is the first work to tackle this disentanglement in the code-and-algorithms domain, and the first to explicitly contrast textual and algorithmic similarities to enforce deeper logical reasoning.

2.3 Code Generation

Programming has become a central tool for solving problems across science, engineering, and industry. With the rise of large language models (LLMs), automatic code generation has advanced rapidly, making it possible to automate tasks that previously required expert programming knowledge. LLMs trained on large corpora of natural language and code demonstrate strong performance across a range of software engineering tasks, from writing utility scripts to solving introductory programming challenges (Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021; Li et al., 2022). These methods usually adopt a brute-force principle by independently generating a very large number of output code candidates, so one of which might be the optimal code solution. These advances highlight the potential of LLMs to serve as general assistants for reasoning about and generating functionally correct code.

Rozière et al. (2023); Li et al. (2023d); Wang et al. (2023c) extended this line of research by allowing LLMs to learn from large-scale code-related data such as open-source GitHub repositories and code commits. Treating code generation as an autoregressive generation task, LLMs can generate code that correctly follows programming syntactic rules and is functionally correct (Chen et al., 2021; Gunasekar et al., 2023; Nijkamp et al., 2023).

Subsequently, Li et al. (2022); Chen et al. (2022a); Ni et al. (2023) proposed to utilize unit test results to filter for more prospective generated code samples. For example, Zhang et al. (2023b) utilized test outcomes as a form of feedback, while Welleck et al. (2023); Le et al. (2022) introduced LLM-generated feedback from predicted probabilities about code correctness. Shinn et al. (2023); Chen et al. (2023c); Madaan et al. (2023) focused on more natural forms of feedback such as reflections and explanations of the code. Le et al. (2024) proposed to cluster sub-module representations of code as a form of collective feedback. These studies leverage the inherent ability of LLMs to perceive arbitrary natural language contexts, including information such as environmental feedback, to iteratively fix problematic code and improve

its correctness. Bi et al. (2024); Zhong et al. (2024) extend this idea of iterative refinement of code.

More related to our work (Li et al., 2025a) is the research for enhancing and controlling the generation procedure of LLMs. Yao et al. (2024); Koh et al. (2024) simulated the step-by-step exploration of thoughts as a tree search to support NLP and multimodal tasks, respectively. In the code domain, Song et al. (2024) used tree search to guide the debugging of generated code. Islam et al. (2024) adopted a multi-agent system to support different stages of code generation, but the exploration is not explicitly defined. Chen et al. (2024) introduced a tree structure to explore sub-functions in generated code. Different from prior approaches, we introduce a tree-based structure as a unified search space for LLM agents to efficiently perform exploration throughout different stages of code generation.

Going beyond, Repository-level research instead examines whether models can exploit project-specific context and dependencies: CoderEval (Yu et al., 2024) evaluates non-standalone functions from real projects; RepoCoder (Zhang et al., 2023a) introduced iterative retrieval and generation; RepoBench (Liu et al., 2024b) and CrossCodeEval (Ding et al., 2023) formalized cross-file retrieval and completion. CodePlan (Bairi et al., 2024) further treats coordinated repository-wide edits as a planning problem. SWE-bench (Jimenez et al., 2024) marked a shift from completion to end-to-end software maintenance by requiring systems to resolve real GitHub issues through test-validated repository patches. It catalyzed systems such as SWE-agent (Yang et al., 2024), which emphasizes interactive repository navigation and execution; Agentless (Xia et al., 2024), which decomposes the task into localization, repair, and patch validation. Follow-on benchmarks broaden this setting through human validation in SWE-bench Verified, visual and front-end issues in SWE-bench Multimodal (Yang et al., 2025a), and continuously refreshed instances in SWE-bench-Live (Zhang et al., 2025a). In a nutshell, this branch of research extends code generation beyond snippets and programming contests to bug repair, feature implementation, package migration, requirements-to-code, project-level test generation, and visual software,

making cross-file retrieval, dependency reasoning, planning, tool use, and execution feedback central concerns.

2.4 Competitive-level Programming Problem Solving

A specifically related research topic within the family of code generation is competitive-level programming problem solving. Compared to simple function/snippet implementation, it requires substantial reasoning towards problem solving; Compared to software Early attempts to apply deep learning to solve competitive-level programming problems (Balog et al., 2017) utilized traditional approaches such as SMT solvers and search to generate short programs for simple problems. Polosukhin and Skidanov (2018) collected a dataset of human-written problem statements and solutions for Codeforces problems and introduced sequence model baselines that could solve a small subset of their dataset. With the advent of Transformers, AlphaCode (Li et al., 2022) achieved significant progress in solving competitive-level programming problems by attaining a rating equivalent to the top 54% of participants on Codeforces by fine-tuning LLMs in the problem-to-solution scenario with the CodeContests dataset collected from Codeforces. Notably, AlphaCode requires sampling 1M program candidates per problem to achieve a 29.6% solve rate on their test set. Zelikman et al. (2023) improves upon AlphaCode by using fewer samples for the same level of performance.

Despite these successes, competitive programming (CP) remains a particularly difficult frontier. Problems in this domain are designed to test algorithmic thinking under strict time and space constraints, where even small reasoning errors or inefficient implementations lead to failure. While models like GPT-4 (OpenAI, 2023b) can occasionally produce correct solutions, they are still far from matching strong human competitors in CP contests. This gap suggests that current LLMs, while highly capable in many natural language tasks, continue to face bottlenecks in structured algorithmic reasoning and reliable program synthesis.

One fundamental limitation lies in the type of reasoning LLMs perform. As argued by Tang et al. (2023b), LLMs rely heavily on semantic associations and are often stronger at natural language-style reasoning than at strict symbolic logic. This difference is important in programming: solving a problem requires identifying the right algorithmic strategy, working out the technical details, and only then producing correct code. Failures in any of these stages lead to incorrect solutions. Existing studies in multi-step reasoning, such as chain-of-thought prompting (Wei et al., 2022b), show that explicitly structuring intermediate reasoning steps improves accuracy on math and symbolic tasks (Zhou et al., 2023; Hua and Zhang, 2022), but CP poses additional difficulties due to large search spaces and strict efficiency constraints.

At the same time, the strengths of LLMs provide an opportunity. Our prior work in Chapter 3 has shown that models are surprisingly effective at generating explanations of code, even when they fail to produce correct solutions (Li et al., 2023b). This indicates that natural language reasoning, when separated from direct code generation, can be a powerful resource. Recent work in distillation (Hsieh et al., 2023; Yue et al., 2023) suggests that transferring reasoning traces into smaller models can improve their problem-solving ability. These findings raise the possibility of harnessing LLMs' explanatory abilities to overcome their weaknesses in direct program synthesis.

2.4.1 Other Programming Tasks

While generating correct code is a central goal, recent research has started to explore the broader space of what "understanding code" means. One direction focuses on code efficiency—generating programs that are not just correct but also optimal in time or space. Works like Shypula et al. (2024), BigOBench (Chambon et al., 2025), and ECCO (Waghjale et al., 2024) evaluate models' ability to reason about and improve algorithmic efficiency in time and space. Similarly, NoFunEval (Singhal et al., 2024) proposed to evaluate LLMs' programming abilities beyond functional correctness. Another important area is code reasoning—testing whether models can

simulate, explain, or analyze program behavior. CRUXEval (Gu et al., 2024) tests models' ability to predict program outputs given specific inputs, revealing weaknesses in dynamic reasoning. Pei et al. (2023) study whether models can identify invariants and reason about variables during code execution. EquiBench (Wei et al., 2025) poses the deceptively simple task of determining functional equivalence between two programs. Efforts have also gone into understanding how well models can explain programs. Li et al. (2023b) test LLMs on providing human-readable explanations of competitive programming solutions, evaluating both clarity and faithfulness. CodeMind (Liu et al., 2024a) proposes the reverse task—converting code into its natural language specification (Code2NL)—as a counterpart to the well-studied NL2Code direction. CodeRAGBench (Wang et al., 2025) has studied whether code retrieval can augment better code generation tasks.

Several existing works have explored generating code explanations using LLMs. MacNeil et al. (2023) integrated LLM-generated code explanations into an interactive e-book on web software development, showing that students found the generated explanations helpful. Leinonen et al. (2023) compared LLM-generated explanations with student-created explanations, finding that the LLM-created explanations were easier to understand and more accurate. Chen et al. (2023c) utilizes self-generated explanations as feedback to its self-debug. In comparison, our work targets explaining competitive-level programming problems, aiming not only to clarify the code implementation but also to point out the key idea behind the solution, its correctness, choice of algorithms, and time complexity.

Another open question is whether models trained for CP-style reasoning can generalize to broader forms of algorithmic understanding. Identifying algorithmically similar problems (ASPs)—that is, recognizing when two problems can be solved with the same underlying approach—represents one such capability. ASP identification is important both for retrieval-based systems and for assessing whether models capture deeper algorithmic structure rather than surface-level patterns. Early benchmarks

suggest that even state-of-the-art LLMs struggle in this setting, performing well below human intuition. This is discussed in Chapter 6.

2.4.2 Code Representation for Code-centered Retrieval

GPT-2 (Radford et al., 2019) and T5 (Raffel et al., 2020)showed that language modeling or unified text-to-text training can support diverse NLP tasks with minimal task-specific architecture, while GPT-3 (Brown et al., 2020) and FLAN (Wei et al., 2022a) further moved this paradigm toward prompt-based and instruction-tuned tasks that can generalize to zero-shot settings.

Task-prefixed setting has been adapted to text embedding since Su et al. (2023a), where the model is trained to encode each input together with a task and domain description, enabling a single model to produce task-specific representations across many downstream tasks. Subsequent LLM-based embedding models (Wang et al., 2024b; Chen et al., 2025a; Lee et al., 2025; Zhang et al., 2025b) substantially improved the capacity and generality of this paradigm. Despite these advances, instruction-aware embedding models still treat instructions primarily as task-level prefixes. The instruction usually identifies the retrieval scenario. Weller et al. (2024) has found that task-prefixed embedding models fail to follow detailed, long-form, case-wise instructions. However, they mainly study natural-language document retrieval where instructions refine the information need or document preference within a fixed target space. Unlike existing works, InstEmbed treats the instruction as the definition of the retrieval goal rather than only as a task prefix.

Code embeddings support semantic code search, program understanding, and retrieval-augmented code generation by mapping natural-language queries and software artifacts into a shared representation space. Early benchmarks such as CodeSearchNet (Husain et al., 2019) framed this problem as retrieving code functions from natural-language descriptions, while pretrained code models (Feng et al., 2020; Guo et al., 2021) improved code representations by leveraging paired natural language,

source code, syntax, and data-flow signals. Recent work has moved toward broader code retrieval settings. CoIR (Li et al., 2025b) unifies diverse code retrieval tasks under a common benchmark; code-specific embedding models (Liu et al., 2025; Qin et al., 2025) further improve retrieval across programming languages, task types, and mixed text-code corpora.

However, recent CodeRAG and agentic-search studies suggest that strong code embeddings are still insufficient for real-world software retrieval. Wang et al. (2025) evaluate finds that current retrievers still struggle to fetch useful contexts for CodeRAG tasks, especially under limited lexical overlap and downstream context-integration constraints. More broadly, Li et al. (2026) argue that fixed sparse or dense retrievers expose too narrow an interface for agentic search, since lexical constraints, local context checks, and multi-step refinement are difficult to express through a retrieval call alone. Jiang et al. (2025) similarly shift attention away from static embedding similarity by training LLMs to generate effective search queries for real search engines and retrievers through reinforcement learning. These works indicate that practical software retrieval requires models to respect exact task constraints and agent goals, rather than only retrieve semantically similar code. Our work addresses this issue from the embedding side by studying goal-conditioned code retrieval, where the same query can require different target modalities depending on the instruction.

2.4.3 Summary

Taken together, these observations motivate a systematic study of how LLMs can bridge the gap between natural language reasoning and executable program synthesis. The central theme of this thesis is to leverage the strengths of LLMs in explanation and natural language reasoning to improve their weaknesses in structured problem solving, symbolic reasoning, and reliable code generation for competitive programming tasks.

We pursue this theme through a sequence of investigations that form a coherent

ladder:

- First, we diagnose the bottlenecks in LLMs' ability to solve CP problems by decomposing the problem-solving process into strategy identification, detail elaboration, and implementation.
- Second, we develop methods that harness LLMs' explanatory and natural language reasoning abilities, distilling them into problem-solving hints and agent-driven reasoning pipelines.
- Third, we extend beyond single-problem solving to benchmarks that test whether models generalize algorithmic reasoning across tasks, focusing on ASP identification and code retrieval.
- Fourth, we extend our work from code generation to code embedding and propose that natural language can also enrich task-prefix-aware embedding models to be instruction-sensitive.
- Finally, we outline future directions that build on these findings, including integrating instruction-sensitive dense-embedding-based code search in a real-world coding agent framework and joint retriever–generator training for context-dependent coding.

By following this trajectory, the thesis aims to both shed light on the fundamental limitations of current LLMs in algorithmic reasoning and to propose practical pathways for improvement. In the following chapters, we will introduce completed individual works addressing challenges in the introduction and answering the mentioned research questions.

Chapter 3: Explaining Competitive-level Programming Solutions Using LLMs

This chapter presents two studies that address LLMs' difficulty with competitive-level programming. The first shows that LLMs can often explain human-written solutions and implement solutions from such explanations. The second uses this observation to synthesize problem-to-editorial training data, showing that learning problem-to-explanation mappings improves problem-solving more effectively than direct problem-to-code fine-tuning.

In this section, we (Li et al., 2023b) aim to identify the gaps in LLMs' capabilities and answer **RQ1: In the process of solving a complex programming problem, where is the primary bottleneck for LLMs?**

3.1 Overview

We treat competitive-level programming as a combined reasoning and code-generation task. Given problem–solution pairs, our method automatically generates structured natural-language explanations containing both implementation descriptions and higher-level analysis. We evaluate these explanations in two ways: whether expert authors judge them to be faithful and useful, and whether they help another LLM (i.e., without solution as context) generate correct solutions. We show that despite poor performance in solving competitive-level programming problems, state-of-the-art LLMs exhibit a strong capacity in describing and explaining solutions. On CodeContests, GPT-3.5 and GPT-4 produce comparable solution descriptions, while GPT-4 shows a better understanding of the key idea behind the solution.

3.2 Motivation

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022; Lyu et al., 2023), commonsense reasoning (Zelikman et al., 2022), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Zhou et al., 2023; Chen et al., 2022b). The chain-of-thought prompting method (Wei et al., 2022b) explicitly instructs LLMs to generate intermediate steps until the final answer is reached, enabling the model to decompose problems and solve them step-by-step. Nevertheless, challenges persist when tackling complex reasoning tasks, such as competitive-level programming problems. For instance, even powerful models like GPT-4 outperform fewer than 5% of human competitors in virtual contests from Codeforces (OpenAI, 2023b). Competitive-level programming problems epitomize problem-solving strategies for algorithmic, mathematical, geometric, and graph-theoretic problems. Solving them necessitates understanding problems, familiarity with algorithms, reasoning skills, creative algorithm development, and efficient, robust implementation.

Previous works on automatically solving programming problems focus on tasks mapping fairly detailed natural language instructions to programs. Li et al. (2022); Ni et al. (2023); Chen et al. (2022a) verified and selected candidate programs by running them on human-written or automatically generated test cases. Shojaee et al. (2023); Chen et al. (2023c); Schäfer et al. (2023) incorporated execution feedback as an intermediate step during code generation to enhance the programming ability of LLMs. While these methods yield promising results for fairly straightforward implementation tasks, they fall short on algorithmic reasoning tasks.

Table 3.1 shows a sample problem from Codeforces.[1] Compared to most instruction-to-program tasks, competitive-level programming, a problem-to-program task, is more challenging. Before implementing the program, one needs to first abstract the problem and create a mathematical representation, come up with potential

[1] `https://codeforces.com/problemset/problem/1670/A`

Problem Statement: Sign Swap

Given an array of n integers $a_1, a_2, \ldots, a_n$, where $a_i \neq 0$, check if you can make this array sorted by using the following operation any number of times (possibly zero). An array is sorted if its elements are arranged in a non-decreasing order. Select two indices i and j ($1 \leq i, j \leq n$) such that a_i and a_j have different signs. In other words, one must be positive and one must be negative. Swap the signs of a_i and a_j. For example if you select $a_i = 3$ and $a_j = -2$, then they will change to $a_i = -3$ and $a_j = 2$.

Solution:

```
a= [*map (int,input ().split ())]
h=sum (1 for v in a if v<0)
b=[abs (a[i]) * (-1 if i<h else 1) for i in range (len (a))]
print ('yes' if sorted (b)==b else 'no')
```

Solution Description: Move negative signs to the front of the array and check if it's already non-decreasing.

Solution Explanation: Swapping different signs for any number of times means you can move negative signs arbitrarily. A non-decreasing array must have negative elements ahead of positive elements and moving negative signs ahead is the optimal operation can be made.

GPT-3.5 Generated Program:

```
a = list (map (int, input ().split ()))
neg_count = sum (1 for x in a if x < 0)
pos_count = len (a) - neg_count
while True:
    if neg_count > pos_count:
        for i in range (len (a)):
            if a[i] < 0: a[i] = -a[i]; neg_count -= 1; pos_count
   += 1; break
    elif pos_count > neg_count:
        for i in range (len (a)):
            if a[i] > 0: a[i] = -a[i]; pos_count -= 1; neg_count
   += 1; break
    else: break
if all (a[i] <= a[i+1] for i in range (len (a)-1)): print ("YES")
else: print ("NO")
```

Table 3.1: An example of a problem and solution from Codeforces with human oracle Solution, Description, and Explanation. The formatting in the problem is simplified. Program generated by GPT-3.5 is incorrect.

solutions, consider time and space complexity constraints, and corner cases, and finally identify a proper problem-solving strategy.

3.3 Method

While <*problem, solution*>[2] pairs are publicly available on the practice websites (Li et al., 2022), natural language descriptions or explanations of "how to solve the problem" are hard to collect at scale, as it requires additional annotations from programming experts. Hence, we propose to automatically generate explanations using an LLM.

In the process of problem-solving, a human typically constructs a solution by progressing from a general idea to a detailed code implementation. However, explaining that solution involves a reverse approach. This entails examining the code on a line-by-line basis, interpreting the role of each function, and then rationalizing the algorithmic steps in relation to the original problem. Therefore, we design a specific-to-general explanation generation method.

While useful explanations might be generated with simple prompts such as "explain the solution", they often lack crucial information and are difficult to evaluate due to the diversity in output. To address this issue, we specifically control the aspects of explanations to include a "problem summary" showing its understanding of the problem; 3 levels of "natural language description of the problem" showing its ability to comprehend the solution from a low-level perspective to a high-level one. Those can be regarded as *Description-level* explanation. The points "used algorithm", "time complexity", "proof of correctness" are *Analysis-level* explanations, showcasing the LLM's general analysis and understanding of the solution. The specific-to-general explaining prompt is described in the left part of Figure 3.1. It is a hierarchical prompt, requesting a step-by-step explanation from detailed code

[2]A solution here refers to a correct program.

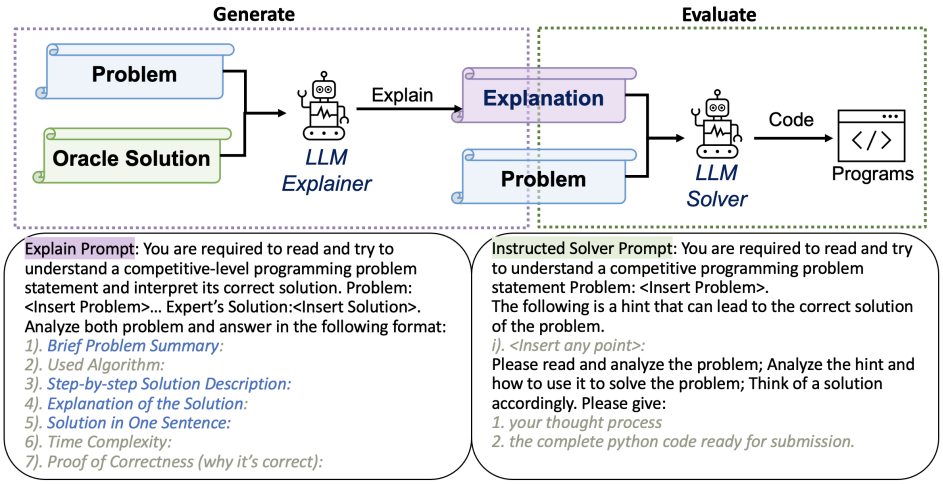


Figure 3.1: The explanation generation and evaluation framework and corresponding prompts (Top). An example of the full explain prompt (Bottom Left). The blue points are descriptions, while the grey points are analysis. We give the explanation based on the oracle solution to the instructed solver as a hint (Bottom Right) to evaluate the quality of the generated explanation.

analysis to a broader understanding of the solution.

3.4 Experiments

In this section, we aim to evaluate the specific-to-general explanations from two different perspectives: 1) How much does it satisfy the authors who wrote the solution? 2) How well do they help LLMs reconstruct correct Python implementations?

Models. We use GPT-3.5 and GPT-4 as our main models to evaluate. We use the CodeContests (Li et al., 2022) test set as our main dataset in this work. It contains 165 real online contest problems from *Codeforces*, the earliest of which dates back to Oct 2021, which is after the knowledge cutoff of GPT-3.5 and GPT-4 (Sep. 2021).

Baseline Prompt: Here's a Coding Challenge below, you will write a program to pass the given tests and the hidden tests.
Problem: <Insert Problem>.
Your goal is to analyze the problem carefully and come up with a correct and efficient solution. At the end of your response, write a complete python solution to the problem.

General-to-Specific Prompt: Baseline Prompt + "Please include the following points in your response:
1). Brief Problem Summary:
2). Useful Conditions in Description:
3). Conclusions derived from above Conditions:
4). Choice of Algorithm:
5). Solution explained:
6). Why above solution is correct:
7). Complete Python Program:

Figure 3.2: The Baseline Solver Prompt and General-to-Specific (G2S) Prompt, which ask LLMs to follow the reasoning steps till it reaches the state of implementation.

Metrics. We employ pass@k (Chen et al., 2021) as our evaluation metric for solve rate. For each problem p_i, we sample k programs generated and evaluate them using **Solve Rate@k** metric: the percentage of problems solved given 1 out of top k programs that pass all hidden test cases, when submitted to Codeforces' online judge. We first filter the programs by their output on the public test cases before submitting them, and also measure **Pass Public@k**: the percentage of solved problems with at least 1 out top k programs that can pass the public test cases given in the examples. The above metrics are abbreviated as 'solve@k' and 'public@k'.

Baseline. Baseline is the original 0-shot CoT prompting; G2S prompt is General-to-Specific prompting where it demonstrates the model to think in the reverse order of the aspects in Figure 3.1. "w/" is to only use a single aspect from the explanation as the hint, unlike using the entire explanation. For example, "w/ Used Algorithm" only gives the keywords of the algorithm to be used. The exact prompt is illustrated in Figure 3.2

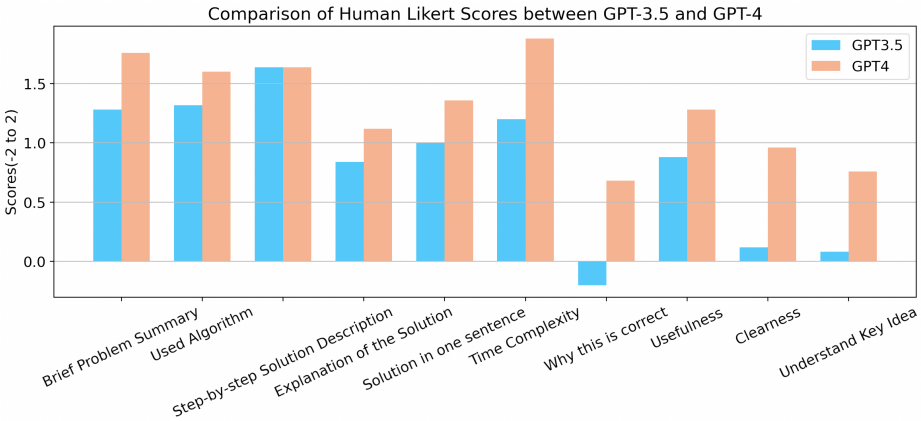


Figure 3.3: Human Likert scores (-2: very poor to 2: excellent) evaluating various aspects of the explanations.

Human evaluation: author Likert scores We measured the quality of LLM-generated explanations using human evaluation. We collected 50 <*problem, solution*> pairs from Codeforces, ensuring that their format remained consistent with those in CodeContests. Recognizing that understanding and explaining others' solutions can be a challenging task for programmers, we employed an annotator-centered evaluation approach. We extracted solutions and corresponding problems from Codeforces for an expert annotator. The Explainer then generates an explanation for the annotator's solution, which was subsequently scored by the author of the explained solution. Note that each explanation is scored by the author of the solution being explained.

We generated explanations for 50 problems with ratings ranging from 800 to 2000, along with their corresponding solutions, and provided these explanations to human experts. They were asked to assign a Likert score from -2 (very poor) to 2 (excellent).

The evaluation consists of ten questions, each one corresponding to a specific aspect of the explanation. We separately assess the quality of the response to each point of our S2G prompt. Furthermore, we developed three criteria to evaluate various aspects of the overall explanation:

1. *Usefulness*: How useful is the explanation as guidance to solve the problem?
2. *Clearness*: How good is the explanation in terms of describing everything clearly and avoiding ambiguity?
3. *Understanding*: How much does the LLM understand the key idea behind the solution?

The average Likert scores over 50 problems are shown in Figure 3.3. Regarding the scores for the solution descriptions (*Step-by-Step Solution Description, Explanation of the Solution, Solution in One Sentence*) and usefulness, both GPT-3.5 and GPT-4 Explainer are positively rated by expert annotators, with an average of 1.16 and 1.36 respectively. However, GPT-3.5 receives near-zero or negative scores on questions including *why it's correct, clearness, and understanding*, showing its inadequate ability to grasp the key idea behind the solution, while GPT-4 performs better ($0.68 \sim 0.88$ score higher) on these aspects. This reveals a clear difference in the abilities of GPT-3.5 and GPT-4 to reason and analyze competitive-level programming solutions. For GPT-3.5, we measure pass@k for $k = \{1, 5, 10\}$, but only pass@1 for GPT-4 due to access limits. To sample k programs, we sample k different human solutions for Explainer and then generate a program for each explanation.

GPT-3.5 Solver				
	solve@1	solve@5	solve@10	public@10
Baseline	1.8	3.6	6.1	13.9
G2S prompt	2.4	5.4	9.1	18.8
GPT-3.5 Solver With Silver Explanation				
w/ Used Algorithm	1.8 *(1.2)*	4.2	6.1	13.3
w/ Step-by-Step Solution Description	13.3 *(15.8)*	32.2	42.4	47.9
w/ Explanation of Solution	6.1 *(4.8)*	17.6	23.6	32.7
w/ One Sentence Solution Description	4.2 *(4.2)*	9.1	13.9	26.1
w/ Time Complexity	1.8 *(2.4)*	3.6	6.7	13.3

Table 3.2: Different aspects of explanations from GPT 3.5 effect on improving program generation. Values are percentages % and 'solve' and 'public' are short for 'Solve Rate' and 'Pass Public Tests'. Solve@1 results in *parentheses* are from GPT-4's generated explanations. The bottom 5 rows correspond to Figure 3.1's points 2,3,4,5, and 6 in the left prompt.

Automatic evaluation. We further investigated the ability of generated explanations to improve problem solving. Our fundamental assumption is that if an explanation accurately describes the algorithm, it should be capable of guiding the implementation of a correct solution. Consequently, we experimented with versions of the Instructed Solver Prompt in Figure 3.1, wherein one point in the explanation (i.e., an aspect of the solution) is provided to the GPT-3.5 Solver as a *hint* for solving the problem.

We compare it with two baseline solvers that, unlike our solver from Figure 3.1, are not conditioned on explanations and only get the problem statement as an input: zero-shot prompt (denoted as Baseline in Table 3.2) and General to Specific (G2S) "step-by-step" prompt shown in Figure 3.2. We also check that explanations do not contain code snippets to ensure the solutions are not directly leaked in explanations. However, note that it is still not a completely "fair" comparison, since the automatically generated 'silver explanations' are conditioned on oracle solutions.

For GPT-3.5, we measure pass@k for $k = \{1, 5, 10\}$, but only pass@1 for GPT-4 due to access limits. To sample k programs, we sample k different human solutions for Explainer and then generate a program for each explanation. Results are shown in Table 3.2. Different *Description-level* aspects of explanations improve both the solve rate and pass public rate. The most detailed aspect, *Step-by-Step Solution Description*, which provides a detailed natural language description of the implementation, offers the most significant benefit to problem-solving, resulting in a solve rate @1 that is 7.4 times higher than the baseline. The impact of *Explanation of the Solution* and *Solution in One Sentence* is comparatively lower due to their concise nature, which offers a less clear path towards the solution. However, providing information on the algorithms used or the expected time complexity does not improve GPT-3.5's problem-solving capabilities.

One observation from Table 3.2 is that solve@10 is significantly less than public@10. For a program that passes the public tests but fails the hidden tests, there are

two possibilities: 1) It is incorrect and only applies to a subset of test data, including the public tests; 2) It is inefficient. As discussed before, in competitive-level programming, a "correct" but slow implementation does not count as a solution, as there are constraints on time and space complexity. Therefore, we further study programs that pass the public tests but may fail hidden tests. As shown in Table 3.3, the baseline has 48.9% of its programs rejected by the online judge due to inefficiency, indicating that GPT-3.5 tends to generate inefficient implementations (e.g., brute force solutions).

	Solve	Wrong Answer	TLE	Other
Baseline	35.1%	15.6%	48.9%	0%
G2S prompt	38.3%	14.1%	47.6%	0%
w/ Used Algorithm	39.1%	18.9%	42.1%	0%
w/ Step-by-step	75.6%	11.4%	11.4%	1.6%
w/ Explanation of Solution	73.6%	11.9%	11.1%	1.4%
w/ One Sentence Solution Description	56.6%	27.9%	14.0%	1.5%

Table 3.3: Final judgment of generated programs that pass the public tests. TLE means time limit exceeded, and *other* includes memory limit exceeded and runtime error.[3]

When provided hints from the solution description, the portion of Time Limit Exceeded (TLE) programs drops significantly. Although GPT-3.5 may still make mistakes in some details or fail to consider corner cases even with hints from the explanation, it is better at avoiding inefficient solutions.

Another interesting observation is that the wrong answer rate for one-sentence explanation-instructed solving is higher than the baseline. One possible explanation is that it is challenging to incorporate corner case handling in a one-sentence solution description, which makes GPT-3.5 more likely to implement an almost-correct program.

Difficulty of the problem. We further study the aiding effect of three levels of *Solution Description* on problems of different difficulty ratings. Codeforces problems

[3]This is for all submissions, i.e., one problem might have up to k submissions, which is different from the problem-wise solve rate.

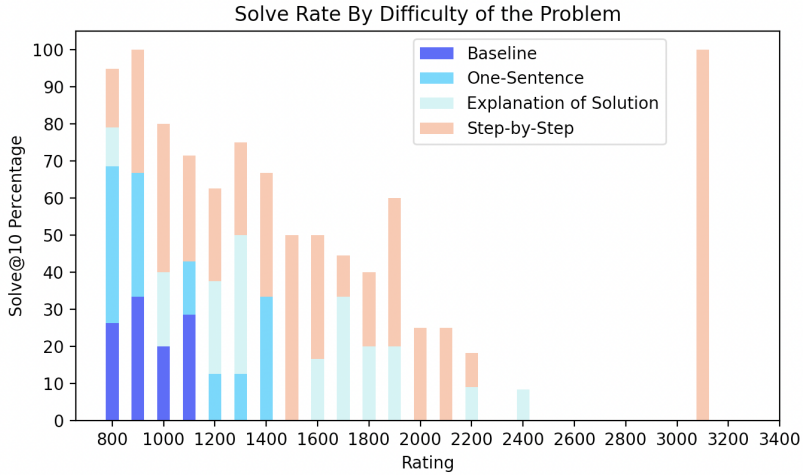


Figure 3.4: The aiding effects of 3 levels of *Solution Description* over different difficulty ratings. The difference in color shows the gain in solve@10. [4]

are given ratings, the higher the ratings are, the more challenging the problem is. Individuals who consistently solve problems with ratings of 2000 are in the 93rd percentile of all participants. As shown in Figure 3.4, the solve rate decreases as the ratings increase and no explanation can help solve complex problems. However, for easier problems, even a one-sentence hint enables GPT-3.5 to solve approximately 70% of problems, compared to the $\sim 30\%$ baseline. Furthermore, hints can effectively help to solve medium-difficulty problems which were previously unsolvable.

Sampling strategies. In our approach, we generate k programs and treat all of them as candidates without re-ranking, making the sampling strategy crucial. We therefore compared three strategies for sampling k programs.

1. Sample k human solutions: for each p_i, we sample $S_i = \{s_i^1, s_i^2, \cdots, s_i^k\}$, and for each of the solutionss_i^j, we generate one explanation e_i^j, and one corresponding program g_i^j.

[4]The 100% solve rate outlier is because there's only one rating=3100 problem in the CodeContests test set and was solved.

2. Sample k explanations: We only take the first solution s_i, and sample $E_i = \{e_i^1, e_i^2, \cdots, e_i^k\}$, for each explanation e_i^j, we generate one corresponding program g_i^j.

3. Sample k programs: We only sample 1 solution s_i and one corresponding explanation e_i, then we sample k programs $G_i = \{g_i^1, g_i^2, \cdots, g_i^k\}$ given the explanations.

w/ OneSent	public@10	solve@10
Baseline	13.9%	6.1%
Sample 10 Human Solution s_i	**26.1%**	**13.9%**
Sample 10 Explanation e_i	24.8%	12.1%
Sample 10 Programs g_i	18.2%	6.7%
Sample 10 g_i from GPT-4 explanation*	18.2%	10.9%

Table 3.4: Comparison of sampling strategies. Strategies are numbered. The first 3 sampling strategies are GPT-3.5 sampling 10 solutions/explanations/programs respectively; the last sampling strategy is GPT-4 Explainer sampling 10 programs from a single explanation.

Table 3.4 shows that the first strategy of sampling from 10 different human oracle solutions is the most effective. Additionally, the second strategy of sampling 10 explanations from one oracle solution yields better results than sampling 10 programs from one explanation (strategy 3). One potential reason is that some human solutions may have poor readability or employ complex implementations that are hard to follow. By sampling different human oracle solutions, there is a higher likelihood that explanations based on clear and concise solutions can serve as better hints. Similarly, sampling diverse explanations can mitigate the issue of misleading, incorrect explanations. We also compared the explanation quality of GPT-4 (i.e., only as an Explainer) and found it to be superior to GPT-3.5 in the same setting (10.9% vs. 6.7%). We skipped other settings due to experimental limitations.

Qualitative analysis. We observed several interesting aspects of the explanations generated by the models. Models can effectively explain code by integrating the

problem statement and the solution. Step-by-step descriptions are often more concise than line-by-line code reports by summarizing complex operations and avoiding restating well-known algorithms (e.g., depth-first-search).

A sample explanation from GPT-3.5 is given in Table 3.5. It describes the solution very well in both specific (step-by-step) and general (one-sentence) levels where it does not need to explain. It summarizes the operations of 'count < 0' and 'multiply -1 or 1' into 'negative on the left, positive on the right' and explains if it's sorted then 'yes' otherwise 'no'. However, if we look at the one-sentence description, there are ambiguous terms like 'original array' or 'move elements', which might mislead the problem-solving if interpreted incorrectly. This is due to natural languages' ambiguous nature compared to programming languages.

Models exhibit shortcomings when explaining solution correctness, as they may not comprehensively account for conditions stipulated in the problem statement. For instance, when explaining the example in Table 3.1, it failed to recognize that "swapping signs of 2 elements multiple times means moving signs arbitrarily along the array" is a crucial condition, which is not mentioned explicitly in natural language. This highlights a potential limitation in the models' ability to extract and incorporate essential information from various parts of the problem statement when generating explanations.

3.5 Conclusion

In this work, we propose explaining competitive-level programming solutions using LLMs. Given a problem and its corresponding human oracle solution given, LLMs can generate structured explanations that are positively scored by human authors. Our evaluation demonstrates that both GPT-3.5 and GPT-4 exhibit reasonable capabilities in generating faithful and clear descriptions, which can guide another LLM to better solve the problem. GPT-4 outperforms GPT-3.5 significantly in analyzing the problem and solution, as well as capturing the key ideas behind the solution.

1). Brief Problem Summary: Given an array of n integers, check if it can be sorted by swapping the signs of any two elements with different signs.
2). Used Algorithm: The algorithm used is a simple implementation of the problem statement.
3). Step-by-step Solution Description: 1. Read the input array a. 2. Count the number of negative elements in the array and store it in h. 3. Create a new array b by taking the absolute value of each element in a and multiplying it by -1 if the index is less than h, otherwise by 1. 4. Check if the sorted array b is equal to the original array b. 5. If they are equal, print "yes", otherwise print "no". **4). Explanation of the Solution:** The solution works by first counting the number of negative elements in the array. We then create a new array b by taking the absolute value of each element in a and multiplying it by -1 if the index is less than h, otherwise by 1. This ensures that all negative elements are on the left side of the array and all positive elements are on the right side. We then check if the sorted array b is equal to the original array b. If they are equal, it means that we can sort the original array by swapping the signs of any two elements with different signs.
5). Solution in one sentence: The solution works by counting the number of negative elements in the array, creating a new array with all negative elements on the left and positive elements on the right, and checking if the sorted array is equal to the original array.
6). Time Complexity: The time complexity of the solution is O (nlogn) due to the sorting operation.
7). Proof of correctness (Why this is correct): The solution is correct because it ensures that all negative elements are on the left side of the array and all positive elements are on the right side. This means that we can sort the array by swapping the signs of any two elements with different signs. The sorted array will be equal to the original array if and only if we can sort the array using the given operation.

Table 3.5: GPT-3.5's explanation to the example in Table 3.1, which understands and clearly describes the key idea behind the solution. Note it comprehends the code from a detailed level in '3)' and a general level in '5)'. blue: correct, red: incorrect)

We conclude the following findings from the experimental results above.

- Despite the poor abilities to solve CP, LLMs exhibit reasonable capabilities in generating faithful and clear descriptions and explanations of the human-written gold solutions.
- GPT-4 outperforms GPT-3.5 significantly in analyzing the problem and solution, as well as capturing the key ideas behind the solution.
- Given the step-by-step description of how to solve the problem, even the weaker LLM GPT-3.5 can generate the correct solution. This highlights the answer to RQ1: These results suggest that, once a correct high-level strategy and implementation outline are supplied, GPT-3.5 is much more capable of producing executable solutions. Therefore, the dominant bottleneck appears to lie in discovering and elaborating the algorithmic strategy, rather than in translating a detailed verbal solution into code.

When a hint is based on an oracle human solution, it effectively guides the LLM to generate improved programs for solving problems. However, a system should be able to learn from human programming solutions to improve its own problem-solving on novel problems without guidance from a human solution. This raises the question: *Can the LLM-generated explanations be utilized to improve subsequent problem-solving?* This leads to the next work.

Chapter 4: Distilling Algorithmic Reasoning from LLMs via Explaining Solution Programs

In this section, we (Li and Mooney, 2024) utilize the findings from our last work and answer **RQ2: Can we use LLMs' strength in explaining and NL reasoning to improve their problem-solving ability?**

4.1 Overview

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022; Lyu et al., 2023), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Chen et al., 2022b; Zhou et al., 2023). Chain-of-thought (CoT) prompting (Wei et al., 2022b) addresses limitations of previous LLMs by instructing them to generate intermediate steps towards the final answer, thereby decomposing complex problems step-by-step.

However, challenges remain, particularly in complex reasoning tasks like algorithmic programming. For example, the majority of human competitors still outperform advanced models like GPT-4 in Codeforces contests (OpenAI, 2023b). Complex programming problems have stringent time and space complexity constraints, where straightforward implementation methods like Chen et al. (2021); Yin et al. (2018); Hendrycks et al. (2021), often yield time-consuming brute-force solutions. Several efforts have been made to tackle this challenging task (Li et al., 2022; Zhang et al., 2023c; Olausson et al., 2023b; Ridnik et al., 2024) by adding extra clustering or verification steps to filter or iteratively refine generated programs. While those methods focus on flow engineering for code generation, there have been limited attempts to explicitly enhance models' intrinsic reasoning abilities in this context.

In this work, we propose a novel approach to distilling reasoning abilities from

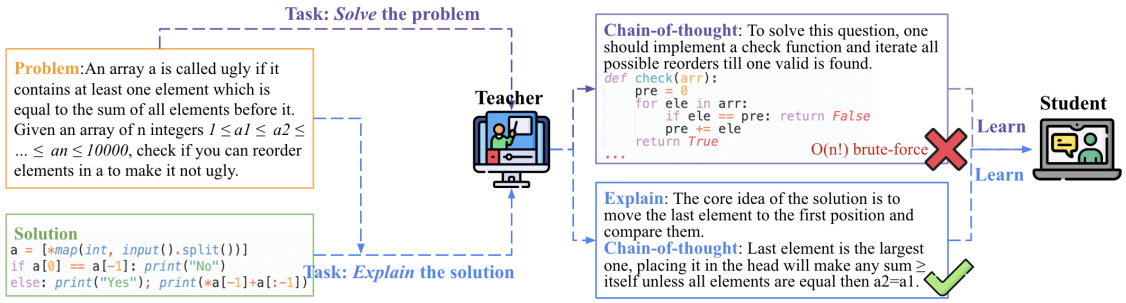


Figure 4.1: Comparison between Solve-based and Explain-based chain-of-thoughts distilling. Top: Solve-based CoT distilling is likely to generate incorrect or inefficient solutions. Bottom: Explain-based CoT distilling can generate high-quality reasoning processes by explaining the oracle solution.

LLMs by leveraging their capacity to explain solutions. We apply our method to solving competitive-level programming challenges. More specifically, we employ an LLM to generate explanations for a set of <problem, solution-program> pairs, then use <problem, explanation> pairs to fine-tune a smaller language model, which we refer to as the Reasoner, to learn algorithmic reasoning that can generate "how-to-solve" hints for unseen problems. Our experiments demonstrate that learning from explanations enables the Reasoner to more effectively guide the program implementation by a Coder, resulting in higher solve rates than strong chain-of-thought baselines on competitive-level programming problems. It also outperforms models that learn directly from <problem, solution-program> pairs. We curated an additional test set in the CodeContests format, which includes 246 more recent problems posted after the models' knowledge cutoff.

4.2 Method

Human-written rationales for solving algorithmic reasoning problems, known as *editorials*, are hard to collect as they are often posted on personal blogs or as tutorial videos. An alternative is to distill such natural-language-described problem-solving strategies from larger models. Distilling explicit chain-of-thoughts (CoT) reasoning processes has been shown as an effective method to learn multi-step rea-

soning from larger models (Hsieh et al., 2023; Yue et al., 2023). Usually, a teacher model is required to solve a set of problems while giving CoT reasoning paths at the same time, as illustrated in Figure 4.1. However, when facing challenging tasks where state-of-the-art models struggle to generate effective solutions[1], it becomes infeasible to gather reasoning processes at scale.

To tackle this problem, we propose to utilize large language models' capacities in understanding and explaining solutions rather than solving problems. Specifically, we leverage a state-of-the-art LLM to read both the problem statement and an oracle human-written solution program, and generate an *editorial*-style chain-of-thought on how to solve the problem by explaining the solution. Then, a student LLM learns the algorithmic reasoning processes from the explicit chain-of-thought paths. We compare our explain-based CoT distilling method with solve-based CoT distilling in Figure 4.1. While solve-based CoT distilling requires the teacher model to reach the correct and efficient solution to hard problems, our explain-based distilling only requires the teacher model to faithfully explain the correct solution. Since explaining competitive-level code is a feasible task for strong LLMs (section 3), explain-based distilling can yield CoT reasoning processes with high quality and less noise. propose We further proposed a reason-then-implement framework to solve algorithmic reasoning problems, utilizing the proposed explain-based CoT distillation. The framework consists of 3 major components: 1) an **Explainer** to annotate explanations for a set of <*problem, solution-program*>; 2) a **Reasoner** to learn to generate intermediate reasoning processes for a given problem; and 3) a **Coder** to implement the solution for an unseen problem *given* the output from the **Reasoner**. The framework and its fine-tuning and inference stages are presented in Figure 4.2.

Explainer: Extracting reasoning processes through explaining solution programs. The **Explainer** is tasked with explaining a solution program. It serves

[1]Experiments show that only 12% of GPT-4's generated solutions are correct given 200 problems randomly sampled from the CodeContests training set.

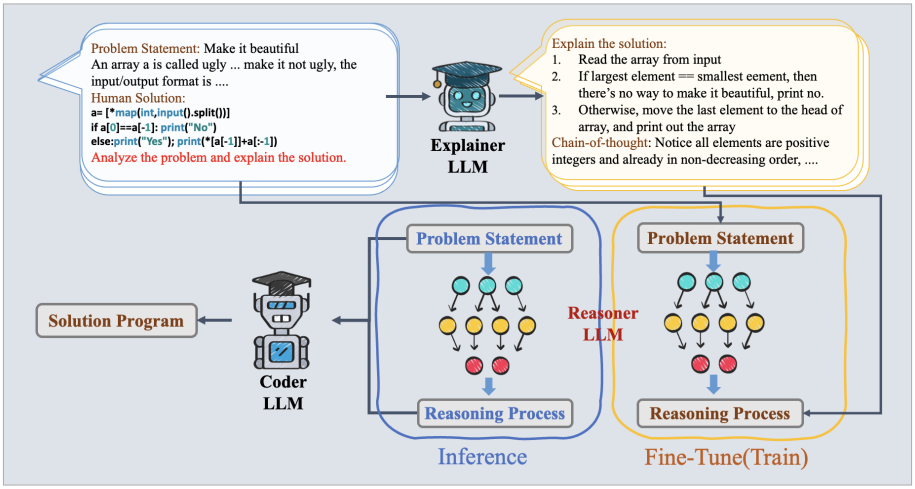


Figure 4.2: The framework of our approach. We use **Explainer** LLM to generate explanations then train **Reasoner** LLM to generate explanations given problem statements. During inference time, given the problem, the **Reasoner** can generate a reasoning process in the same format as solution explanations, which could be provided to the **Coder** as a hint to solve the problem better.

the role of Teacher in Solve-based CoT distilling, as described in Figure 4.1. Given a pair, $< p_i, s_i >$, it generates an explicit reasoning process e_i in natural language. This is inspired by how human competitors learn problem-solving skills from past problems: they learn by reading editorials, step-by-step guidelines on approaching and solving the problems. While human-written editorials are hard to collect or annotate at scale, we ask an LLM to automatically generate them. We design an editorial-style chain-of-thought reasoning template and ask the **Explainer** to follow it to explain solutions. Specifically, an editorial for an algorithmic reasoning problem refers to a comprehensive explanation or walk-through on how to solve a problem, which includes problem analysis, strategy development, solution explanation, time/space complexity analysis, etc. We leverage the capabilities of LLMs to explain solution code to generate automated explanations for problem-solution pairs $< p_i, s_i >$. The **Explainer** is prompted to create a detailed explanation, d_i, for each pair. Our focus is predominantly on the reasoning process, encompassing the following critical aspects that are often included in human-expert-written editorials:

1. Problem Restatement: Summary and analysis of the problem.
2. Conceptual Evolution: How one approaches the problem and the development of a problem-solving strategy.
3. Key to Solution: The key idea behind the solution.
4. Solution Description: Brief, verbal description of the solution, focusing on the high-level algorithm.
5. Step-by-Step Solution Explanation: A more detailed description, focusing on the steps in the implementation.
6. Common Pitfalls: Some common mistakes one could make when approaching the problem, or edge/corner cases to be considered.

Based on the length of p_i and s_i, as well as the difficulty ratings (ranging from 800 to 3600), we filter the training set from Li et al. (2022) to curate a dataset $\{< p_1, s_1 >, \cdots, < p_n, s_n >\}$. Utilizing GPT-4-0613 as the Explainer, we generate "silver" explanations for these pairs, resulting in a comprehensive problem-solution-explanation dataset $< p_1, s_1, d_1 >$, ..., $< p_n, s_n, d_n >$. After further cleaning, we have 8248 triplets in total.

Reasoner: fine-tuned to generate reasoning processes for problems. Given the inherent diversity and potential lack of readability of the code available for algorithmic reasoning problems, our approach focuses on fine-tuning the Reasoner on problem statements p_i and reasoning processes d_i, which are $< p_0, d_0 >, < p_1, d_1 > \cdots, < p_n, d_n >$. These problems and reasoning processes, rich in semantics, encapsulate the essential steps for problem resolution, including the algorithms used and specific problem-solving approaches employed. The **Reasoner** serves the role of Student in Solve-based CoT distilling, as illustrated in Figure 4.1. We adopt a weighted fine-tuning strategy, with simpler, more recent problems weighted more heavily during training. Overly challenging problems might lead to low-quality noisy explanations

that hurt the training of the **Reasoner**. Recent solutions usually feature more in-date implementations (e.g., Python 3 rather than Python 2), enhancing the code's interpretability. We fine-tune an LLM on 8,248 $\{p_i, d_i\}$ pairs, and then use the fine-tuned model as the final Reasoner. At the inference time, it generates $\hat{d}_j$ for the jth problem statement p_j. The generated reasoning process $\hat{d}$ can be considered as a natural-language "hint" given to the **Coder** to help solve the problem.

Coder: reasoner-hinted code implementer. Finally, we have a zero-shot Coder to generate code utilizing hints from the **Reasoner**. Since the focus of this work is enhancing *algorithmic reasoning* rather than *code implementation* abilities, we do not further fine-tune the **Coder**. As described earlier, the reasoning process $\hat{d}_j$ for problem p_j contains a problem restatement, conceptual evolution, key to the solution, solution description, step-by-step solution explanation, and common pitfalls, making it a detailed hint to assist the implementation of the solution. We concatenate $< p_j, \hat{d}_j >$ as the input to produce the input for the Coder, which it then uses to analyze the problem together with the reasoning hint, to generate programs that solve the problem.

4.3 Experiments

In this section, we would like to know, how much performance gain distilling from explanations can bring to the CP solving task.

4.3.1 Experimental Settings

Model. We use GPT-4-0613 (OpenAI, 2023a,b) as the **Explainer** since it's the strongest model that we have access to. We also choose the strongest models with fine-tuning access as the **Reasoner** and **Coder**. For a closed model, we choose GPT-3.5-turbo-1106 (henceforth GPT-3.5); for an open model, we choose Deepseek-

Coder-7B (henceforth Deepseek 7b) (Guo et al., 2024).[2] The temperature t is set to 0.5 when sampling multiple (>1) is needed in our main experiments. The context window is set to 4096 and we truncate single examples with more than 4096 tokens. We use **solve@k** from the previous section as our main evaluation metric.

Data. As suggested by Huang et al. (2023b), to test the capacity of closed models like GPT, using recently released data can reflect the model's performance more faithfully. To ensure novel test data given that GPT-3.5 was trained on data including problems and solutions from Codeforces, we extracted 246 recent problems from Codeforces as our main test set, *CF Prob*. It contains 246 real online contest problems, the earliest of which dates to Aug. 2023, guaranteeing they were posted after the knowledge cutoff for GPT-3.5 (Jan. 2022). We also experimented with the 165 problems from the CodeContests benchmark (Li et al., 2022) test set, whose earliest problem was published in Oct. 2021. Table 4.1 gives statistics on the difficulty ratings of both sets.

	Difficulty Ratings				
Dataset	total	[800, 1000]	(1000, 1500]	(1500, 2000]	(2000, 3600]
CodeContests	165	18.2%	17.0%	20.0%	44.8%
CF Prob	246	22.0%	23.6%	18.3%	36.2%

Table 4.1: Difficulty statistics (higher ratings = more difficult) for CodeContests and our proposed CF Prob. Both are sourced from Codeforces.

Evaluation metrics. Same as our previous work (Li et al., 2023b), we adapt pass@k (Chen et al., 2021) into **solve@k**: when given to Codeforces' online judge, it measures the percentage of problems solved by having at least *one* program in the top k that passes all hidden tests. For each problem p_i, we sample k programs generated by the **Coder**. To reduce unnecessary submissions, we first select candidate programs by executing and examining their output on the public test cases before submitting

[2] `https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5`

them. Measuring solve rates using the online judge provides a fairer comparison between models and human participants, as it also requires efficiency in solutions, rejecting brute-force solutions when an efficient algorithm exists for a problem.

We have models generate 10 programs to compute solve@10, then compute solve@1 and solve@5 by calculating the probability of having at least one solution in the top k:

$$\textit{solve@k} = \frac{1}{n}\sum_{i=1}^{n} P_i \;; \quad P_i = 1 - \frac{\binom{10-g_i}{k}}{\binom{10}{k}} \;\; \textit{if } 10 - g_i > k \;\; \textit{else } 0$$

where g_i is the number of "pass" programs in 10 tries for ith problem.

Zero-shot baselines.

- **Direct Prompting** Given the problem statement p_i, directly prompt the model to generate solution programs.
- **Naive Chain-of-thought** Adapt the well-known "Letś think step-by-step" CoT prompting proposed in (Kojima et al., 2022).
- **Editorial Chain-of-thought** Instruct the model to analyze the problem following an editorial chain-of-thought style, giving aspects identical of what's required from the **Explainer**.
- **Zero-Reasoner&Zero-Coder** Use a 0-shot Reasoner to generate natural language reasoning processes as hints for the 0-shot Coder.

Fine-tuning methods.

- **Fine-tuned Coder** Fine-tune the **Coder** to generate code given the problem, using the same 8248 <*problem, solution-program*> pairs from CodeContests (Li et al., 2022).

- **Fine-tuned Reasoner &Zero-Coder** Use the Reasoner fine-tuned on solution explanations to generate natural-language reasoning hints for the 0-shot **Coder**. In addition to using the *full* reasoning processes. We also separately tested the effect of each editorial aspect (e.g., common pitfalls) and selected the highest performing aspect as the *best* aspect.

4.3.2 Main Results

Dataset	**CF-Prob**						**CodeContests**					
Coder/Reasoner Model	GPT-3.5			Deepseek 7b			GPT-3.5			Deepseek 7b		
Zero-shot Methods												
	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10
Direct Prompt	1.1	2.7	3.3	0.4	0.9	1.6	1.9	3.5	4.2	0.8	1.2	1.8
Naive CoT	1.2	2.7	3.3	0.7	1.4	2.0	1.8	3.7	4.8	0.8	1.4	1.8
Editorial CoT	1.1	2.7	3.6	0.9	2.0	2.4	1.5	3.7	4.8	1.0	1.8	2.4
0-Reasoner Coder	1.2	2.5	3.3	0.7	1.9	2.4	1.4	3.5	4.2	1.1	2.0	2.4
Fine-tuning Methods												
Fine-tune Coder	0.5	0.8	1.6	0.6	1.7	1.8	0.8	1.2	1.8	1.1	1.4	1.9
Fted Reasoner w/Full	1.1	3.2	4.9	0.6	2.1	3.7	1.5	4.2	6.1	1.4	2.8	3.6
Fted Reasoner w/Best	1.1	3.7	6.1	1.0	2.9	4.1	2.4	5.6	7.2	1.4	2.9	4.2
Relative Increment		+37%	+69%		+53%	+71%		+51%	+50%		+45%	+75%

Table 4.2: Performance of baselines and our methods. Fine-tuned Reasoner (Full) refers to using all aspects from the reasoning process, while Fine-tuned Reasoner (Best) only uses the best aspect. Relative Increment (by percentage) is what's over the best baseline. Solve@k: Solve rates (Percentage) when sampling k.

The results are presented in Table 4.2. With our fine-tuned **Reasoner**, the **Coder** achieves the best solve@10 and solve@5 rates across open/closed models and two datasets, supporting the effectiveness of our method. Compared to using the original model as the Reasoner (0-shot Reasoner), the best aspect (Step-by-Step) from the reasoning process alone can boost solve@10 from 3.3% to 6.1%, suggesting that our fine-tuning method does distill some algorithmic reasoning ability into the student model. Note that solely fine-tuning <*problem, solution-program*>pairs actually hurts performance. One reason might be that solutions to Codeforces problems are often written under a time constraint with poor readability, making it difficult for the model to generalize given the limited amount of fine-tuning data. Since instruction-tuned models would generate intermediate thought processes by default, we experimented

to explicitly forbid the model from any reasoning/analysis preceding implementation. Compared to the direct prompt, the solve@10 rate on CF Prob w/GPT-3.5 drops from 3.3% to 2.0%. Even with this setting, the generated code often contains comments and meaningful variable/function naming that are less in human competitors' code. This observation further supports our claim that meaningful, semantic-rich natural language can help the model generalize to unseen problems.

4.3.3 Effect of Tuning and Sampling

Fine-tuning strategy. As discussed in previous sections, harder problems are more likely to exceed the Explainer's capacity, which can make their silver explanations noisier. We therefore compare three training strategies for DeepSeek-Coder-6.7B: uniform fine-tuning on the full dataset, fine-tuning with higher weights on simpler problems, and uniform fine-tuning on only the simpler subset. Table 3.8(a) shows that up-weighting simpler problems performs best, suggesting that noisy but diverse data can still help when cleaner examples receive greater emphasis. We compare three strategies to fine-tune deepseek-coder 6.7b: 1) Uniformly fine-tune on the entire dataset; 2) Fine-tune with simpler problems weighted more, and 3) Uniformly fine-tune on the simple subset only. The solve@10 rates as depicted in Table 4.3 a, indicate that the strategy of just up-weighting simpler problems yields superior results. This suggests that in scenarios where the training data is noisy and limited, including noisier data while assigning greater weight to cleaner data can serve as a viable approach to enhancing generalization.

Program sampling strategy. Since our method partitions problem-solving between the Reasoner and Coder, it can adopt various sampling strategies. Specifically, the Coder can either a) implement different programs given one result from the **Reasoner** or b) implement one program for each of the different results from the **Reasoner**. Formally, to sample k solutions, one can sample k different results from the **Reasoner**, or alternatively, sample k programs given only one reasoning result.

For problem p_i we sample M reasoning results $\hat{d}_i^{\,1}, \hat{d}_i^{\,2}, \cdots, \hat{d}_i^{\,M}$ and T programs c_j for each process $\hat{d}_i^{\,j}$, where $M \times T = k$. We show different choices of M, T in Table 4.3 b. For cases where 1 deterministic sequence is needed (T=1 or M=1), we do not sample; otherwise, temperature is set to 0.5.

Data Split	Data Size	Training Steps	Data Weight	solve@10
Simple+Hard	8248	1547	Uniform	3.3
Simple+Hard	8248	1617	2xSimple:1xHard	**3.7**
Simple	4691	1172	Uniform	3.3

(a)fine-tuning strategy comparison w/Deepseek Coder

Sample Reasoner	Sample Coder	solve@10
M=1	T=10	2.8
M=2	T=5	3.7
M=5	T=2	4.1
M=10	T=1	**4.9**

(b)sampling strategy comparison w/GPT-3.5

Table 4.3: (a)Fine-tune settings testing whether to up-weight simpler problems. Training steps are kept similar unless data points are significantly fewer. (b)Sampling more from **Reasoner** vs. sampling more from **Coder**. Both solve@10 (percent) are Fine-tuned Reasoner w/Full on CF Prob.

We found that for fixed $k = 10 = M \times T$, sampling more reasoning processes M is more important than sampling more implementations for the same reasoning process T. The diversity in reasoning often reflects usage of different strategies or algorithms. Thus, sampling from the **Reasoner** potentially allows the **Coder** to try different methods for a problem. We therefore used $temp = 0.5$ and $M = 10, T = 1$ for the experiments in Table 4.2 (a) and the following section.

4.3.4 Ablation Study

Since providing the full reasoning process with all aspects can distract the model, we found that providing a single aspect can yield a higher overall solve rate. We therefore studied the effects of using different aspects of the explanations generated by the **Reasoner**.

The results in Table 4.4 show that each aspect alone can achieve better solve rates against the un-fine-tuned **0-shot Reasoner**, proving the effectiveness of our explanation fine-tuning. Among all the different aspects, Conceptual Evolution achieves the best solve@1 and solve@5 while Step-by-Step Solution Explanation achieves the best solve@10.

	Solve@1	Solve@5	Solve@10
*0-shot Reasoner	1.2	2.5	3.3
w/ Full Reasoning process	1.1	3.2	4.9
w/ Conceptual Evolution	**1.5**	**4.1**	5.3
w/ Key to Solution	1.0	3.2	4.5
w/ Solution Description	1.1	3.5	4.9
w/ Common Pitfalls	1.2	3.4	4.9
w/ Step-by-Step Solution Explanation	1.1	3.7	**6.1**

Table 4.4: Ablation Study on the different aspects. Rather than fine-tuning another model, we give only each individual aspect of the **Reasoner**'s output to the **Coder**. Solve@k: Solve rates (percentage) when sampling k.

Since Conceptual Evolution provides a solving strategy, it indicates how one should think about the problem and where to start, while also touching on the high-level idea behind the solution, as well as the choice of algorithm. Therefore, by giving a good starting point, it allows the **Coder** to follow up the idea and implement a correct solution. On the other hand, *Step-by-Step Solution* focuses more on implementation details, giving specific instructions on what and how to implement, making it easy for the **Coder** to follow. However, it is not as robust as higher-level aspects: If the **Reasoner** makes a mistake in the details, the **Coder** may not spot the bug and produce an incorrect implementation.

4.3.5 Program Submission Status

In our experiments, we find that, compared to our method, programs sampled with zero-shot methods pass more public tests, but yield significantly lower solve rates. This indicates that the pre-trained large language models like GPT-3.5 and Deepseek Coder tend to give brute-force solution programs initially. We further study this phenomenon by analyzing the statistics of the statuses of submitted programs.

In competitive-level programming, online judge systems give TLE (Time Limit Exceeded) to submissions that do not run efficiently enough to be considered correct. As illustrated in Figure 4.3, the baseline has 42% to 57% of its programs rejected due to time inefficiency. Our fine-tuned **Reasoner**, on the other hand, has learned to

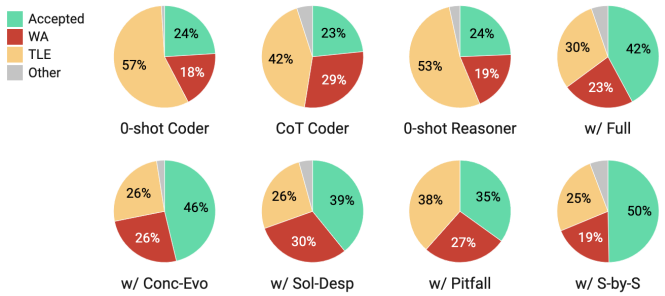


Figure 4.3: Final online judgment of programs that pass public tests. Accepted: correct; TLE: time limit exceeded; WA: wrong answer (s) on private tests; Other: memory limit exceeded, runtime error etc.[3]

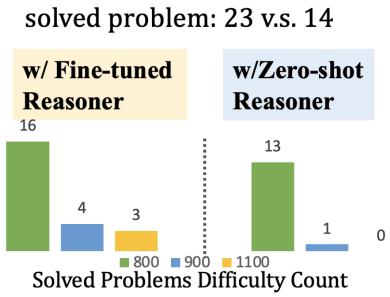


Figure 4.4: The problem difficulty statistics for problems solved with fine-tuned or zero-shot **Reasoner** when sampling 100 reasoning processes per problem.

avoid brute-force strategies. With Step-by-Step Solution Explanation, 50% of the programs that pass the public tests are accepted solutions, indicating that the Reasoner learns to use efficient algorithms rather than implementing brute-force solutions.

4.3.6 Solved Problems Analysis

Given that compared to zero-shot settings, using the Fine-tuned Reasoner only helps solve several more problems, we investigate further whether this improvement is due to learning from explanations or is merely coincidental. Thus, we aggregated all solved problems from 5 experiments with different sampling strategies and temperatures with both a zero-shot baseline Reasoner and our fine-tuned Reasoner. In other words, for each problem, we evaluated 50 programs generated under the zero-shot setting and another 50 under the explanation Fine-tuned setting. marking problems solved by either to obtain a final set of 23 "solvable" problems.

Then, we measured solve@100 for this "solvable" subset. For each problem in the solvable set, we sample 100 explanations from both the Zero-shot and Fine-

[3]This is for all submissions, i.e., one problem might have up to k submissions, which is different from the problem-wise solve rate.

Problem Statement: Yarik recently found an array a of n elements and became very interested in finding the maximum sum of a non-empty subarray. The subarray he chooses must have alternating parities for adjacent elements. For example, $[1, 2, 3]$ is acceptable, but $[1, 2, 4]$ is not, as 2 and 4 are both even and adjacent. You need to help Yarik by finding the maximum sum of such a subarray.
Fine-tuned Reasoner's Reasoning Process ✓: ... checking all possible subarrays would be inefficient. Recognizing that the problem exhibits optimal substructure and overlapping subproblems, dynamic programming becomes a natural choice. ... the maximum sum of a subarray ending at index i can be extended by including $a[i]$ if it maintains the alternating parity condition. This insight leads to the iterative update of dp and the tracking of the maximum sum.
Zero-shot Reasoner's Reasoning Process ×: To solve this problem, we can use a dynamic programming approach. We can iterate through the array and keep track of the maximum alternating sum ending at each element. We will maintain two values, one for the maximum alternating sum ending at the current element with an odd index and another for the maximum alternating sum ending at the current element with an even index.

Table 4.5: An algorithmic reasoning problem from *CF Prob* with reasoning processes from the fine-tuned and 0-shot **Reasoner**.

tuned Reasoners. By asking the same Coder to implement them, we found that the Explanation-Fine-tuned Reasoner achieves a **100%** solve@100 while Zero-shot Reasoner only achieves **60.9%** on this set.

The difficulty distribution of these problems is shown in Figure 4.4. We observed that with a fine-tuned **Reasoner**, more difficult problems can be solved, while the zero-shot original model mainly solves problems with the lowest difficulty rating (800). This highlights that learning from explanations can enhance the **Reasoner**'s ability to address more complex problems.

4.3.7 Case Study

Table 4.5 shows a problem[4]that can be solved by 14% of sampled programs from experiments with Fine-tuned **Reasoner** but none by the zero-shot Reasoner. We compare the output of our **Fine-tuned Reasoner** to that of the 0-shot **Reasoner**.

This problem can be solved using dynamic programming while specifying the rule of having "alternating parities for adjacent elements". The Fine-tuned **Reasoner** analyzes the inner logic of how to update the DP status and final answer correctly, considering the condition. By contrast, the 0-Shot **Reasoner** only gives plausible surface-level reasoning output by connecting "alternating parities" to keywords like "odd" or "even".

4.3.8 Generated Reasoning Process Analysis

We ask a human expert to evaluate the generated reasoning processes by Fine-tuned and Zero-shot Reasoner and compare them. Unlike in Table 4.5, we look at all 10 reasoning processes for both. We found that while the 0-shot model shows the correct general understanding that it can be solved as a dynamic programming problem, it fails to further reason meaningfully. 9 out of 10 reasoning processes suggest maintaining 'MaxOdd' and 'MaxEven' variables to consider odd and even positions separately. Only one reasoning process suggests a variant of Kadane's algorithm, but fails to understand what does " must have alternating parities for adjacent elements" means.

On the contrary, with the fine-tuned **Reasoner**, 3 out 10 samples realize 'odd' and 'even' should be considered as the condition to update the sum, and 2 out 10 use a greedy method instead. This observation indicates a more human-mimicking property: by learning from explanations, the **Reasoner** tends to try different strategies.

As shown in Table 4.6, the main difference between 2 programs is their used

[4]`https://codeforces.com/contest/1899/problem/C`

Problem Statement: Yarik recently found an array a of n elements and became very interested in finding the maximum sum of a non-empty subarray. However, Yarik doesn't like consecutive integers with the same parity, so the subarray he chooses must have alternating parities for adjacent elements. For example, $[1, 2, 3]$ is acceptable, but $[1, 2, 4]$ is not, as 2 and 4 are both even and adjacent. You need to help Yarik by finding the maximum sum of such a subarray.

Fine-tuned Reasoner Generated Process ✓:

... checking all possible subarrays would be inefficient. Recognizing that the problem exhibits *optimal substructure and overlapping subproblems*, *dynamic programming* becomes a natural choice. ... the maximum sum of a subarray ending at index i can be *extended* by including $a[i]$ if it *maintains the alternating parity condition*. This insight leads to the iterative update of dp and the tracking of the maximum sum.

Non Fine-tuned Reasoner Generated Process ×:

To solve this problem, we can use a dynamic programming approach. We can iterate through the array and keep track of the maximum alternating sum ending at each element. We will maintain two values, one for the maximum alternating sum ending at the current element with an odd index and another for the maximum alternating sum ending at the current element with an even index.

GPT-3.5 Coder Generated Program:

```
def max_alternating_subarray_sum (arr):
    n = len (arr)
    dp = [0] * n
    dp[0] = arr[0]
    for i in range (1, n):
        if arr[i] % 2 != arr[i-1] % 2:
            dp[i] = max (dp[i-1] + arr[i], arr[i])
        else:
            dp[i] = arr[i]
    return max (dp)

t = int (input ())
for _ in range (t):
    n = int (input ())
    arr = list (map (int, input ().split ()))
    result = max_alternating_subarray_sum (arr)
    print (result)
```

```
def max_alternating_sum (arr):
    n = len (arr)
    dp_odd, dp_even = [0] * n, [0] * n
    dp_odd[0] = arr[0]
    dp_even[0] = 0
    for i in range (1, n):
        dp_odd[i] = max (dp_even[i-1] + arr[i], arr[i])
        dp_even[i] = max (dp_odd[i-1] - arr[i], 0)

return max (dp_odd[-1], dp_even[-1])
t = int (input ())
for _ in range (t):
    n = int (input ())
    arr = list (map (int, input ().split ()))
    result = max_alternating_sum (arr)
    print (result)
```

Table 4.6: An algorithmic reasoning problem from *CF Prob* with reasoning processes from the finetuned and 0-shot **Reasoner**. The incorrect process leads to a failed solution (right side).

algorithm. The correct reasoning process can guide the **Coder** to implement the correct solution, suggesting a better understanding of the problem condition.

4.4 Conclusions and Future Work

In this study, we propose to distill large language models' abilities of explaining solutions into reasoning abilities used to help solve problems. More specifically, we use an LLM (Teacher) to explain, in natural language, the reasoning processes for <*problem, solution-program*> pairs and fine-tune a smaller LLM (Student) on such generated reasoning processes. Using a two-phase framework of reason-then-implement where the student model plays the role of a **Reasoner** to instruct a zero-shot **Coder** to generate the implementation. Experiments on real problems from Codeforces demonstrate that our explain-based distilling outperforms several strong 0-shot baselines as well as fine-tuning with code solutions alone. Although absolute solve rates remain low for challenging competitive-level programming problems, our method can significantly improve performance and solve relatively 37% to 75% more problems than the strongest baselines across all experiments. Our training set for fine-tuning only contains 8248 data points and 8M tokens, and it suggests the efficiency of the proposed distilling method. Quantitative and qualitative analyses reveal that the fine-tuned Reasoner learns to avoid brute-force implementations and favors more efficient programs compared to its non-fine-tuned counterparts. We also curated an up-to-date test set for algorithmic reasoning challenges.

Automatically generating and learning from explanations of problem solutions to improve reasoning presents a promising path forward for tackling a broader array of complex reasoning tasks. Future investigations could extend this insight into domains where current LLMs struggle, such as abstract problem-solving in mathematics or logical deduction in law.

Chapter 5: CodeTree: Agent-guided tree search for code generation with large language models

Sections 3 and 4 show that LLMs are more reliable when reasoning in natural language and implementing from verbal descriptions than when solving CP problems directly. This chapter investigates whether this separation can be used in a zero-shot agentic setting, addressing two questions. **RQ3: whether reasoning and implementation can be separated throughout the pipeline**, and **RQ4: how our agentic framework can systematically explore the solution searching space**.

5.1 Overview

Recently, we have witnessed significant impacts of large language models (LLMs) beyond the NLP domain, such as in coding tasks (OpenAI, 2023b; Touvron et al., 2023; Wang et al., 2023c; Rozière et al., 2023). However, different from traditional NLP tasks, coding tasks require generated code to be fully executable and functionally correct, i.e., containing no programmatic syntax errors and passing all possible test cases (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021). Given the extremely large search space in code, early methods propose to sample a very large number of generation outputs (for example, Li et al. (2022) generated up to 1 million samples per problem) to increase the chance of generating a correct code solution.

More recently, several approaches adopted a "vertical" strategy in which LLMs first generate one (or very few) generation output, and then iteratively refine this output multiple times, often conditioned by some forms of external feedback (Le et al., 2022; Chen et al., 2022a; Shinn et al., 2023). While these approaches are more cost-effective by focusing only on a small subset of the search space (i.e. starting from an initial output candidate), the performances of these approaches are bounded

Approach	Explore	Exploit	Execution feedback	AI feedback	Multi-agent	Action
CodeRanker (Inala et al., 2022)	✓			✓		
AlphaCode (Li et al., 2022), MBR-Exec (Shi et al., 2022), CodeT (Chen et al., 2022a)	✓		✓			
LEVER (Ni et al., 2023), Coder-Reviewer (Zhang et al., 2023d)	✓		✓	✓		
Self-correct (Welleck et al., 2023), ILF (Chen et al., 2023a), Self-refine (Madaan et al., 2023)		✓		✓		
CodeChain (Le et al., 2024)		✓	✓			
Self-debug (Chen et al., 2023c), Self-repair (Olausson et al., 2023a), Reflexion (Shinn et al., 2023)		✓	✓	✓		
CAMEL (Li et al., 2023a)				✓	✓	
ChatDev (Qian et al., 2024), MetaGPT (Hong et al., 2023), AgentVerse (Chen et al., 2023b)			✓	✓	✓	
Self-collaboration (Dong et al., 2023), AgentCoder (Huang et al., 2023a)		✓	✓	✓	✓	
CodeTree (ours)	✓	✓	✓	✓	✓	✓

Table 5.1: We compare CodeTree with related methods in 6 aspects: (1) *Explore* which adopts a brute-force approach to independently generate a large number of code candidates; (2) *Exploit* which focuses on self-refinement using a small subset of output solutions; (3) *Execution feedback* which uses code execution outcomes to improve code qualities; (4) *AI feedback* which enables synthetic feedback generated by LLMs to improve output code; (5) *Multi-agent* which adopts multiple LLM agents to play different roles in the code generation process; and (6) *Action* where LLM agents can take different actions and facilitate decision-making.

by the local optima of the chosen search space. To address this problem, we propose CodeTree, a framework for LLM agents to efficiently explore the search space in different stages of the code generation process.

Related to our work, several methods for NLP reasoning tasks were introduced to control and enhance the generation procedure of LLMs. For example, Wang et al. (2023a) proposed to enhance LLMs with chains of thought and statistically select the right solutions based on majority voting. Zhou et al. (2023) decomposed a task into smaller sub-tasks and addressed them by increasing the order of difficulty. There is growing interest in agentic approaches to code generation (Shinn et al., 2023; Wang et al., 2023a). Instead of producing one-shot outputs, LLM agents can generate candidate programs, execute them, reflect on the results, and iteratively refine their solutions. Such frameworks address the brittleness of single-pass generation and move closer to the problem-solving process used by human programmers. However,

designing agent systems that can efficiently explore large spaces of potential solutions remains challenging, especially for competitive programming where the number of possible strategies is vast. Yao et al. (2024) proposed to improve LLMs by adopting a tree-based structure to explicitly simulate the exploration of thoughts in a tree. We are motivated by this line of research and proposed CodeTree, a new generation framework to effectively explore the search space of code generation tasks through a tree-based structure. Specifically, we adopted a unified tree structure to explicitly explore different coding strategies, generate corresponding coding solutions, and subsequently refine the solutions. In each stage, critical decision-making (ranking, termination, expanding) of the exploration process is guided by both the environmental execution-based feedback and LLM-agent-generated feedback. We comprehensively evaluated CodeTree on 7 code generation benchmarks and demonstrated the significant performance gains of CodeTree against strong baselines.

Using GPT-4o as the base model, we consistently achieved top results of 95.1% on HumanEval, 98.7% on MBPP, and 43.0% on CodeContests. On the challenging SWEBench benchmark, our approach led to significant performance gains. Our maximum budget experiment beats the then-SoTA (i.e., OpenAI o1-preview w/ long reasoning chain) with GPT-4o on HumanEval, MBPP, and CodeContests with fewer total tokens generated on average.

5.2 Method

CodeTree follows a tree-expansion paradigm to explore the solution-searching space while interacting with the environments and agents. In each stage, critical decision-making (ranking, termination, expanding) of the exploration process is guided by both the environmental execution-based feedback and LLM-agent-generated feedback.

We define 3 standard agents, Thinker, Solver, and Debugger, to equip the strategy-planning, solution implementation, and solution improvement, respectively,

posing comprehensive roles needed for code generation. A CodeTree starts from the input problem as the tree root and subsequent nodes represent code solutions. At any node of the tree, one can either explore sibling nodes (other strategies from the same parent node) or its children (refinements of this node). Within CodeTree, agents can interact with each other through a tree expansion guided by a Critic Agent, searching for the optimal code solution.

Please refer to Figure 5.1 for an overview of our method and Figure 5.2 for a simplified version of instruction prompts to our LLM agents.

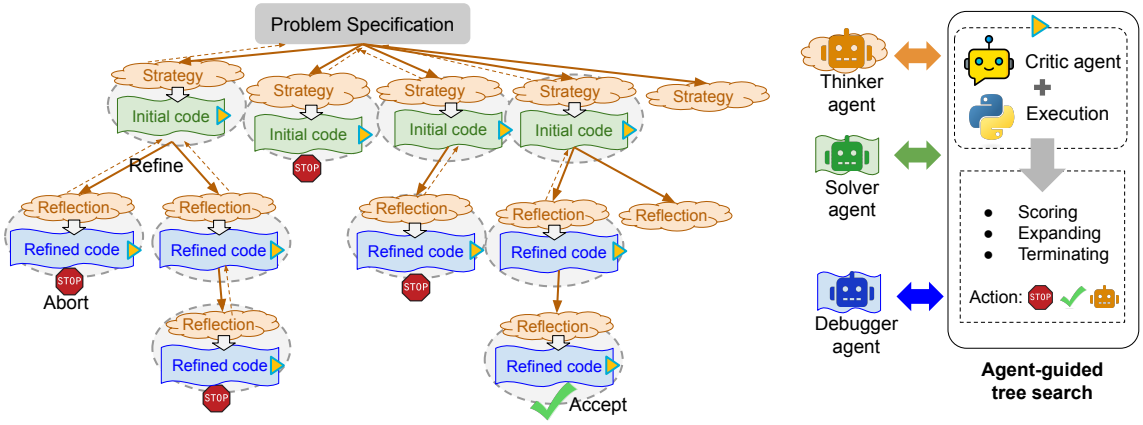


Figure 5.1: CodeTree creates a unified search space for exploration throughout the multi-stage code generation process: strategy generation by a "Thinker" agent, initial code generation by a "Solver" agent, and code improvement by a "Debugger" agent. To effectively perform exploration within the tree structure, we incorporate both environmental execution-based feedback as well as AI-generated feedback (generated by a "Critic" LLM agent).

We first introduce three unit LLM agents, specifically targeting different parts of the code generation process, including strategy thinking, code implementation, and code debugging.

Strategy generation with Thinker agent. Following the setting in Yao et al. (2024), requesting LLMs to generate a list of different natural language *thoughts* can enhance the diversity of solutions. We propose to adapt this technique to allow an

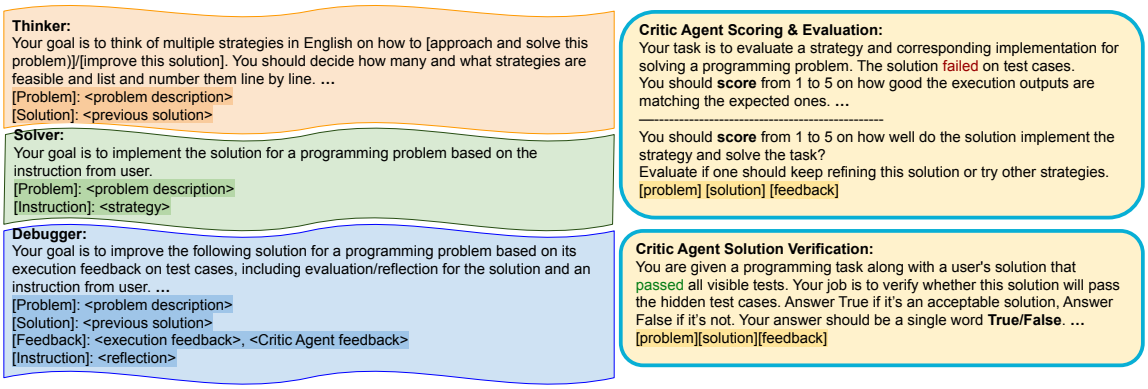


Figure 5.2: Simplified versions of instruction prompts used for Thinker, Solver, Debugger, and Critic agents. Some details are omitted for illustration purposes.

LLM "Thinker" agent to sequentially generate a set of high-level strategies given an input coding problem. Each strategy is generated autoregressively conditioned on previously generated strategies. By allowing models to first generate coding strategies, we enable LLMs to tackle coding problems using their reasoning capabilities learned from the text domain, which answers **RQ3: We can always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline.** The expressiveness of generated strategies in natural language can potentially guide the code-generation process toward more diverse exploration paths. Notably, we let Thinker Agent dynamically decide the number of generated coding strategies, given the fact that different coding problems can have more or fewer feasible strategies.

Solution generation with Solver agent. Given a complete generated strategy from Thinker Agent, we let a "Solver" agent LLM implement a set of initial code solutions. By including the strategy as part of the input instruction, we can condition Solver Agent to produce strategy-specific code candidates. From our previous research in Chapter 3, (Li et al., 2023b; Li and Mooney, 2024), LLMs exhibit abilities to implement solutions based on verbal descriptions of solutions.

Solution refining with Thinker & Debugger agents. Prior approaches such as (Chen et al., 2023a,c; Shinn et al., 2023; Madaan et al., 2023) found that syntactic mistakes or even logical flaws in generated code can be fixed by allowing LLMs to iteratively refine and regenerate the code. This self-refinement capability is typically strengthened by some forms of feedback about the code qualities (e.g. execution results, compiler signals): We name two types of feedback F_i, 1. execution results of the visible test set with the compiler's output and expected output. 2. Self-critique given by the critic agent.

Node evaluation with Critic Agent. CodeTree builds a heterogeneous tree for each problem, where the tree root represents a problem specification $(D, \{(i_j, o_j)\})$ and every subsequent tree node represents a generated candidate code solution Each node has relevant attributes including its collective code feedback F_i and its corresponding strategy and reflections: S_i and R_i. Typically, adding a tree node is a two-step process: 1) generate a code solution from the corresponding strategy, 2) evaluate the generated solution and obtain execution feedback. For a given solution and corresponding F_{exe}, Critic Agent performs an evaluation, measuring how promising it is, which results in F_{cri}. We separately evaluate how well: 1) the execution outputs of test cases match expected outputs on visible test cases; and 2) the solution robustly implements its corresponding strategy towards problem-solving. For one program and its corresponding feedback F_i, the Critic Agent will evaluate whether the current solution is worth refining, or it should not be explored, making decision between refinement and abort. For one solution that passes all visible test cases, it might potentially over-fit the visible test cases and could fail hidden test cases. Hence, the critic agent will verify if this solution is robust and generalizable to unseen test cases.

Tree Expansion with Critic Agent. Unlike previous tree-structure search methods (Yao et al., 2024; Islam et al., 2024), we do not predefine the entire tree with fixed width and depth. Instead, we introduce a Critic Agent to dynamically expand

the tree based on potential strategies. It will guide the expansion and spanning of the tree, taking actions based on its evaluation of the current node. To guide the search for a correct solution, at each node, Critic Agent has an action space of three actions: **Refine**: Continue exploring from the current node by generating multiple reflections for this node; **Abort**: Prune this node due to its low evaluation score, and retrace the exploration to its sibling nodes; and **Accept**: Accept the current node as the final solution and terminate the search early.

Agent collaboration. Throughout the expansion of the tree, the task-specific agents collaborate with Critic Agent, utilizing its feedback, and follow its guidance to perform exploration. The flexibility of the tree expansion and search is determined by LLM agents' decision-making, e.g., determining the number of strategies and deciding the search path. During inference time, practically, we limit the number of exploration steps to avoid large computation overhead. Whenever a termination signal (i.e. to accept a code solution) is found or the maximum number of exploration steps is reached, a code candidate is selected based on its evaluation score.

5.3 Experiments

In this section, we aim to evaluate if CodeTree can efficiently search for the correct solution and which agent is playing the most crucial role.

5.3.1 Experimental Settings

Evaluation metrics and parameters. We applied pass@1 (Chen et al., 2021) as our evaluation metric: only one code candidate will be eventually selected and submitted for the final evaluation with hidden test cases. We set the generation budget to be 20 samples per coding task. To fairly compare our approach with other baselines, we adopted the same generation budget in all methods. For ablation experiments without using the Critic Agent, we followed similar strategies from (Shinn

et al., 2023; Chen et al., 2023c): we select a solution that passes all visible test cases as the final solution to be evaluated with hidden test cases.

Benchmarks. We conducted experiments on 2 categories of code generation tasks: 1) Function implementation where a coding task is to complete a single function following a specific function signature: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and their EvalPlus variants from (Liu et al., 2023), denoted as HumanEval+ and MBPP+; and 2) Program implementation where a coding task is to solve an algorithmic problem: CodeContests (Li et al., 2022) and APPS (Hendrycks et al., 2021). The sizes of test set are 164, 378 and 165 for HumanEval (+), MBPP (+) and CodeContests respectively. For APPS, we randomly sampled 150 samples (50 for each level of difficulty) from the test split.

Baselines. We introduce the following baselines: **Direct** instructs the model to generate code directly from the input problem; **CoT** (Wei et al., 2022b) instructs the model to provide chain-of-thought reasoning before giving the solution program; **Reflexion** (Shinn et al., 2023) utilizes solution's execution feedback to generate self-reflections. The reflections are used to iteratively refine the solution; **MapCoder** (Islam et al., 2024) proposes an agent collaboration system to plan, solve, test, and refine the solution. We set #plans=4, #debug-round=5 and generation budget=20; and **Resample** follows a similar principle as Li et al. (2022): resample solutions repeatedly and filter them with visible test cases.[1]

Models. We studied our method on three models with different model sizes and capacities. We experimented on large language models from the GPT and Llama 3.1

[1]We set sampling temperature=1 for Resample, and report the best results over 2 runs. For other methods, we report the single run's results with deterministic inference.

Model	Method	HumanEval	HumanE+	MBPP	MBPP+	Codecontests	Avg.
GPT-4o-mini	Direct	86.6%	78.7%	87.8%	73.3%	10.3%	67.3%
	CoT	84.8%	78.0%	89.2%	74.3%	12.7%	67.8%
	Reflexion	92.1%	83.5%	96.6%	**78.6%**	21.8%	74.5%
	MapCoder	91.5%	78.0%	90.0%	-	-	-
	Resample	89.0%	80.5%	94.3%	76.8%	18.2%	71.8%
	CodeTree-BFS	93.3%	82.1%	91.5%	72.3%	20.6%	72.0%
	CodeTree-DFS	92.7%	81.1%	87.6%	71.4%	20.6%	70.7%
	Strategy List	90.2%	80.5%	90.5%	69.6%	22.4%	70.6%
	CodeTree	**94.5%**	**84.8%**	**96.8%**	77.0%	**26.4%**	**75.9%**
GPT-4o	Direct	88.4%	81.7%	92.3%	75.9%	20.6%	71.8%
	CoT	92.1%	84.1%	93.7%	77.2%	24.8%	74.4%
	Reflexion	94.5%	84.8%	97.9%	79.6%	41.8%	79.7%
	MapCoder	92.7%	81.7%	90.9%	-	-	-
	Resample	93.9%	84.8%	96.2%	77.0%	32.7%	76.9%
	CodeTree-BFS	94.5%	84.1%	93.9%	70.7%	35.8%	75.8%
	CodeTree-DFS	**95.1%**	83.5%	91.5%	76.2%	36.4%	76.5%
	Strategy List	**95.1%**	82.3%	92.6%	73.3%	36.4%	75.9%
	CodeTree	94.5%	**86.0%**	**98.7%**	**80.7%**	**43.0%**	**80.6%**
Llama-3.1-8B	Direct	63.4%	54.3%	73.4%	63.8%	6.1%	52.2%
	CoT	65.9%	56.1%	74.6%	65.3%	4.2%	53.2%
	Reflexion	79.9%	69.5%	90.2%	72.0%	13.5%	65.0%
	Resample	**82.3%**	71.3%	**91.0%**	**73.8%**	**15.2%**	**66.7%**
	CodeTree-BFS	80.5%	68.3%	91.0%	69.3%	15.8%	65.0%
	CodeTree-DFS	80.5%	68.9%	89.7%	70.4%	15.2%	64.9%
	Strategy List	82.3%	70.1%	**91.0%**	72.5%	13.9%	66.0%
	CodeTree	82.3%	**72.0%**	90.5%	73.3%	12.1%	66.0%

Table 5.2: Experimental results by pass@1 on HumanEval, MBPP, EvalPlus, and CodeContests: methods are baseline methods that generate program solution only once, are methods with solution generation budget of 20 samples like our methods. are CodeTree variants with or without Critic Agent to guide the tree search. Note that MapCoder does not work with Llama-3.1-8B as noted by Islam et al. (2024).

family. Specifically we use **GPT-4o-mini**, **GPT-4o**[2], and **Llama-3.1-8B** [3]

5.3.2 Main Results

We compared CodeTree with other baselines in Table 5.2. We noticed that Reflexion and Resampling serve as strong baselines for HumanEval and MBPP datasets

[2]`https://openai.com/index/hello-gpt-4o/`

[3]`https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct`. Note that we reported our replicated results which might be different from the original reported ones.

Model	GPT-4o-mini		GPT-4o
Benchmark	HumanEval	HumanEval+	CodeContests
CodeTree-BFS			
$d = 3, w = 3$	93.3%	82.3%	36.4%
$d = 2, w = 4$	**95.1%**	**84.1%**	37.6%
$d = 2, w = 5$	94.5%	83.4%	**39.4%**
CodeTree-DFS			
$d = 3, w = 3$	92.7%	81.1%	36.4%
$d = 4, w = 2$	92.1%	81.1%	37.0%
$d = 5, w = 2$	92.1%	81.7%	36.4%

Table 5.3: Pass@1 results of CodeTree-BFS/DFS on HumanEval, HumanEval+, and CodeContests: d indicates max search depth while b indicates max search width.

given the same solution generation budget, comparable to CodeTree-BFS/DFS. CodeTree with Critic Agent outperforms all other baselines in 4 out of 5 benchmarks for GPT-4o-mini and GPT-4o. For instance, CodeTree achieves pass@1=43.0% on competition-level coding tasks in the Codecontests benchmark (i.e. 10.3% absolute performance gain over the Resampling baseline), showing its advantage in solving hard problems.

We found that CodeTree-BFS almost always performs better than CodeTree-DFS, suggesting that exploring diverse strategies is more effective than iteratively refining from one solution. Interestingly, on the Llama-3.1-8B model, Resampling achieves the best results on 4 benchmarks. This observation indicates that small language models may not be suitable for multi-agent frameworks like CodeTree, where models are required to follow task-specific roles and instructions and perform distinct tasks with reasonable accuracy.

understand and follow complex instructions, or to perform distinct tasks for agents, and thus behave unexpectedly.

5.3.3 Analysis of Search Strategies

Given the performance gaps between CodeTree-BFS and DFS, we conducted additional experiments to analyze these tree search strategies without Critic Agent.

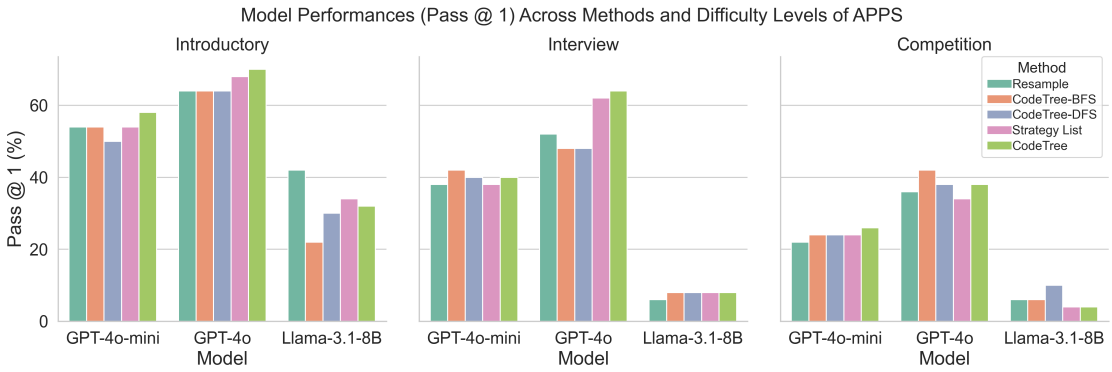


Figure 5.3: Results of pass@1 on the APPS test subsets. We randomly select 50 samples from Introductory, Interview, and Competition separately. We apply Resample and our methods with GPT-4o, GPT-4o-mini, Llama-3.1-8B.

We reported the results on HumanEval/HumanEval+ with GPT-4o-mini and Codecontests with GPT-4o in Table 5.3. Compared to DFS/BFS strategies with $d = 3$ and $w = 3$, we observed that forcing the model to search wider (i.e. more diverse strategies with $w > 3$) in BFS and only debug up to 1 iteration (i.e. $d = 2$) improved the performance of pass@1. However, for DFS, prioritizing deeper search (i.e. larger number of iterations for refinement with $d > 3$) does not boost the performance significantly. Finally, we noted that $w = 4$ works better for HumanEval and $w = 5$ works better for CodeContests, indicating that more complex problems can benefit from exploring a larger number of coding strategies. This finding supports our proposed CodeTree in which Critic Agent can dynamically determine the number of child nodes to explore given the context.

5.3.4 Analysis by Problem Difficulty

We further evaluated CodeTree against coding problems with diverse difficulty levels. We randomly sampled 50 problems from each level of difficulty ("introductory", "interview", and "competition") from the test split of the APPS benchmark, creating a total test set of 150 problems. We reported the results in Figure 5.3.

The results demonstrate that CodeTree performs better for simpler problems

Model	**GPT-4o-mini**	
Benchmark	HumanEval	HumanEval+
CodeTree	94.5%	84.8%
w/o verification	91.5%	81.7%
w/o node abort	91.5%	81.1%
w/o scoring	92.7%	82.1%

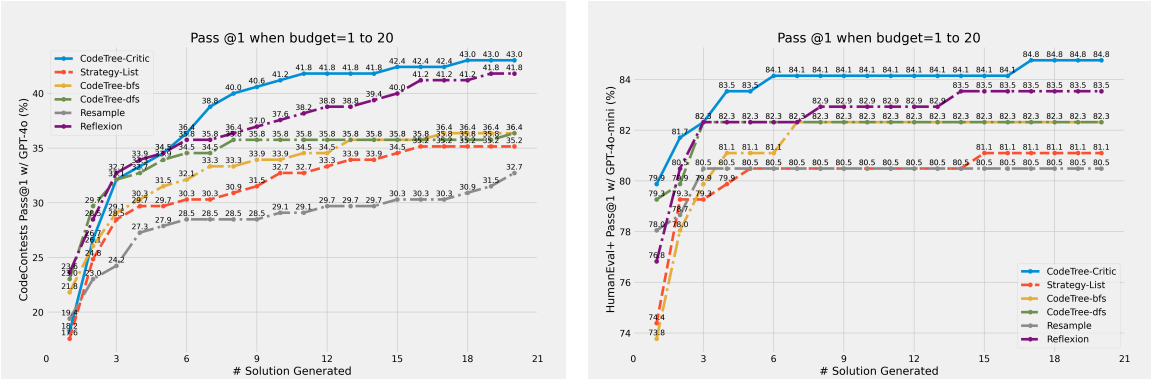
Table 5.4: Ablation study for different tasks of Critic Agent. We used GPT-4o-mini to evaluate corresponding settings and reported the pass@1 on the HumanEval and HumanEval+ benchmarks.

(Introductory for GPT-4o-mini; Introductory and Interview for GPT-4o), but still struggles to solve very hard problems (e.g., Competition-level). We hypothesized that while CodeTree improves the search efficiency towards the correct solution, a generation budget= 20 is still limited for highly complex problems.

5.3.5 Ablation Study

Results in Table 5.2 indicate, Critic Agent plays a crucial role in CodeTree over naive search methods like BFS/DFS. We further analyzed which task is the most crucial for Critic Agent. Specifically, we conducted the following ablation experiments: (1) w/o Solution Verification, where we excluded the verification task for any solution passing visible tests; (2) w/o Node Abort Evaluation, where we let the agents keep exploring till we reach the max depth or whenever a solution is accepted; (3) w/o Node Scoring, where the environmental feedback is solely execution outputs, without Critic Agent's evaluation.

The results in Table 5.4 show that all proposed tasks are crucial for Critic Agent to guide tree expansion and solution search. Among these tasks, node abort and solution verification tasks are the most effective and have significant impacts on final performance.



(a) Performance on CodeContests with GPT-4o

(b) Performance on HumanEval+ with GPT-4o-mini

Figure 5.4: CodeTree: Cumulative pass@1 curves while new solutions are generated within budget= 20.

5.3.6 Search Efficiency

While the performance of CodeTree is very strong with a generation budget of 20 samples, it is important to understand how our tree search strategy are more efficient than other related methods. Specifically, we conducted experiments when limiting the solution generation budget between 1 to 20 samples. In Figure 5.4, we plot the pass@1 curves w.r.t the number of sampled solutions: (a) shows results on CodeContests with GPT-4o; (b) shows results on HumanEval+ with GPT-4o-mini. In both figures, not only does our proposed CodeTree perform better than other methods, but it also achieves a relatively high pass@1 even when the generation budget is small (e.g., < 9 samples). On CodeContests, even when CodeTree starts with a lower performance with limited generation budgets (i.e., < 5 samples), its performance soon improves significantly during later exploration of the tree. This observation shows that CodeTree has the potential to continue solving more problems given larger solution generation budgets.

To better demonstrate the efficiency of our method, we carried out experiments with a large budget = 30 and compared the cost and token usage against the

Model	MBPP+	HumanEval+	Codecontests
Resample	77.0% (0.3k)	84.8% (0.4k)	41.2% (2.4k)
Reflexion	79.6% (0.5k)	85.4% (0.7k)	46.1% (6.0k)
CodeTree	**82.8%** (0.7k)	**89.6%** (0.9k)	**49.1%** (4.8k)
O1-preview	80.2% (1.0k)	89.0% (1.0k)	46.6% (7.1k)

Table 5.5: Efficiency Comparison: Resample, Reflexion and CodeTree performances when budget=30 w/GPT-4o. % denotes pass@1 while () denotes average inference tokens (thousands) generated per problem.

art reasoning model: OpenAI's o1-preview model[4]. In Table 5.5, given the budget of 30, the flexibility of CodeTree can avoid redundant search when solving simpler problems while exhaustively exploring strategies for harder problems. CodeTree is able to outperform O1-preview with fewer tokens and less cost, showing its efficiency in solution search for programming problems.

5.3.7 Results on SWEBench

Recently, Jimenez et al. (2024) introduced a novel coding task for generating code patches given an input Github repository and the corresponding pull request. We attempted to adapt CodeTree to this benchmark by making two major changes: first, we replaced the problem specification with the textual description of input pull request; secondly, we adapted the strategy generation stage as the retrieval stage where we instructed the Thinker agent to explore relevant code contexts (by code files, methods, or lines of code) from the input Github repository. We extended the implementation of the retrieval and code patch generation stages from Xia et al. (2024) and integrated our CodeTree framework for the exploration of different trajectories (from context retrieval to code patch generation and ranking). From Table 5.6, we observed that CodeTree can lead to significant performance gains as compared to related approaches like CoT (Wei et al., 2022b) and Reflexion (Shinn et al., 2023). The results also demonstrate the versatility of our method on complex coding tasks

[4]`https://openai.com/index/introducing-openai-o1-preview/`

Approach	% Resolved
(Xia et al., 2024)	24.2%
(Xia et al., 2024) + CoT	23.6%
(Xia et al., 2024) + Reflexion	25.3%
(Xia et al., 2024) + CodeTree	**27.6%**

Table 5.6: Results on the SWEBench benchmark: All methods using GPT4o-mini as the base model. Compared to CoT and Reflexion, CodeTree can lead to more significant performance gain.

like repo-level code generation which often requires extremely large search space to find an optimal solution.

5.3.8 Conclusion

We introduce CodeTree, a new framework of agent-guided tree search for code generation tasks. By introducing a tree-based structure as a unified search space, CodeTree includes a Critic agent to guide the tree search and make critical decisions such as termination, expansion, and scoring of tree nodes. Following the findings from our previous works, we intentionally separate strategy exploration, solution implementation, to gain additional diversity in the solution search space. CodeTree facilitates multi-agent collaboration (among Thinker, Solver, and Debugger agents) to find the correct solution within a limited solution generation budget. **This answers our RQ3: Can we always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline?**

Results suggest that separating strategy generation from implementation can improve search efficiency when the base model is strong enough to follow role-specific instructions. However, the benefit is not universal: smaller models may lose performance because they cannot reliably perform the distinct agent roles.

In this work, we leverage LLMs' natural language reasoning capabilities in a zero-shot manner. This is largely reflected in Thinker Agent who gives distinct

problem-solving or code-fixing strategies, and Critic agent, who scores, verifies and acts on a node of a candidate solution.

Hereby, we can answer the second research question in the first section of this chapter: **RQ4: How can we build an agent that systematically explores this decomposed reasoning space?** In CodeTree, natural-language reasoning is used primarily by the Thinker, which proposes problem-solving and debugging strategies, and by the Critic, which scores, verifies, and chooses actions for candidate nodes. The system answers RQ4 by combining role-constrained agents, separated contexts for efficiency, and a search policy controlled by both deterministic signals—such as execution feedback and scores—and probabilistic Critic judgments.

Chapter 6: AlgoSimBench: Identifying Algorithmically Similar Problems for Competitive Programming

Chapters 3 and 4 show that natural-language reasoning can improve CP solving, either through distilled explanations or through agentic search. However, improved solve rates do not tell us whether the model has learned reusable algorithmic structure or has merely become better at generating and verifying candidate programs. This motivates a diagnostic shift: instead of asking whether a model can solve a single problem, we ask whether it can recognize when two different problems require the same underlying algorithmic idea.

Having studied methods that improve CP solve rates, we next test whether such problem-solving ability corresponds to transferable algorithmic understanding. We operationalize this question through algorithmically similar problem identification: if a model understands the underlying strategy of a problem, it should be able to recognize another problem solved by the same strategy even when the surface stories differ.

6.1 Overview

Recent reasoning-enhanced Large Language Models (LLMs) have achieved promising results in solving complex competitive programming problems. However, it remains unclear whether these reasoning abilities generalize to relevant tasks, like identifying algorithmically similar problems (ASPs). We introduce AlgoSimBench, a benchmark of 402 multiple-choice questions curated in an adversarial setting: each given reference problem is paired with one algorithmically similar problem and three distractors that are semantically close but algorithmically dissimilar. This design forces models to rely on algorithmic reasoning rather than superficial textual cues.

Our evaluation shows that LLMs consistently struggle under this setting. To address this gap, we propose Attempted Solution Matching (ASM), which leverages LLM-generated solution attempts to assess similarity, yielding an average accuracy improvement of 9% across models. Beyond LLM evaluation, AlgoSimBench also probes code retrieval methods; when combined with BM25 (Trotman et al., 2014), ASM achieves an additional 11.8% gain over state-of-the-art embedding models. AlgoSimBench offers a challenging testbed that facilitates future studies on LLMs and retrieval methods.

6.2 Introduction

Large Language Models (LLMs) trained on both natural language and code have shown remarkable capabilities in automating various aspects of code generation and software engineering (Chen et al., 2021; Jimenez et al., 2024). Although solving competitive programming (CP) problems has become a popular and challenging benchmark to evaluate LLMs' code reasoning abilities, it remains largely focused on code generation (Li et al., 2022; Shi et al., 2024). This raises an open question: can models trained on problem-solving generalize their reasoning to more abstract or alternative forms of algorithmic understanding?

Recent works have highlighted several limitations of LLMs' code reasoning abilities. Gu et al. (2024) find that LLMs often struggle to predict program outputs given specific inputs. Similarly, Wei et al. (2025) shows that LLMs struggle to determine whether two programs are functionally equivalent. In this work, we propose to study an underexplored but crucial subskill that underlies many forms of code reasoning: the ability to identify algorithmically similar problems.

In the context of programming and problem-solving, "algorithmically similar problems" (ASPs) are those that—despite differences in context, surface wording, or background story—can be solved using similar algorithmic strategies (Figure 6.1). This notion of similarity goes beyond superficial textual resemblance; it lies in the

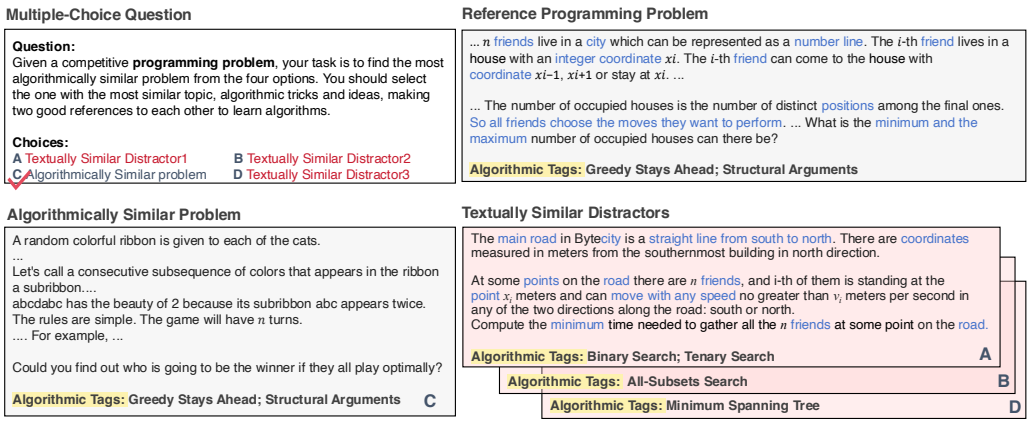


Figure 6.1: An example from AlgoSimBench, with one algorithmically similar problem and a textually similar distractor as an example. The blue-highlighted text makes the reference problem and the distractor look similar.

shared algorithmic structure, solution path, and problem-solving "tricks" required.

To curate AlgoSimBench, we collect and filter human-expert-annotated CP problems with fine-grained labels from 4 CP websites, from which we develop a set of 402 multiple-choice questions (MCQs). Each MCQ requires identifying an ASP for a reference problem from 4 options: one ASP and 3 algorithmically-dissimilar but textually similar distractors.

We intentionally design the task to challenge superficial textual similarity analysis, emphasizing true structural algorithmic reasoning. Inspired by adversarial benchmarks such as Winograd and WinoGrande (Levesque et al., 2011; Sakaguchi et al., 2019), AlgoSimBench deliberately inverts the usual correlation between surface semantics and problem similarity in terms of solving. By making distractors highly semantically similar yet algorithmically different, we enforce that success does not come from shallow lexical overlap but requires deep reasoning about problem-solving and algorithmic properties.

AlgoSimBench can be applied to evaluate LLMs in both an end-to-end selection setting and under a retrieval-based setting (see Sec. 6.5.1). AlgoSimBench can encourage the development of better LLMs to reason independently of superficial tex-

tual features, as well as better embedding models to represent entailed logical features in text.

Our initial study shows that program solutions are better for identifying ASPs than natural-language problem descriptions. This motivates our proposed method: attempted solution matching (ASM), in which LLMs first generate an attempted natural-language or programming-language solution for each candidate problem, and then these solutions are compared instead of problem statements to identify ASPs. ASM leverages attempted solutions as a bridge to problem similarity, yielding robust gains on AlgoSimBench for both End-to-End and Retrieval-Based selection.

Finally, we found that ASM can help improve the selection of exemplars for In-Context Learning (ICL) in the context of competitive programming. Unlike previous methods (Xiong et al., 2026; Shi et al., 2022), ASM does not require any information from human-generated oracle solutions, and yet achieves comparable performance.

6.3 AlgoSimBench Dataset

AlgoSimBench is a challenging benchmark, designed to evaluate LLMs' algorithmic-reasoning ability beyond code generation, code embedding and code retrieval. It is collected from competitive programming problems, written, solved, labeled, and categorized by human experts. We collect problems and expert annotations from competitive programming communities, constructing 402 MCQs for ASP identification: Each question contains a given reference problem and 4 options, one algorithmically similar problem and three algorithmically dissimilar distractors. In this section, we introduce how we collect the labeled problems and the method used to construct adversarial MCQs.

6.3.1 Data Sources and Statistics

Many competitive programming platforms provide algorithm tags, but these are usually too general (e.g., dynamic programming) to fully capture the solution

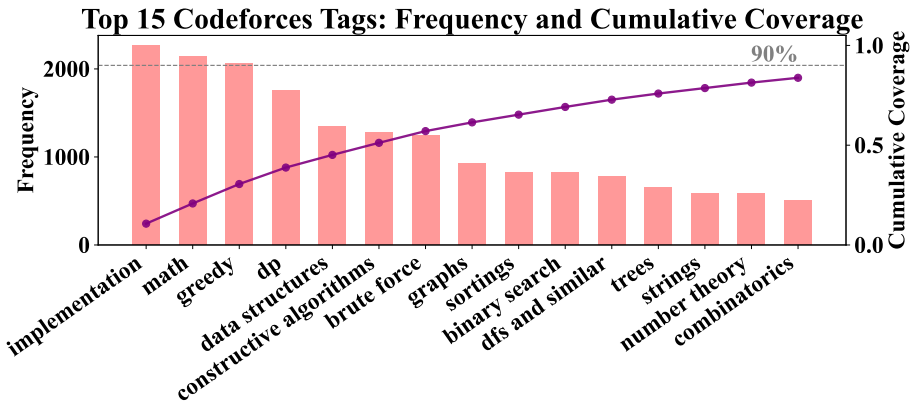
strategy. To identify ASPs, we need more specific tags, such as Longest Increasing Subsequence or 0/1 Knapsack, characterizing a specific technique needed to solve the problem. A comparison between the distributions of AlgoSimBench's labels and Codeforces' tags are given in Figure 6.2. From four different CP community websites, we collect problems with corresponding correctness-verified human-written solutions (subsequently called *oracle solutions*) and fine-grained algorithmic labels. We manually unified labels from these different sources and removed duplicates and overly broad or ambiguous tags, resulting in 1,317 cleanly labeled problems. The final 402 MCQs utilized 903 problems and 231 labels from the filtered set.

Our data was collected from four websites for competitive programming participants and learners, namely:

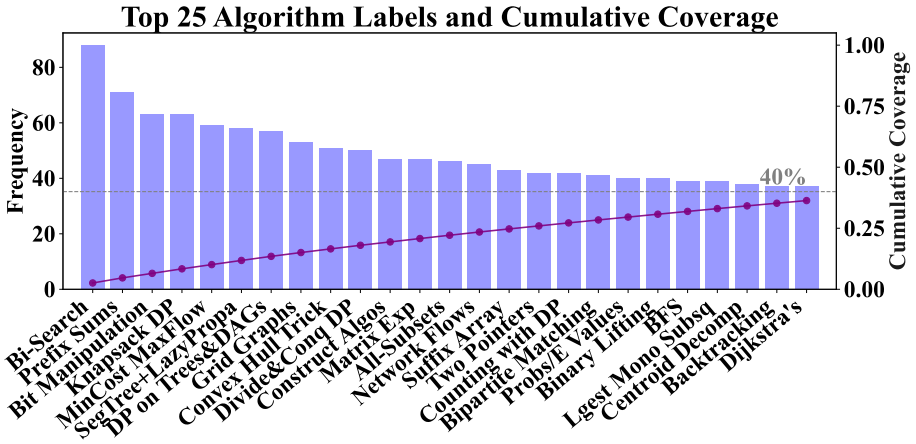
- The Ultimate Topic List: `https://youkn0wwho.academy/topic-list`
- CP Algorithms: `https://cp-algorithms.com/`
- ProgVar: `https://progvar.fun/`
- USACO-Guide `https://usaco.guide/`

1,522 problem descriptions and solutions were collected from the following competitive programming websites: Codeforces (`https://codeforces.com/`), AtCoder(`https://atcoder.jp/`) and CodeChef(`https://www.codechef.com/`).

We filtered out problems with broad, ambiguous, or duplicate annotations, giving a final set of 1,317 problems. Detailed information on these problems is available in Table 6.1. Not all these problems were selected for the final MCQs, some were ignored due to the lack of similar problems or distractors or failing textual similarity thresholds. However, the full set serves as a good source of programming problems for other tasks such as algorithmic tagging.



(a) Top 15 Codeforces broad algorithm tags.



(b) Top 25 fine-grained labels in our dataset.

Figure 6.2: AlgoSimBench: Comparison between Codeforces tag distribution and our fine-grained tag distribution.

	Codeforces	AtCoder	CodeChef	total
# problems(s_i)	1114	182	18	1317
# Tokens/s_i	507	346	468	489
# Solutions/s_i	1	1	1	1
# Tags/s_i	1.12	1.08	1.0	1.11

Table 6.1: Statistics of Programming Problems in AlgoSimBench.

Finally, 903 problems were selected for use in the MCQs, we excluded a portion of problems with high-frequency tags like binary search, to enhance the diversity of the chosen problems. Some statistics on these problems are given in Table 6.2.

	AlgoSimBench MCQ
# questions	402
# fine-grained tags	231
# categorized tags	204
Avg # Tokens/question	2681

Table 6.2: Statistics of MCQs in AlgoSimBench.

6.3.2 Adversarial Task Design

As Jain et al. and Jain et al. (2021); Wei et al. (2025) found, language and code models are sensitive to functionality-preserving attacks and struggle to disentangle program logic from natural language semantics.

Competitive programming problems usually contain a background story, connecting the algorithms to a real-life scenario. We intentionally invert the correlation between algorithmic similarity and semantic textual similarity. That is, algorithmically similar problems should differ as much as possible in wording and surface descriptions, while dissimilar problems should appear similar in text but differ in algorithmic structure. Specifically: The similar option shares the closest match in problem topics and algorithmic approach but differs significantly in text and overall linguistic meaning. The distractors belong to different algorithmic categories but are chosen to be textually close to the original problem. The idea underlying this setting

is that *models should be able to ignore features irrelevant to problem solving and base their judgment of algorithmic similarity solely on problem logic and structure.*

6.3.3 Multiple-Choice Question Construction

We obtain a problem set, S, each with a problem statement, s_i, a corresponding solution program, c_i, and a set of algorithmic labels $Y_i = \{y_1, \cdots, y_k\}$ We create multiple-choice questions as follows: Given a reference CP problem $p \in S$, we find an algorithmically similar problem, $q \in S$, and three algorithmically dissimilar problems $\{d_1, d_2, d_3\} \subset S$.

Algorithmic similarity filter. To ensure algorithmic similarity between the reference problem p and the similar choice q, they should have the exact same set of labels $Y_p = Y_q$. On the other hand, the distractors d_i should have no label overlap with the reference problem p. To further enforce the minimal algorithmic similarity of each d_i, we apply the following conditions to d_i's labels: 1. Any Y_{d_i} does not share label **sub**categories with Y_p; 2. Any Y_{d_i}' is lexically dissimilar with Y_p; We then manually filter and remove edge cases.

Textual similarity filter. Once we select candidates for the correct option as well as distractors, we ensure textual dissimilarity and similarity as follows. For the correct ASP option, q, among all candidates, we select the problem with the lowest textual similarity to p. For the distractors d_i, we select the n options with the highest textual similarity to p, ensuring $Sim(p, d_i) > Sim(p, q)$. We compute textual similarity using a dense embedding model (Lewis et al., 2020a).

6.4 Methods

Given the definition of algorithmic similarity, we hypothesize that it is better reflected in the solution code than in the problem statement — a finding supported

	Statement	Solution
BM25 (Robertson et al., 1995)	12.7	32.6
BART (Lewis et al., 2020a)	13.2	33.3
CodeBert (Feng et al., 2020)	7.0	36.3
GraphCodeBert (Guo et al., 2021)	6.7	35.6
CodeSage-v2 (Zhang et al., 2024)	15.2	39.1
SFR-Code (Liu et al., 2024c)	21.6	39.6
Jina-Code-V2 (Günther et al., 2024)	10.4	42.8
CodeRankEmb (Suresh et al., 2025)	7.7	36.3

Table 6.3: AlgoSimBench MCQ Retrieval Accuracies (%) across different retrieval methods. Cosine similarity is used for all dense models.

by Shi et al. (2024).

6.4.1 Hypothesis: solutions manifest algorithmic features

To test this hypothesis, we apply a direct similarity-based retrieval approach where the goal is to retrieve ASPs (from the options in our MCQ problems) using either problem statements or oracle solution code as the problem representation. We treat the 4 options q, d_1, d_2, d_3 as a mini-corpus and use the reference problem p as the query. Retrieval is successful when q is the highest-ranking option. A variety of dense and sparse retrieval methods were tested using this setting. We use cosine similarity to calculate the similarity scores. As shown in Table 6.3, retrieval based on oracle code solutions consistently achieves higher accuracy, significantly better than random. This supports our hypothesis: Solutions explicitly encode algorithmic features that remain implicit or latent in the problem.

Since oracle solutions are generally not available in real-world use cases, we propose Attempted Solution Matching (ASM)—a method that uses an LLM to generate a solution attempt for each problem, and then compares these attempted solutions to identify ASPs. Figure 6.3 illustrates the approach.

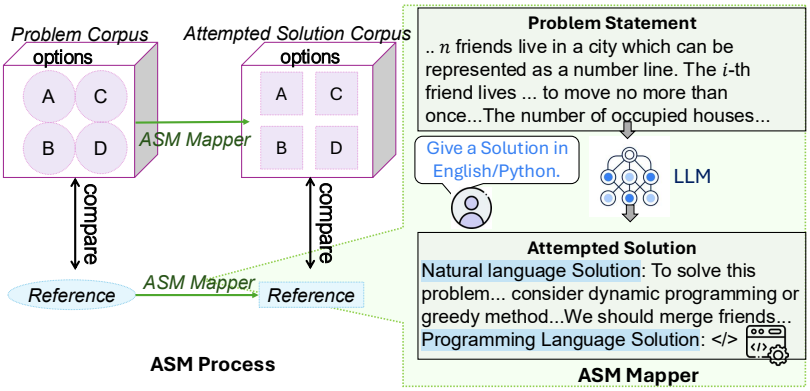


Figure 6.3: Illustration of Attempted Solution Matching. Both the given reference problem (reference) and options (corpus of problems) are mapped to attempted solutions by instructing an LLM to solve the problems.

6.4.2 Attempted Solution Matching

Episodic Retrieval (Shi et al., 2024) uses similarity among solution programs to identify similar problems, but it compares LLM-generated code for the query problem with oracle-solution code for the potential retrieval targets. This assumes that the initial solution contains the correct algorithmic signal. However, this breaks down when the LLM selects the wrong approach, where such errors can accumulate, decreasing LLMs' capacity to identify algorithmically similar problems.

To mitigate this mismatch between initial attempts and oracle solutions, we draw insights from the "generate-to-retrieve" paradigm (Gao et al., 2023) and propose matching first-attempt solutions to other first-attempt solutions. Our method comes with a simple assumption: when attempting to solve a problem, the algorithmic ideas and problem-solving strategies are naturally reflected in both the model's natural language reasoning and its generated code, revealing the underlying methods being applied. As a result, algorithmically similar problems tend to produce similar solution attempts.

In ASM, each problem statement (i.e., problem description) is first mapped to a first-attempt solution using an LLM, and ASPs are identified by evaluating the

similarities between attempted solutions. The attempted solution can take two forms: A natural language explanation of the solution strategy (NL-solution), denoted as ASM-NL; or a solution written in a programming language (PL-solution), ASM-PL. While both should contain algorithmic information, ASM-NL can include content on problem analysis, strategy exploration, algorithm selection, and problem solving; on the other hand, ASM-PL focuses on the complete implementation, whose content is rich in details but lacking in problem analysis.

6.5 Experiments

In this section, we start with evaluating LLMs' abilities to identify ASPs with AlgoSimBench under End-to-End Selection. Experiments are conducted to analyze how different factors affect LLMs' performance. Next, we explore using AlgoSimBench to test the abilities of various retrieval methods under the Retrieval-Based Selection setting. We evaluate several baselines as well as ASM-NL and ASM-PL under both settings. Finally, we apply ASM to in-context learning to retrieve better examples for aiding the generation of solutions to programming problems.

6.5.1 Experimental Settings

Datasets & metrics. AlgoSimBench, consists of 402 MCQs, where we measure the accuracy as the rate of selecting the correct ASP out of 4 options. To evaluate the performance of ICL exemplar selection, we test ASM and various baselines on USACO (Shi et al., 2024), a dataset with 307 "olympiad" competitive programming problems. We calculate the increase in the pass@1 solution rate (Chen et al., 2021) over random exemplar selection for different exemplar retrieval methods.

Model	Statement	Summary	ASM-NL	ASM-PL	Solution*
Qwen3-0.6B-thinking	25.8	26.1	**33.3** (↑ 7.5)	31.3 (↑ 5.5)	30.6
Qwen3-4B-thinking	43.3	41.2	53.0 (↑ 9.7)	**53.9** (↑ 10.6)	59.2
Qwen3-8B-thinking	47.5	46.3	**58.2** (↑ 10.7)	57.0 (↑ 9.5)	62.4
Llama3.1-8B-Instruct	23.9	24.1	**26.4** (↑ 2.5)	24.4 (↑ 0.3)	29.6
GPT-4o-mini	35.6	35.8	**43.8** (↑ 8.3)	42.5 (↑ 7.0)	54.4
GPT-4o	41.5	38.1	**53.2** (↑ 11.7)	53.0 (↑ 11.5)	63.4
o3-mini-medium	65.9	63.4	74.4 (↑ 8.5)	**75.1** (↑ 9.2)	72.6
Deepseek-R1	63.7	57.7	69.2 (↑ 5.5)	**70.4** (↑ 6.7)	70.6
Deepseek-V3	55.2	53.2	**64.9** (↑ 9.7)	62.7 (↑ 7.5)	66.7
Claude-3.5-Sonnet	44.2	45.0	**54.7** (↑ 10.5)	53.0 (↑ 8.8)	70.5
Gemini-2.0-Flash	51.2	48.5	**57.9** (↑ 6.7)	55.5 (↑ 4.3)	69.7
Qwen3-Coder-480B	50.7	51.0	**68.4** (↑17.7)	62.4 (↑11.7)	66.9
Avg	45.1	44.2	**54.8** (↑ 9.7)	53.4 (↑ 8.3)	59.7

Table 6.4: End-to-End Selection Performances (Accuracy%) on AlgoSimBench-MCQ over different models and methods. Results in bold indicate the best performing non-oracle method for each model. Absolute performance gain over the **Statement** baseline is marked with ↑. *Solution* is an oracle-input setting where oracle solutions were directly provided to the model. ASMs are statistically significantly better than Statement/Summary with $p < 0.01$.

Models. For End-to-End Selection, we evaluate 12 open and closed LLMs: Qwen3-0.6B, 4B, 8B and Qwen3-Coder-480B [1]. Llama-3.1-8B-instruct[2], GPT-4o-mini, GPT-4o[3], o3-mini-medium[4], Deepseek-R1 (DeepSeek-AI et al., 2025a), Deepseek-V3 (DeepSeek-AI et al., 2025b), Claude-3.5-Sonnet[5] and Gemini 2.0 Flash.[6] For Retrieval-Based Selection, we tested various metrics for measuring the similarity between the query and candidate answers. We used a sparse retriever, BM25 (Robertson et al., 1995), and cosine similarity of dense embeddings from a text embedding model BART (Lewis et al., 2020a), code embedding models GraphCodeBert (Guo et al., 2021), CodeSage-v2

[1] `https://huggingface.co/collections/Qwen/qwen3`
[2] `https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct`
[3] `https://openai.com/index/hello-gpt-4o/`
[4] `https://openai.com/index/openai-o3-mini/`
[5] `https://www.anthropic.com/news/claude-3-5-sonnet`
[6] `https://deepmind.google/technologies/gemini/flash/`

(Zhang et al., 2024), SFR-Code-400M_R (Liu et al., 2024c), Jina-Code-V2 (Günther et al., 2024), CodeRank Embedding (Suresh et al., 2025).

Baselines. We compare ASM to using the following baseline problem representations. **Statement**: Original natural-language descriptions are used to represent problems, **Summary**: To mitigate aspects unrelated to problem-solving, an LLM is first prompted to summarize and abstract the problem, minimizing narrative elements and formatting details irrelevant to the structure of the problem. **Solution:** Each problem is replaced with a correct solution written by an expert programmer, which serves as an "oracle" upper baseline.

ASM settings. As discussed before, for Attempted Solution Matching, we have a *Natural Language* setting, **ASM-NL**, where LLMs are asked to describe the solution in English given the problem, which is later used as reference and options; a *Programming Language* setting **ASM-PL**, where LLMs are asked to first generate Python-implemented solutions for the reference and options.

Evaluation settings. We evaluate models under two task settings: 1) End-to-End Selection: where a full MCQ is presented to the LLM in a single prompt with a reference problem and four candidate options, and models are directly asked to select the most algorithmically similar problem. 2) Retrieval-Based Selection: where each MCQ is framed as a retrieval task, where the query is the reference problem and the 4 candidates form a corpus of potential retrievals. For both settings, it is considered correct if the retrieved option is the algorithmically similar one.

6.5.2 End-to-End Selection

We apply the same prompts to all End-to-End Selection experiments, with a definition of ASPs and encouragement to use chain-of-thought reasoning. The only difference is to provide problems or attempted solutions as the options.

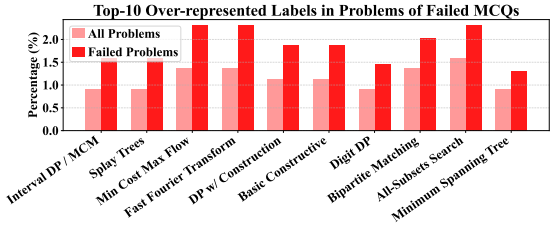
6.5.2.1 Main Results

Table 6.4 reports MCQ accuracy for all twelve models. Under the Statement setting, most models perform poorly, with Llama-3.1-8B near random and reasoning-oriented models such as DeepSeek-R1 and o3-mini achieving the strongest results. Replacing statements with oracle solutions improves performance, indicating that solution representations expose algorithmic structure more clearly. Summaries generally do not help, whereas ASM-NL and ASM-PL consistently improve accuracy, with average gains of 9.7 and 8.3 percentage points over Statement. These gaps suggest that LLMs recognize algorithmic similarity more reliably when prompted to reason through solution attempts than when comparing raw problem statements.

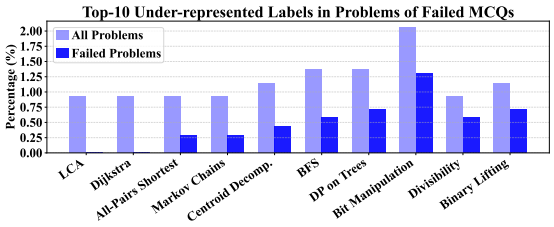
Summarizing problems actually hurts the performance of all models except 2. We hypothesize that SoTA LLMs can already ignore superficial similarities when comparing programming problems. Our methods, **ASM-NL** and **ASM-PL**, consistently improve absolute performances across all models with an average improvement of 9.7% and 8.3%, respectively. **ASM-NL** outperforms **ASM-PL** on 10 out of 12 models, indicating that most models can generate and compare solutions described in natural language better than actual code solutions. For o3-mini-medium, both ASMs even outperform the oracle-solution baseline and give the best performances overall. The performance gaps between **Statement** and **ASM** also suggest that LLMs cannot generalize their code reasoning ability to similar tasks when they are out of a code generation context.

6.5.2.2 Effects of Problem Attributes

We further analyzed failure cases in AlgoSimBench to answer two research questions: 1. Do models make more mistakes on more difficult problems? 2. Are there some algorithms that are especially difficult to identify from problem descriptions?



(a) Top-10 over-represented (greater portion, percentage) labels in failed MCQs compared to all MCQs.



(b) Top-10 under-represented (lower portion, percentage) labels in failed MCQs compared to all MCQs.

Figure 6.4: Comparison of tag distributions between problems in failed and all MCQs. Over/Under-represented is calculated by comparing the label distributions of all problems to those of failed problems.

Difficulty-level. The Pearson correlation between a problem's Codeforces' difficulty rating and the probability that an LLM will get it wrong is **-0.15** with p-value = 0.45, indicating that models do not perform worse on ASP if the problems are harder. This highlights an important distinction: *identifying a problem's category and the appropriate algorithm is not the same as being able to solve it.* For instance, recognizing that a Segment Tree is needed does not necessarily imply the ability to implement a correct solution for range updates and queries. Some problems are easy to *classify* but difficult to *solve*—reinforcing the need to evaluate models beyond final solution correctness.

Algorithm labels. The over-represented and under-represented algorithmic labels characterizing problems that models tend to get wrong (i.e., highest increment and decrement in label proportion) are shown in Figure 6.4. We found that some algorithmic types, though often seen as difficult (e.g., Centroid Decomposition) are not hard to identify. Some other simple problems, like shortest-path/Dijkstra, are also easy to identify. On the other hand, algorithms like DP and Fast-Fourier-Transform can be applied in very different circumstances, making such problems difficult to identify.

In a nutshell, LLMs' ability to completely solve programming problems and their ability to identify relevant algorithmic strategies are two fairly independent axes.

Model	PL pass@1	ASM-PL	Solution
Qwen3-0.6B	1.8	31.3 (↑ 5.5)	30.6
Qwen3-4B	16.0	53.9 (↑ 10.6)	59.2
Qwen3-8B	17.9	57.0 (↑ 9.5)	62.4
GPT-4o-mini	19.6	42.5 (↑ 7.0)	54.4
Claude-3.5-Sonnet	41.0	53.0 (↑ 8.8)	70.5
Deepseek-R1	52.1	70.4 (↑ 6.7)	70.6

Table 6.5: Comparisons of models' performances on solving problems (PL pass@1) and utilizing generated or oracle programs to identify algorithmically similar problems (ASM-PL and Solution). Numbers given are % accuracy.

The latter ability is more determined by a problem's type and topic than its difficulty level.

6.5.2.3 Effectiveness and Efficiency of ASM

Effectiveness. As discussed earlier, models' abilities to identify ASPs are not directly reflected by their ability to fully solve the problems. With *Attempted* Solution Matching, models are able to identify ASPs based on solutions that are not fully correct. To illustrate this robustness, we tested the quality of the generated programs for ASM-PL and surprisingly found that even when programs are incorrect, they can still substantially help identify algorithmically similar problems. The quality of generated programs in ASM-PL is evaluated by pass@1 (Chen et al., 2021). In Table 6.5, we find that for weaker models, with pass@1 ranging from 1.8% to 19.6%, matching mostly-incorrect attempts still gives significant performance gains compared to directly matching problems statements, For Qwen models, ASM performs close to matching oracle solutions. While Claude-3.5-Sonnet has similar performance to Deepseek-R1 when solving problems and matching oracle solutions, it performs much worse when matching its own generated programs. This suggests that it does not identify the correct algorithm during failed solution attempts.

	Accuracy% (↑)	**Cost** (↓)
Statement w/GPT-4o	41.5	6.67X
ASM-PL w/GPT-4o-mini	42.5	1.46X
ASM-NL w/GPT-4o-mini	43.8	1.00X
Statement w/Qwen3-4B-think	43.3	6.23X
ASM-PL w/Qwen3-4B	46.3	5.49X
ASM-NL w/Qwen3-4B	47.3	1.00X
Statement w/Qwen3-8B-think	47.5	3.80X
ASM-PL w/Qwen3-8B	51.7	2.16X
ASM-NL w/Qwen3-8B	51.4	1.00X

Table 6.6: Cross-model and Cross-mode performance/efficiency comparison for GPT-4o and Qwen3 models. *-think* refers to model with the thinking-mode enabled.

Efficiency. To illustrate the efficiency of ASM, we compare the cost (in generated tokens) of ASM-NL and ASM-PL against simply matching problem statements. We find that ASM significantly reduces the number of reasoning tokens used to solve problems. ASM-NL costs 80% as many tokens on average across all models. ASM-PL costs 190% as many tokens as it needs a longer CoT to generate a complete solution than a high-level NL strategy. Even so, to *obtain similar accuracy*, ASM-PL is more efficient than either relying on the built-in thinking mode of LLMs or using a larger model within the same model family. In Table 6.6, we define cost as # gen_tokens $\times$ *unit_price*. ASM-NL can achieve better accuracy while costing only 15% to 26% as much. ASM-PL also costs less than comparing statements. In other words, both ASMs can achieve better accuracy in non-thinking mode or with smaller models, yielding a lower overall cost.

The detailed numbers of generated tokens are given in Table 6.7, where "Statement" and "Solution" include the tokens needed to process the corresponding MCQ, while the other three methods include the cost of both processing the MCQ as well the preliminary step of using the LLM to generate the summary or first attempt solution. The cost of ASM-NL is 1.5x to 2.6x compared to the original statement baseline, while ASM-PL is less efficient, with an inference time that is 2.6x to 5.3x.

Since the accuracy of ASM-NL is about the same or higher than that of ASM-PL, this demonstrates the advantages of generating natural language solutions to help identify ASPs. Using x1.7 tokens is still affordable given 9.7% absolute performance gain.

The two cost analyses show: ASM is not token-cheaper than statement matching under the same model because it requires generating attempted solutions. However, it can be cost-effective when replacing a larger with a smaller model or a thinking-mode selector with a non-thinking model, while achieving better overall performance. Its preprocessing cost can be amortized in retrieval-based settings.

Model	**Statement**	**Summary**	**ASM-NL**	**ASM-PL**	**Solution***
GPT-4o-mini	367	788 (x2.1)	937 (x2.6)	1366 (x3.7)	389
GPT-4o	375	667 (x1.8)	857 (x2.3)	1366 (x3.6)	420
o3-mini-medium	2136	2770 (x1.3)	3180 (x1.5)	11268 (x5.3)	2012
Deepseek-R1	8381	9467 (x1.1)	13371 (x1.6)	28020 (x3.3)	6988
Deepseek-V3	511	803 (x1.6)	1087 (x2.1)	1845 (x3.6)	483
Claude-3.5-Sonnet	297	537 (x1.8)	588 (x2.0)	1269 (x4.3)	278
Gemini 2.0 Flash	394	654 (x1.7)	997 (x2.5)	1023 (x2.6)	327
Avg	1780	2241 (x1.3)	2974 (x1.7)	6594 (x3.7)	1557

Table 6.7: Number of inference tokens on AlgoSimBench-MCQ over different models and methods. *Solution* is a gold-input setting where oracle code solutions were directly provided to the model. ASM-NL and ASM-PL stand for attempted solution matching in natural language and programming language, respectively. The numbers in parentheses show the relative additional cost compared to just processing the original problem statements.

6.5.3 Effects of MCQ Option Position

As pointed out in (Zheng et al., 2024), large language models are not robust selectors. Therefore, we construct AlgoSimBench MCQs with a random permutation of options, making sure that the correct answer appears in each position with the same probability. To further study whether LLMs might be biased towards selecting particular positions, we experimented with putting the correct answer in a particular

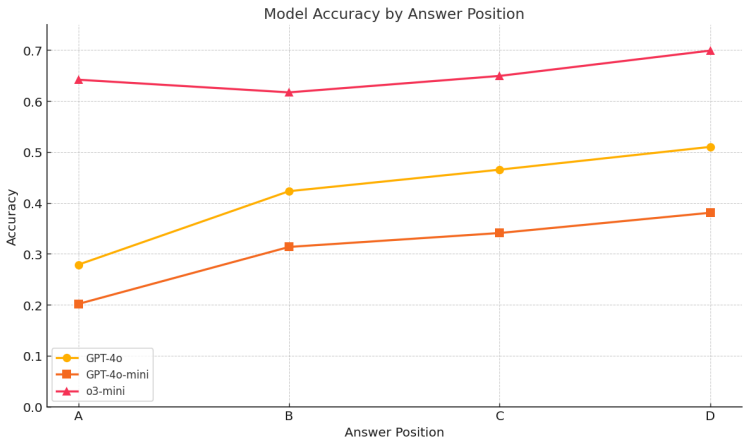


Figure 6.5: MCQ accuracy if the correct option is fixed to a particular position.

Model / Input	**seed1**	**seed2**	**seed3**	**Seed=1000**	**Mean ± Std**
GPT-4o-mini (statement)	33.5%	34.0%	36.8%	35.5%	35.0% ± 1.5%
GPT-4o-mini (ASM-NL)	43.3%	43.0%	43.5%	43.8%	**43.4% ± 0.3%**
GPT-4o (statement)	41.0%	40.3%	42.5%	41.5%	41.3% ± 0.9%
GPT-4o (ASM-NL)	53.9%	50.2%	54.7%	53.2%	**53.0% ± 2.0%**

Table 6.8: Performance under different random seeds for MCQ option layout.

position: A, B, C or D. The results in Figure 6.5 show that LLMs tend to select later choices, further stressing the necessity of randomly permuting options.

We keep the random seed = 1000 in our experiments. While the results are generally robust under different random seeds as shown in Table 6.8.

6.5.4 Effects of Difficulty level

As we discussed in the experiments section, the difficulty level rarely correlates with models' performances in End2End-Selection. We provide the distribution comparison of failed problems and all problems in Figure 6.6.

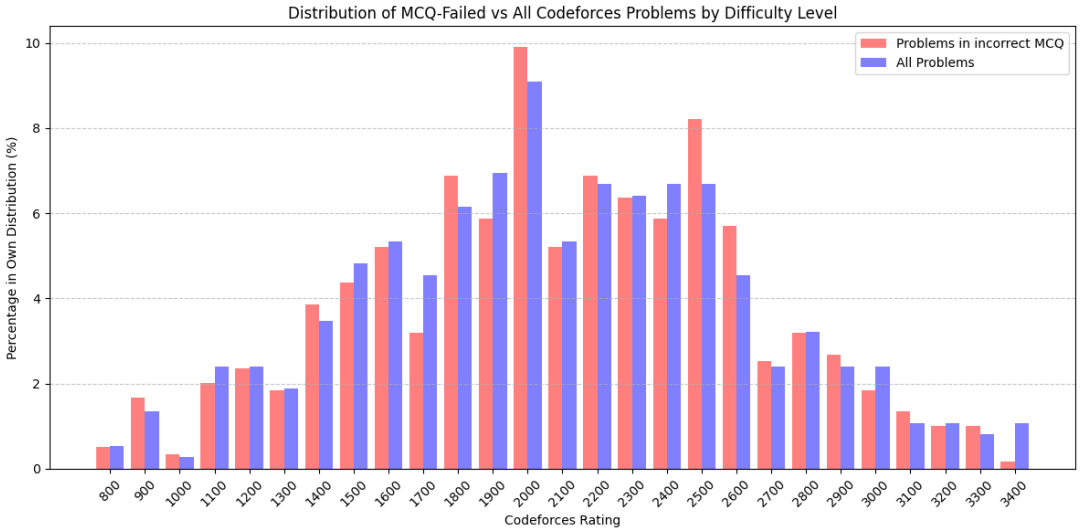


Figure 6.6: Difficulty-level distribution comparison of failed problems and all problems.

6.5.5 Retrieval-Based Selection

In this setting, we treat AlgoSimBench as a retrieval problem as discussed in section 6.5.1. Results for retrieval based on problem statements were shown in Table 6.3. Here, we use an LLM to first map each statement to a **Summary**, an NL solution (**ASM-NL**), and a program solution (**ASM-PL**). We evaluate how well retrieval using these different representations is able to identify the correct ASP. Results are shown in Table 6.9.

ASM-NL again outperforms other methods. Unlike retrieval based on statements (see Table 6.3), using a Summary is better than random guessing. Since AlgoSimBench is constructed so that the textual problem-statement similarity is a misleading indicator of ASPs, summarization is helpful but still provides limited explicit information about algorithms.

An interesting finding is that simple term-matching (BM25) performs better than dense embedding models. A possible reason could be that algorithmic similarity is often better indicated by keywords like algorithmic terms or descriptions of the

	Summary							ASM-NL							ASM-PL						
	BM25	Bart	GCB	CR	CS	Jina	SFR	BM25	Bart	GCB	CR	CS	Jina	SFR	BM25	Bart	GCB	CR	CS	Jina	SFR
GPT-4o-mini	25.6	23.4	20.6	20.1	19.9	20.6	28.4	**35.3**	29.1	22.4	23.6	22.9	24.9	26.1	34.8	26.1	32.8	30.3	24.4	29.4	30.1
GPT-4o	25.8	24.6	26.1	24.9	19.2	21.6	28.9	**42.5**	30.1	32.1	28.4	28.1	31.1	34.6	35.6	28.8	29.1	29.4	21.9	26.6	26.6
o3-mini-medium	39.3	32.6	31.3	29.1	29.9	30.6	34.1	**49.0**	35.6	38.3	40.8	40.3	40.8	39.3	48.8	28.4	39.3	37.3	35.6	46.3	35.1
Deepseek-V3	29.9	26.1	22.4	24.4	24.1	32.1	33.1	**46.0**	33.1	32.1	33.8	37.6	36.8	36.3	45.5	32.3	35.1	38.6	31.1	40.8	37.8
Deepseek-R1	30.1	24.1	25.9	30.1	28.4	30.8	35.1	**52.2**	32.8	31.1	45.8	41.3	43.3	40.0	45.0	32.8	35.9	44.0	38.8	49.8	40.0
Gemini-2.0-Flash	27.1	24.4	18.6	24.4	21.9	23.1	28.4	**41.0**	34.3	26.6	33.6	34.1	33.1	36.8	38.0	36.8	35.6	33.8	29.9	36.1	33.3
Claude-Sonnet-3.5	29.4	25.1	25.4	26.1	25.4	29.1	32.3	**39.6**	28.1	28.6	32.6	31.6	36.3	34.6	35.0	29.1	29.9	29.4	26.4	30.8	33.6
Avg	29.6	25.8	24.3	24.0	22.1	25.3	30.2	**43.7**	31.9	30.2	30.4	30.8	32.4	33.7	40.4	30.6	34.0	32.3	26.7	32.7	32.3

Table 6.9: MCQ accuracy (%) for different LLMs used to generate summaries and attempted solutions, and different retrieval metrics: GCB=GraphCodeBert, CR = CodeRankEmbed, CS = CodeSage-v2, Jina = jina-v2-code, SFR = SFR-Embedding-Code$_{400M}$. For comparison, retrieval performance using the problem statement or oracle solution are given in Table 6.3.

core idea, rather than a full implementation.

Although Retrieval-Based Selection performs significantly worse than End-to-End Selection, it scales much better to retrieving ASPs from a large corpus. End-to-end selection requires fitting all potential retrievals into an LLM input window and is difficult to scale beyond choosing from only four options in an MCQ. In contrast, retrieval-based selection only requires running an LLM on individual problems once to produce more effective problem representations that can then be efficiently retrieved from a large corpus using standard IR techniques (such as BM25) or dense retrieval with compressed text representations (Izacard et al., 2022; Li et al., 2023c).

6.5.6 ICL Exemplar Selection with ASM

Since ASM can help find algorithmically similar problems, we apply it to enhance code generation. Shi et al. (2024) showed that including in-context exemplars of solved problems can enhance LLMs' abilities to solve competitive programming problems. They proposed Episodic Retrieval, using an LLM's solution along with the problem statement to retrieve the most similar human solutions using BM25. Then, the retrieved problem is included as an ICL exemplar, hopefully improving the ability to solve the given problem. ASM, on the other hand, directly matches LLM attempted solutions for both the given problem and problems in the corpus. Utilizing

Exemplar Selection	GPT-4o-mini	GPT-4o
w/o Exemplar	9.4%	17.3%
Retrieve w/Random	10.4%	17.9%
Retrieve w/Statement	11.4%	16.9%
Episodic Retrieval*	12.4%	18.6%
Retrieve w/ASM-NL	**13.7%**	19.2%
Retrieve w/ASM-PL	13.0%	**19.9%**

Table 6.10: ICL enhanced by different exemplar selection methods. Results are 1-shot Pass@1 on the USACO Benchmark. *Episodic Retrieval requires human-generated oracle solutions.

their methodology, we explored how ASM could be used for improved selection of ICL exemplars.

We also applied two baselines that select either a *Random* exemplar or retrieve the most similar exemplar using problem statements. Table 6.10 shows that with the same Retriever (BM25, which outperformed various dense-embedding methods), ASM methods achieve the best pass@1 (Chen et al., 2021) performance. While the absolute improvement is modest, this might be limited by how much performance gain ICL with a single example can bring to competitive programming (Tang et al., 2023a; Patel et al., 2024).

6.6 Conclusion and Discussion

This work introduced AlgoSimBench, a novel benchmark designed to evaluate models' ability to reason about algorithmic similarity between competitive programming problems. This benchmark: 1) provides a focused evaluation of the algorithmic reasoning abilities of LLMs, decoupled from full solution generation; and 2) enables the study of retrieval methods that go beyond surface-level textual similarity, instead capturing deeper structural and problem-solving-related semantics.

We propose Attempted Solution Matching (ASM), which leverages LLM-generated first-attempt solutions—either in natural language or code—to better reflect the core

algorithmic aspects of each problem. In both end-to-end selection and retrieval settings, ASM improves the accuracy of identifying ASPs and improves exemplar selection for in-context learning.

Some key findings were: 1) the performance gap between comparing statements and attempted solutions suggests limited generality of the algorithmic reasoning skills learned from code generation alone; 2) Identifying ASPs depends more on algorithm type rather than problem difficulty, making it a separate axis from problem-solving; 3) BM25 outperforms dense retrievers, indicating that keyword signals capture algorithmic features more effectively than generic semantic embeddings.

We hope AlgoSimBench inspires future work on identifying algorithmic similarity, code retrieval, and retrieval-augmented reasoning.

Conceptually, AlgoSimBench shows that models can be misled by surface-level semantic features such as background stories. The results suggest that even models capable of solving individual CP problems may not reliably recognize shared algorithmic structure across problems. Thus, CP solve rate alone should not be treated as sufficient evidence of generalizable algorithmic understanding. This answers our research question RQ5: success on CP solving does not automatically imply robust, generalizable algorithmic understanding. In particular, models that can solve some problems may still fail to recognize shared algorithmic structure when surface semantics are adversarially misleading.

Chapter 7: InstEmbed: Instruction-Sensitive Code Embeddings via Task-Contrastive Training

7.1 Motivation

Following the findings from AlgoSimBench, we aim to improve the language models' performance as dense retrievers for specified tasks like identifying algorithmically similar pairs.

The Attempt Solution Matching (ASM) component of AlgoSimBench currently relies on a large language model (LLM) to rewrite user queries in order to improve retrieval quality. However, this approach is computationally expensive and does not scale well to large corpora. Interestingly, we observe that simple term-matching methods such as BM25 often outperform dense embedding models in retrieving algorithmically similar problems or code snippets. This counterintuitive result raises the question of why dense retrieval methods underperform in this setting compared to traditional sparse retrieval.

One explanation lies in the fundamental differences between sparse and dense methods. BM25 is keyword-focused: it ranks documents by term frequency–inverse document frequency (TF–IDF) weighting (Robertson and Zaragoza, 2009), giving high importance to rare but discriminative tokens such as algorithm names ("Dijkstra," "FFT"), mathematical operators, or domain-specific jargon. In contrast, dense retrieval methods map entire sentences into continuous vector representations using neural encoders, typically by mean-pooling over token embeddings or using the final hidden state of a special token (e.g., [CLS]) (Wang et al., 2024a; Izacard et al., 2022). These embeddings are designed to capture holistic semantic meaning rather than emphasize localized, task-specific signals. Furthermore, current embedding models are trained on mixed-domain data without explicit task conditioning; robustness across tasks depends largely on how positive and negative pairs are constructed during

training (Jain et al., 2021; Gao et al., 2021b; Suresh et al., 2025), not on architectural mechanisms for disambiguating retrieval intents.

A plausible hypothesis is that features critical for identifying algorithmic similarity are concentrated in a small set of discriminative tokens—such as algorithm names, mathematical terms, or concise descriptions of the core idea—rather than in the broader contextual or implementation details of a solution. In contrast, dense retrieval methods based on LLM-derived embeddings assume that a sentence (or document) can be represented by a single global embedding vector, typically obtained via mean pooling or last-token representations. These embeddings are designed to capture holistic semantic meaning, not task-specific signals. As a result, current dense models may dilute or obscure the fine-grained, keyword-level distinctions that are crucial for algorithmic retrieval tasks. Moreover, in domains such as code retrieval, the target notion of similarity is inherently multi-faceted: queries may seek functionality similarity, algorithmic similarity, useful references (e.g., function documentation), or stylistic similarity (e.g., comments). Without task-specific conditioning, embedding models cannot easily disambiguate these different intents.

7.2 Introduction

Information Retrieval (IR) serves as the backbone for navigating large-scale knowledge bases, a task that has been fundamentally transformed by dense embedding models trained via contrastive learning. Early milestones, such as (Reimers and Gurevych, 2019) and (Gao et al., 2021b), established the viability of mapping semantic textual similarity into continuous vector spaces. This paradigm evolved rapidly toward unified, instruction-aware embedding spaces, popularized by works like INSTRUCTOR (Su et al., 2023a) and subsequent state-of-the-art (SoTA) models trained from Large Language Models (LLMs) (Zhang et al., 2025b; Lee et al., 2025). While these models have achieved significant breakthroughs on massive retrieval benchmarks like MTEB (Muennighoff et al., 2023), semantic-similarity-based

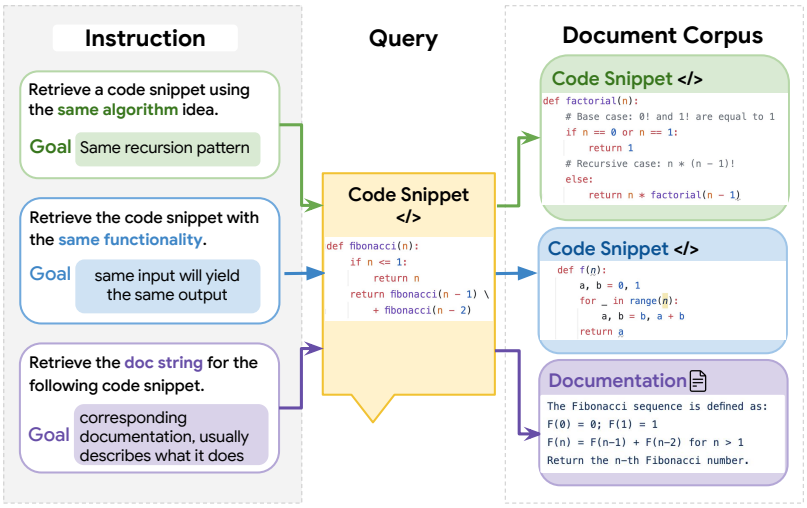


Figure 7.1: Illustration of instruction-oriented code retrieval. Each document corresponds to an instruction under the same query.

dense embedding retrieval is less commonly used in practical Retrieval-Augmented Generation (RAG) applications recently. Jiang et al. (2025) found that static semantic embeddings fail to capture the complex, exact-match phrasing required for real-world evidence seeking, Li et al. (2026) found that agents working with exact-match lexical tools like grep yield higher performance for retrieval tasks with less cost. These findings suggest that while dense retrievers excel at broad semantic overlap, they struggle to capture highly constrained, accurate information in a real-world setting.

This limitation is particularly acute in the domain of automated software engineering. Code retrieval is an inherently complex, highly structured task that demands precise navigation across heterogeneous corpora containing source code, API documentation, unit tests, and natural language discussions. As highlighted by recent benchmarks, CodeRAGBench (Wang et al., 2025), finding the "useful" code is rarely a static semantic matching problem, as its definition might differ in different RAG contexts. Li and Mooney (2025) find that simple BM25 retriever performs better at matching chain-of-thoughts for programming problems than cutting-edge dense em-

bedding models. This leads to our question: *Can current instruction-aware embedding models genuinely understand and follow the instruction?* Specifically, whether they can identify the correct retrieval target based on how similarity is defined in this task and focus on relevant information. The Figure 7.1 illustrates a simple code retrieval task under this setting. Surprisingly, we found that SoTA models fail at following such simple instructions, ranking the iterative Fibonacci code snippet as the top document regardless of the instruction is. The same query can have several semantically related but mutually wrong answers. A useful code retriever must understand which relation the instruction asks for.

To address this, we propose InstEmbed, a novel training framework designed to enforce instruction sensitivity in the embedding space by formulating a task-contrastive learning objective. In addition, we explicitly model retrieval instructions across three explicit dimensions: (1) similarity definition: Specifies the underlying relationship required between the query and the target, (2) target object: Defines the expected modality (e.g., code, documentation, function snippet) or format of the retrieved document, and (3) contextual focus: Directs the model's attention toward specific query components. Given a <Task, Query, Doc> triplet, InstEmbed constructs a set of new task from it, each one adapts the same query, but with a distinct task purpose and corresponding target document. Documents for the same query but under different tasks mutually server as hard negatives, since they are semantic relevant (i.e., positive in a different context), but cannot address what the instruction required. Training to distinguish different goals in different tasks for the same query forces the model to bind semantic matching directly to the instruction content.

With less than 1M open-source training data, spanning across 14 coding tasks, and one-stage contrastive training from (instruction, query, positive, negative) quadruples, InstEmbed-0.5b trained from Qwen2.5-Coder (Hui et al., 2024) achieves 0.7 better retrieval score on MTEB-Code than open embedding models with similar sizes (Zhang et al., 2025c; Qin et al., 2025; Zhang et al., 2025b). The InstEmbed

training framework yields a 1.1-point gain in retrieval score compared to training with in-batch and corpus-mined hard negatives without the task-contrastive setting.

To systematically evaluate models' abilities to follow instructions under conditions that mirror the complexity of software engineering workflows, we design two families of diagnostic evaluations based on tasks from MTEB (Muennighoff et al., 2023) and AlgoSimBench (Li and Mooney, 2025). *Target-Aware* tests embedding models' abilities retrieve different targets (e.g., code or doc) to the same query under different instructions. *Noise-Robust* tests if models can ignore irrelevant information (e.g., environments for program execution) and retrieve the original document as required. We found that current SoTA embedding models perform poorly on two diagnoses, while InstEmbed-0.5b surppasses them on both in-distribution and out-of-distribution tasks.

Our contributions are threefold:

- We formulate instruction-oriented code retrieval as a problem where the instruction defines what counts as the correct document, rather than serving only as a task prefix.
- We propose InstEmbed, a task-contrastive training framework that models instructions through target object, similarity definition, and contextual focus, and uses instruction-violating documents as hard negatives.
- We introduce Target-Aware and Noise-Robust diagnostic evaluations to measure whether embedding models can follow fine-grained code retrieval instructions.
- We show that InstEmbed improves instruction sensitivity over comparable baselines while maintaining competitive standard code retrieval performance with a simple one-stage contrastive training setup.

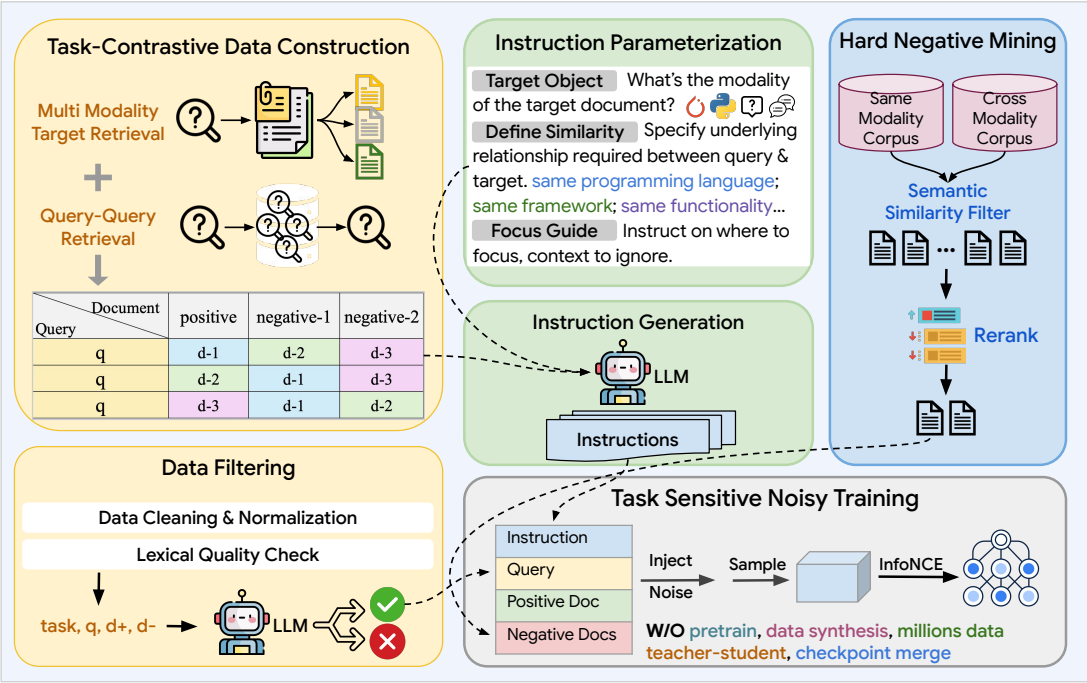


Figure 7.2: An overview of InstEmbed method. Task-Contrastive data is constructed by switching different tasks and corresponding positive targets (modality), filtered by heuristic cleaning as well as LLM annotation. While specific, diverse instructions generated by LLMs are injected along with optional noise. Hard negative mining is adapted to select and re-rank semantically similar documents from corpus of different modalities. Instructions and noise are applied to queries when training with InfoNCE loss.

7.3 Method

InstEmbed trains a code embedding model where the retrieval is sensitive to the instruction that defines the task and specifies the wanted target, rather than treating the instruction as only a task label prepended to the query. Figure 7.2 gives an overview of the framework. Starting from existing code retrieval data, we first construct task-contrastive examples by creating alternative retrieval targets for the same or structurally related query. We then generate explicit instructions that specify the target object, the intended similarity definition, and the contextual focus. Documents that are correct under one instruction but violate another instruction are used as hard negatives. Additional semantically similar hard negatives are mined from same-modality and cross-modality corpora, then filtered by heuristic checks, a reranker, and LLM verification. Finally, we train the model with an InfoNCE objective (van den Oord et al., 2019), optionally injecting irrelevant query context to improve robustness.

7.3.1 Instruction Parameterization

7.3.2 Task-Contrastive Data Construction

The core idea of InstEmbed is that one query can support multiple retrieval tasks, but each task has a different correct target. Given a seed retrieval example, we expand it into a set of task-specific training instances:

$$\mathcal{T}(q) = \{(I_k, q_k, d_k^+)\}_{k=1}^{K},$$

where I_k is the instruction, q_k is the query, and d_k^+ is the positive document for the k-th retrieval task. In many cases, the query is shared across tasks. In other cases, q_k is a structurally transformed version of the original query, such as a query-to-query retrieval instance.

We construct task-contrastive examples through two complementary strategies.

Multi-modality target retrieval. For datasets containing multiple target modality types, we create separate retrieval tasks by switching the requested target object. For example, the same natural language question may be paired with a SQL query, a table schema, an executed answer, or a natural language explanation. Similarly, a programming problem may be paired with a solution program, unit tests, or a related problem description. Each target type defines a distinct retrieval task, even when the query content is unchanged.

Query-query retrieval. When a dataset does not provide rich paired metadata across target modality types, we construct additional tasks by transforming part of the corpus into query-to-query retrieval. For example, programming problems can be paired with algorithmically similar problems, StackOverflow questions can be paired with similar questions or titles, and code-edit instructions can be paired with related edit descriptions. This increases the diversity of retrieval goals without relying on synthetic document generation.

The output of this stage is a set of related tasks in which the same query or query family is associated with multiple plausible but instruction-specific targets. These task variants provide the supervision needed to distinguish "related to the query" from "correct for this instruction."

7.3.3 Instruction Parameterization

For each task-specific training instance, we generate an instruction that makes the retrieval goal explicit. Following the structure shown in Figure 7.2, each instruction is described by three factors.

Target object. The target object specifies the modality or format of the document that should be retrieved. Examples include source code, docstrings, SQL queries, table schemas, natural language explanations, unit tests, Git diffs, question titles,

answers, or similar programming problems. This factor answers the question: *what kind of target should the model retrieve?*

Similarity definition. The similarity definition specifies why the retrieved document should match the query. Depending on the task, the desired match may be based on functional equivalence, algorithmic similarity, shared programming language, shared framework usage, matching database entities, equivalent edit behavior, or another task-specific criterion. This factor answers the question: *what makes a document correct for this query?*

Focus guide. The focus guide specifies the crucial and ignorable parts in the query for retrieving the specific target. For example, in text-to-SQL retrieval, the instruction may tell the model to focus on database entities, table relations, filter conditions, and aggregation requirements while ignoring conversational phrasing. In code retrieval, it may tell the model to focus on the required algorithm or API behavior while ignoring surrounding background context. This factor answers the question: *where to pay attention to this query?*

We use these factors to produce natural language instructions. This keeps the training format close to inference-time usage, where users or agents may provide free-form retrieval requests. Importantly, the instruction is not treated as a shallow task prefix. It specifies the target, the matching criterion, and the relevant query scope, all of which determine which document should be considered positive.

7.3.4 Instruction-Violating Hard Negatives

Task-contrastive construction gives us a direct source of hard negatives. For a fixed query or query family, a document that is positive for one instruction can be an incorrect answer for another instruction. Therefore, for each active training instance (I_k, q_k, d_k^+), we treat positives from alternative tasks as instruction-violating

negatives:

$$\mathcal{D}^{-}_{\text{task}}(I_k, q_k) = \{d_j^{+} \mid j \neq k,\ (I_j, q_j, d_j^{+}) \in \mathcal{T}(q)\}.$$

These negatives are difficult because they are genuinely related to the query. They may share functionality, entities, language, problem topic, or software context. However, they fail to satisfy the current instruction. For example, a docstring may correctly describe a function, but it is a wrong target when the instruction asks for source code. An iterative implementation may have the same input-output behavior as a recursive implementation, but it is wrong when the instruction asks for the same algorithmic pattern.

This is the main difference between InstEmbed and standard task-prefixed contrastive learning. Standard hard negatives usually teach the model to separate relevant documents from unrelated or less relevant documents. Instruction-violating negatives teach the model a finer distinction: a document can be semantically close to the query and still be wrong because it does not match the instruction.

7.3.5 Semantic Hard Negative Mining

Instruction-violating negatives capture mistakes across task goals, but they do not cover all difficult retrieval errors. We therefore also mine semantically similar hard negatives from both same-modality and cross-modality corpora. Given an instruction-query pair, we first retrieve candidate documents with high semantic similarity. These candidates are likely to be close to the query in the embedding space, making them useful negatives for contrastive training.

We then apply a semantic filtering and reranking stage to remove candidates that directly satisfy the instruction. This step is necessary because highly similar documents may include true positives or near-duplicates. The remaining documents are used as additional hard negatives:

$$\mathcal{D}^{-} = \mathcal{D}^{-}_{\text{task}} \cup \mathcal{D}^{-}_{\text{semantic}}.$$

This combines two sources of difficulty: documents that are wrong because they violate the instruction, and documents that are wrong despite being semantically close to the query.

7.3.6 Data Filtering and Verification

Because task expansion and hard-negative mining can introduce noisy examples, we apply multiple filtering stages before training. First, we perform data cleaning and normalization to remove entries that are overly short or of bad lexical quality. Second, we use source-key matching and text matching to detect near-duplicates and possible false negatives. Third, we use a lightweight instruction-following LLM as a verification gate: given an instruction, query, and candidate document, the verifier determines whether the document satisfies the instruction. Invalid positives and ambiguous negatives are removed.

This filtering stage is important for instruction-sensitive training. If a document is incorrectly labeled as a negative despite satisfying the instruction, the model receives a contradictory signal. We therefore prioritize precision in the constructed training tuples over maximizing the number of training instances.

7.3.7 Task-Sensitive Noisy Training

To train the contextual focus component of the instruction, we inject irrelevant but plausible context into a subset of training queries. The noise can appear as a prefix or suffix and may include system descriptions, execution environments, pseudo-user profiles, conversational background, or other text that commonly appears in agent workflows. The positive document and hard negatives remain unchanged, but the instruction is updated to tell the model which parts of the query should be ignored.

Formally, for a training instance $(I, q, d^+, \mathcal{D}^-)$, we sample a noisy variant with probability ϵ:

$$q_{\text{noise}} = \text{InjectNoise}(q),$$

and construct a revised instruction I_{noise} that explicitly identifies the relevant focus and irrelevant context. The noisy training instance becomes:

$$(I_{\text{noise}}, q_{\text{noise}}, d^+, \mathcal{D}^-).$$

This teaches the model that longer input is not always more important input. When the instruction says to ignore environment logs, user profiles, or background descriptions, the embedding should remain close to the same positive document rather than drifting toward targets related to the noise.

7.3.8 Training Objective

We train InstEmbed with a single-stage contrastive objective. The query encoder takes the instruction and query together as input, while the document encoder takes the candidate document:

$$h_q = f_\theta([I; q]), \qquad h_d = f_\theta(d).$$

We compute scaled cosine similarity:

$$s(q, d) = \frac{\cos(h_q, h_d)}{\tau},$$

where τ is the temperature.

For each training instance, the candidate set contains the positive document, in-batch negatives, instruction-violating negatives, and mined semantic hard negatives:

$$\mathcal{D} = \{d^+\} \cup \mathcal{D}^-_{\text{batch}} \cup \mathcal{D}^-_{\text{task}} \cup \mathcal{D}^-_{\text{semantic}}.$$

We apply a contrastive training rate α for alternative tasks and a noisy training rate σ for noise injection, and only utilize a portion of the artificial training instances. Then the model is optimized with the InfoNCE loss:

$$\mathcal{L} = -\log \frac{\exp(s(q, d^+))}{\sum_{d \in \mathcal{D}} \exp(s(q, d))}.$$

When source-key matching or text matching identifies a candidate negative as a possible false negative, we mask the corresponding loss term. This prevents the model from being penalized for assigning high similarity to documents that may also satisfy the instruction.

Overall, InstEmbed uses the same basic contrastive training objective as standard embedding models, but changes the supervision. Instead of relying only on broad semantic positives and mined negatives, it trains on examples where the instruction decides which of several related documents is correct. This makes the learned embedding space sensitive to the target object, similarity definition, and contextual focus specified by the instruction.

7.4 Experiments

7.4.1 Experimental Setting

Benchmarks: We evaluate InstEmbed from two complementary perspectives. First, we measure whether instruction-sensitive training preserves general code retrieval ability on standard benchmarks. Second, we evaluate whether the model can follow fine-grained retrieval instructions in settings where the same query is associated with multiple naturally related targets. For general retrieval, we use MTEB-Code as the main benchmark suite.[1] For instruction-sensitive evaluation, we construct diagnostic tasks from MTEB-Code, AlgoSimBench, HumanEval-Retrieval, and DS1000-Retrieval. These diagnostic tasks are designed to evaluate behaviors that are not directly measured by standard code retrieval benchmarks: selecting the requested target type and ignoring irrelevant context.

Model & Data: We conducted our experiments with Qwen2.5-Coder-0.5B as the base model. We use Qwen3.5-4B as the Reranker to only output binary labels. We use

[1]We exclude SyntheticText2SQL, CosQA, and CodeFeedbackST because of severe benchmark issues identified during preprocessing.

open-source training splits from the above-mentioned benchmarks and *conala (Yin et al., 2018), xlcost (Zhu et al., 2022), self-instruct-starcoder, github-jupyter-parsed* from CodeParrot[2], carefully filtering out same-source data overlapping with test sets. After data cleaning, we only sample a small portion from each task, yielding ˜1M training data in total. Unlike Qwen3-Embedding (Zhang et al., 2025b) or Jina-Code-Embeddings (Kryvosheieva et al., 2025), we do not synthesize any high-quality data pairs for training.

Hyper-Parameters We set the number of hard negatives to be 7 in total; batch-size=256; max sequence length =512; warmup steps = 300; learning-rate = 1e-5; contrastive sample rate = 10%; noise sample rate = 5%.

Model and Baselines We include state-of-the-art models from the MTEB-Code leaderboard to compare. We select open-source, similar-size models from the leaderboard, Jina-Code-Embedding-0.5B and Qwen3-Embedding-0.6B, to conduct our diagnostic evaluation tasks to compare. We also train a baseline model using the same training data as ours, standard InfoNCE loss and task-prefixed instructions like (Qin et al., 2025).

7.4.2 Main Experiments

We evaluate the performance for code retrieval tasks from MTEB (Muennighoff et al., 2023). Despite the fact that InstEmbed is designed for task-specific, instruction-sensitive code embedding; it still achieved promising performance. Our proposed InstEmbed achieves on-par performance with state-of-the-art models. Table 7.1 reports retrieval performance on MTEB-Code. Although InstEmbed is designed primarily for instruction-sensitive code retrieval, it remains competitive on standard retrieval benchmarks. InstEmbed-0.5B achieves an average NDCG@10 score of 81.13,

[2]`https://huggingface.co/codeparrot/datasets`

Model	Size	Data	Apps	CoiR-CSN	CodeEdit	CF-MT	CSN-CC	CSN	CT-Contest	CT-DL	SOQA	Avg.
C2LLM-7B	7B	3M	86.71	89.79	81.49	90.66	**97.90**	91.07	92.51	34.13	94.85	84.35
Qwen3-Embedding-8B	8B	12M	91.07	89.51	76.97	89.93	96.35	92.66	**93.73**	32.81	94.75	84.20
Qwen3-Embedding-4B	4B	12M	89.18	87.93	76.49	89.51	95.59	92.34	90.99	35.04	94.32	83.49
jina-code-embeddings-1.5b	1.5B	3M	86.63	86.45	84.43	86.18	91.12	91.38	92.54	37.32	92.37	83.16
jina-code-embeddings-0.5b	0.5B	3M	84.17	85.73	83.25	85.73	90.41	90.68	90.37	**41.69**	91.04	82.56
F2LLM-v2-0.6B	0.6B	6M	90.45	87.21	62.16	89.56	92.65	90.11	90.27	34.19	94.35	81.22
gemini-embedding-001	–	–	**93.75**	81.06	81.62	85.33	84.69	91.33	89.53	31.47	96.71	81.72
F2LLM-v2-330M	330M	6M	83.86	85.66	59.00	88.64	89.66	88.56	87.93	36.86	92.00	79.13
C2LLM-0.5B	0.5B	3M	61.02	86.71	71.39	88.63	96.29	89.20	84.27	33.99	89.40	77.88
Qwen3-Embedding-0.6B	0.6B	12M	75.34	84.69	64.42	86.39	91.72	91.01	86.05	31.36	89.99	77.88
embeddinggemma-300m	300M	¿10M	84.39	75.54	62.10	80.26	73.71	90.15	85.51	33.52	86.47	74.63
Ours-Baseline	0.5B	1M	72.19	**93.70**	66.70	85.74	91.01	92.18	87.43	32.62	90.89	79.16
Ours-InstEmbed	0.5B	1M	78.21	90.14	77.12	86.41	89.02	93.30	90.10	33.17	92.71	81.13

Table 7.1: Retrieval performance (ndcg@10) across code-related benchmarks. Scores are reported as percentages. Task names: CoiR-CSN: CoiR-CodeSearchNet; CodeEdit:CodeEditSearch; CF-MT: CodeFeedback-MT; CSN-CC: CodeSearchNet-ccr; CSN: CodeSearchNe.t; CT-Contest: CodeTransOcean-Contest; CT-DL: CodeTransOcean-DL; SOQA: StackOverflowQA

outperforming the same-data baseline by 1.97 points and C2LLM-0.5B by 3.25 %. It is 1.43 % below jina-code-embeddings-0.5B, which uses a larger training set including LLM-synthesized data for MTEB-Code. On the contrary, InstEmbed is trained on only one million filtered instances without synthetic pair generation.

7.4.3 Diagnostic Evaluation Tasks

To test the conditional relevance capabilities of embedding models, we construct two diagnostic evaluation suites. These tasks move beyond static semantic matching to evaluate the models' adherence to specific instruction dimensions.

7.4.3.1 Target-Aware

This task evaluates the model's ability to shift its retrieval distribution toward a specific target object modality based strictly on the instruction I, keeping the query q constant.

Setup: For each domain, we pool all available target modalities into a single, mixed corpus. We then issue identical queries but vary the instruction to ask for

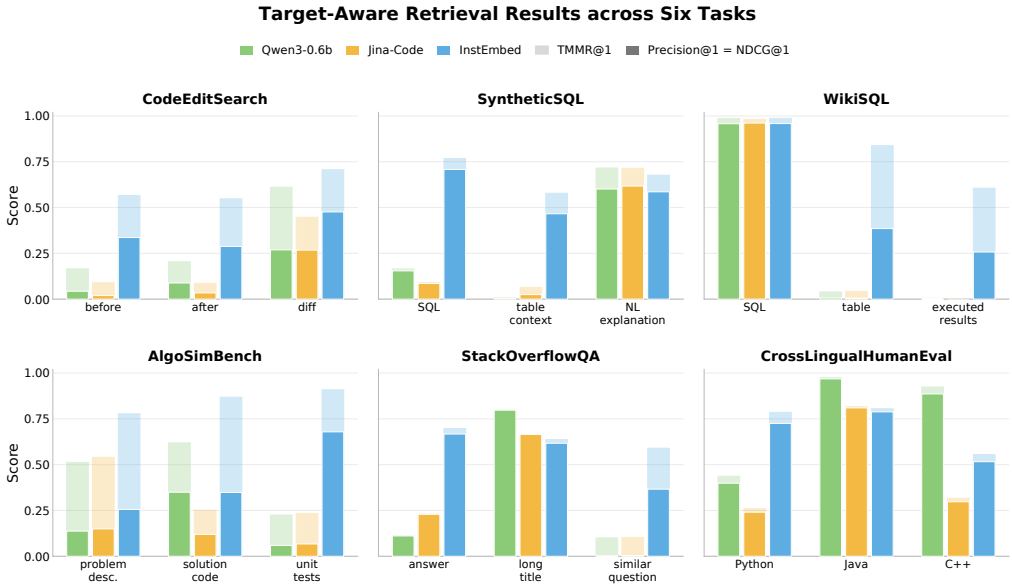


Figure 7.3: TMMR@1 (Target Modality Match Rate @ Top 1) and Precision@1 for Task-Aware, comparison across jina-code-embedding-0.5B, Qwen3-code-embedding-0.6B, InstEmbed-0.5B.

different target formats. We evaluate across the following data and modalities:

Task Name	Query	Target 1	Target 2	Target 3
CodeEditSearch	Edit Instruction	Before-edit code	After-edit code	Git diff
SyntheticSQL	NL Question	SQL Query	Create Table SQLs	NL Explanation
AlgoSimBench	Problem Description	Problem desc. (Q2Q)	Source code	Unit tests
StackOverflowQA	Technical Question	Similar Questions	Question Titles	Answer
WikiSQL	NL Question	SQL Query	Table Entries	Answer (Executed Res)
HumanEval	NL Instruction	Python code	Java code	PHP code

Table 7.2: Datasets, query types, and the distinct target modalities used to evaluate target-awareness. Targets cannot match across modalities since the instruction would specifically ask for certain modality (e.g., python code; question). All "NL" refers to natural language, English specifically in this context.

Evaluation Metrics: As we care about the precision of top retrieved document, we use **Precision@1** as the primary ranking metric. Additionally, to explicitly measure instruction adherence, we report the **Top-1 Target Modality Matching Rate (TMMR@1)**, which calculates the percentage of queries where the top-retrieved document correctly belongs to the modality requested by the instruction.

TMMR@1 is a loose evaluation of whether an embedding model can retrieve the document of the targeted modality. For example, if the instruction is asking for Python code, TMMR@1=1 if top 1 retrieved target is indeed Python code, regardless of whether it corresponds to the query.

Results: We illustrate TMMR@1 and Precision@1 in 7.3, where NDCG@1 = Precision@1 = Recall@1, as each query corresponds to only one document. Our InstEmbed model achieves top1 robustness in 5 out of 6 tasks. While AlgoSimBench, StackOverflowQA and CrossLingualHumanEval are out-of-distribution tasks for InstEmbed, it can still perform well on retrieving the correct target. It's worth noticing that TMMR@1 beats Precision@1 by a large gap, meaning that the model can find the targets within the instructed scope, but cannot further locate the corresponding document. We note that on 5 tasks, both baseline models are largely biased to one modality, achieving high TMMR@1 on this modality alone. Qwen3-Embedding-0.6B is sensitive to the instruction's mention of the programming language, and achieves even better performance than InstEmbed.

7.4.3.2 Noise-Robust

This task evaluates the contextual focus of the model, specifically its ability to ignore irrelevant context appended to the query q when explicitly commanded by the instruction I.

Setup: We synthesize a noisy query q_{noise} by injecting irrelevant yet fluent and dense text into the original query q, and we update I to explicitly instruct the model to ignore this noise. We design three domain-specific noise injection tasks:

- **APPS-Retrieval:** Appending random, high-level system descriptions (e.g., *"We are designing a banking system for a financial company..."*) to simple coding queries.
- **DS1000-Retrieval:** Injecting dense operational environment logs (e.g., CPU,

Task	# Queries	T# Noise	T# Query	T# Doc
APPS	3,765	118.77	309.14	82.58
DS1000	1,998	71.09	136.11	70.16
StackOQA	1,994	89.41	162.34	168.72

Table 7.3: Statistics of noise-injected retrieval queries. T# represents the average number of tokens.

memory, OS, Python version, Anaconda dependencies) before the data science query.

- **StackOverflowQA-Retrieval:** Prepending pseudo-user profiles (e.g., community name, join year, expertise tags, answered counts) to the actual technical question.

We give the statistics of the noise injected in 7.3, on average, noise is about 38% to 55% of the length of the query across tasks.

Evaluation Metrics: We report the standard **NDCG@10** both before and after noise injection to measure retrieval degradation. Normalized Discounted Cumulative Gain at rank 10 (NDCG@10) is given by the following: Given relevance score rel_i for the item ranked at position i, we compute

$$\text{DCG@10} = \sum_{i=1}^{10} \frac{2^{rel_i} - 1}{\log_2(i+1)}, \qquad \text{NDCG@10} = \frac{\text{DCG@10}}{\text{IDCG@10}},$$

where IDCG@10 is the maximum possible DCG@10 obtained by sorting the results by decreasing relevance. NDCG@10 ranges from 0 to 1, with higher values indicating that more relevant items are ranked closer to the top.

A drop in NDCG@10 is calculated as ΔNDCG@10. Besides ranking degradation, we report the similarity score drop ΔS to directly measure representation-level robustness, which measures the exact decrease in cosine similarity between the original query representation and the positive document (d^+), versus the noisy query representation and the same document:

$$\Delta S = \text{sim}(h_{q_{\text{noise}}}, h_{d^+}) - \text{sim}(h_q, h_{d^+})$$

If an embedding model truly follows the instruction to ignore irrelevant context, adding such context should not substantially move the query representation away from the intended target. A small ΔS therefore indicates that the model preserves the original retrieval intent in the embedding space, even before considering corpus-level ranking effects.

Results: The numbers are shown in Table 7.4. We find that no model is affected drastically across tasks, even with long noise prefixes injected into the query. Injecting system design prompts into APPS interferes with performance the most. Nevertheless, our model still achieves the best robustness across the two other models with similar sizes.

Task	Model	NDCG@10	ΔNDCG@10	ΔSim Score
APPS	jina-code-embedding-0.5b	82.29	-5.41	-2.56
	qwen3-embedding-0.6b	72.96	-16.10	-12.76
	instembed-0.5b	78.21	-1.21	-0.44
DS1000	jina-code-embedding-0.5b	60.62	-4.73	-3.02
	qwen3-embedding-0.6b	60.33	-3.79	-2.12
	instembed-0.5b	61.22	+0.15	+0.09
SO-QA	jina-code-embedding-0.5b	88.35	-1.09	-0.4
	qwen3-embedding-0.6b	89.24	-4.11	-2.09
	instembed-0.5b	90.65	-0.48	-0.02

Table 7.4: NDGC@10, NDGC@10 Drop (Δ) and Similarity Score Drop (ΔS) results for Noise Robust diagnose test. It evaluates how robust an embedding model is when injecting plausible noise into the query.

7.4.4 Ablation Study

To study which component in Figure 7.2 contributes most to InstEmbed, we apply an ablation study on InstEmbed. The ablation study examines which components of InstEmbed contribute to general retrieval, target awareness, and noise robustness. We compare the full model against three variants: removing task-contrastive data, removing noise injection, and removing the reranker/data-filtering stage. We

also include the same-data baseline trained with regular InfoNCE and mined hard negatives. We present the numbers in Table 7.5.

Model **Metrics (avg.)**	**MTEB-Code** ncdg@10	**Task-Aware** precision@1	**Noise-Robust** Δ NDCG@10
InstEmbed	81.13%	50.13	-1.54
w/o task-contra	80.65%	27.91	-2.60
w/o noise	81.74%	49.83	-5.99
w/o reranker	80.20%	52.24	-1.86
Baseline	79.16%	25.68	-5.79

Table 7.5: Ablation study on InstEmbed. Specifically, we study the following settings: 1) InstEmbed w/o task-contrastive data, where all task-contrastive negatives are replaced by mined hard negatives; 2) InstEmbed w/o noise injection; 3) InstEmbed w/o data filtering or reranker to clean the hard negative candidates.

The most important result is that task-contrastive data is essential for target-aware retrieval. Removing task-contrastive data reduces Target-Aware Precision@1 from 50.13 to 27.91, a drop of 22.22 points. This ablation is close to the same-data baseline, which reaches 25.68. The comparison strongly supports the central training idea: ordinary contrastive learning with mined hard negatives is not enough to teach the model to select among different valid targets for the same query.

Noise injection has a different effect. Removing noise injection barely changes Target-Aware Precision@1, but it sharply worsens noise robustness: the average NDCG@10 drop increases from -1.54 to -5.99. This confirms that noise-aware training specifically teaches contextual focus rather than simply improving general retrieval.

The reranker and filtering stage mainly affect data quality and general retrieval performance. Removing the reranker lowers MTEB-Code performance from 81.13 to 80.20 and slightly worsens noise robustness, but its Target-Aware Precision@1 is not lower than the full model. This suggests that reranking and filtering are useful for improving the reliability of mined hard negatives and overall representation quality, but they are not the main driver of target-awareness. The main driver of target-awareness is the task-contrastive construction itself.

Overall, the ablation study shows that different components support different parts of instruction sensitivity. Task-contrastive data teaches target selection. Noise injection teaches contextual focus. Reranking and filtering improve the quality of the contrastive signal and help maintain general retrieval performance. This component-level interpretation is stronger than simply saying that every module improves every metric.

7.5 Conclusion

We propose InstEmbed, a contrastive training framework for instruction-sensitive code embeddings. Instead of treating instructions as coarse task prefixes, InstEmbed learns from naturally occurring multi-target structures in code retrieval data, where the same query may be associated with different valid artifacts under different instructions. By parameterizing instructions through target object, similarity definition, and contextual focus, and by using alternative targets as hard negatives, InstEmbed teaches the embedding space to distinguish documents that are merely related from documents that satisfy the current retrieval need. Experiments show that InstEmbed improves target-aware and noise-robust retrieval while maintaining competitive performance on standard code retrieval benchmarks. These results suggest that future code embedding models should move beyond static semantic similarity and learn to represent retrieval intent more directly, especially for agentic software engineering workflows.

Chapter 8: Future Work

8.1 Motivation

From the results InstEmbed in Chapter 7, three limitations remain to apply code retrieval to real-world applications like CodeRAG or Code Agent Frameworks. First of all, the model trained with a contrastive-task setting still struggles with out-of-domain tasks. Despite the diversity of LLM-generated instructions, the variety of tasks is still bounded by the accessibility of open-source retrieval data. In addition, though the representation of the query is conditioned on the instruction, such a design cannot be applied to the document side due to its enormous computation cost. Last but not least, while we can instruct InstEmbed to retrieve content that "addresses the problem", for tasks like CodeRAG, gold "most helpful content" is infeasible, as this is highly dependent on the Code Generator LLMs as well.

In this section, three high-level directions are introduced to further improve learning code embeddings for real-world applications.

8.2 Multi-Facet Embeddings via Instruction-Transformed Embedding for Code Search

In AlgoSimBench (chapter 6), we observe that simple term-matching methods such as BM25 often outperform dense embedding models in retrieving algorithmically similar problems or code snippets. This counterintuitive result raises the question of why dense retrieval methods underperform in this setting compared to traditional sparse retrieval. One explanation lies in the fundamental differences between sparse and dense methods. BM25 is keyword-focused: it ranks documents by term frequency–inverse document frequency (TF–IDF) weighting (Robertson et al., 1995), giving high importance to rare but discriminative tokens such as algorithm names ("Dijkstra", "FFT"), mathematical operators, or domain-specific jargon. In contrast,

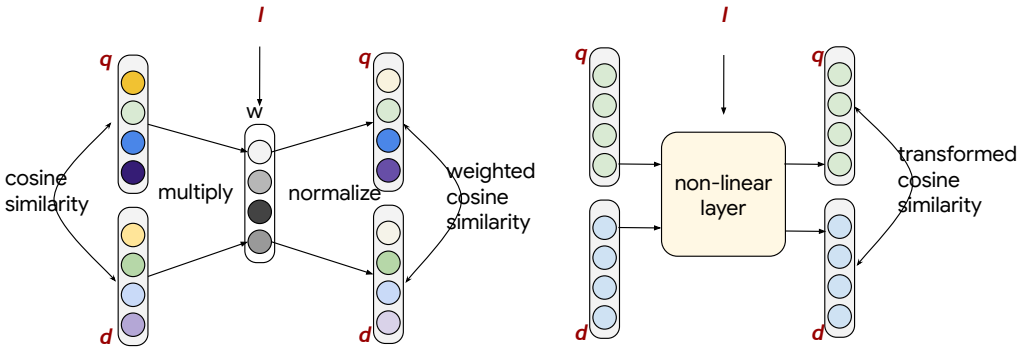


Figure 8.1: Instruction-weighted(left) and Instruction-transformed(right) representations to calculate cosine similarity.

dense retrieval methods map entire sentences into continuous vector representations using neural encoders, typically by mean-pooling over token embeddings or using the final hidden state of a special token (e.g., [CLS]) (Wang et al., 2024b; Izacard et al., 2022). These embeddings are designed to capture holistic semantic meaning rather than emphasize localized, task-specific signals.

While InstEmbed (Chapter 7) can learn a task-specific query representation by conditioning the query representation on different instructions, this design cannot extend to document representation due to high cost during test time, where each document in the entire corpus needs a full LM-pass to gain an instruction-conditioned representation. Instead of learning task-specific embeddings, we can learn to transform general representations to task-specific representations conditioned on instructions. We present two potential variants: *instruction-weighted* and *instruction-transformed*, and illustrate them in Figure 8.1.

In *instruction-weighted*, the instruction is represented, transformed, and normalized into weights, representing how crucial each embedding feature (i.e., vector dimension) is. Applying reweighting to the general representations of both the query and the document can obtain instruction-conditioned representations with a small cost of element-wise multiplication. *instruction-transformed* adopts a similar idea, but it costs dot-product transformations of vectors for better expressiveness. We'll

try two variants and select the best one based on the trade off between perfromance and cost.

8.3 Training Code Embeddings from Downstream Code Generation Feedback

Recent progress in applying Reinforcement Learning (RL) has transformed RAG from static pipelines into dynamic, self-optimizing frameworks. Milestones include training agents to decide when to retrieve for cost-efficiency (Kulkarni et al., 2024), optimizing what to retrieve by rewriting queries (Gao et al., 2025), and training generators to handle noisy contexts (Huang et al., 2025). The current state-of-the-art has moved towards holistic optimization, either by jointly training the entire pipeline or by incorporating it as a team of collaborative agents (Chen et al., 2025b). These advancements, however, have been predominantly validated on natural language tasks like open-domain question answering, where reward signals are based on textual similarity and factual correctness.

Despite these advances, the direct application of existing RAG+RL methodologies to the code domain is challenging. Beyond general RL hurdles like sample inefficiency and complex reward engineering, the core limitation is a mismatch in the definition of "helpfulness." Current reward functions are tailored for natural language, using metrics like F1 scores or semantic similarity. In contrast, the utility of a retrieved code snippet is determined by its functional correctness, syntactic validity, and logical compatibility, for which textual similarity is a poor proxy. Code generation offers a powerful and objective reward signal through execution feedback—whether the code compiles and passes unit tests.

To address the issue of lacking ground-truth query-document pairs for tasks like CodeRAG, one could explore leveraging the downstream performance of a code generation model as a reward signal for retrieval training. The method itself is similar to Yan and Ling (2025); Lin et al. (2024). We could adapt a different setting:

instead of relying solely on static annotations, 1. Instead of hand-crafted preference, our reward signal is given by executing the code against test cases, which is the gold evaluation for correctness, 2. The retriever is encouraged to explore candidate contexts (e.g., by occasionally selecting from beyond the top-ranked results), appending them to the generator's prompt, and observing whether the augmented input improves code generation quality—such as increased test pass rate or reduced generation loss. Instances that yield measurable improvements can then be treated as implicit positive supervision, while less effective alternatives provide contrastive signals. This approach naturally transforms retrieval training into a feedback-driven process where "helpfulness" is defined operationally by its impact on the generator's success. In effect, the embedding model learns to capture similarity not merely in surface-level semantics, but in terms of downstream utility for code generation tasks. An even more ambitious direction is to move beyond retriever optimization alone and consider a co-training framework in which both the retriever and the generator are updated in tandem (Lu and Liu, 2024; Le et al., 2025). While the retriever can be shaped to prefer content that empirically improves task outcomes, the generator itself may also learn to better interpret and leverage the retrieved evidence. By allowing both components to adapt iteratively, the system has the potential to converge toward a cooperative equilibrium: the retriever surfaces content tailored to the generator's evolving strengths, and the generator becomes increasingly skilled at grounding its outputs in the retrieved material. Such a joint optimization paradigm remains largely underexplored, but it opens a promising avenue for making retrieval- augmented generation more robust and context-sensitive across diverse problem settings.

8.4 Learning Code Embeddings for Real-world Code Search Application

Software engineering agents do not retrieve code for one static notion of relevance. Given the same issue, an agent may need the implementation to modify, a test

that exposes the failure, documentation that defines an API contract, or the interface component responsible for a visual symptom. These targets can be semantically close but incompatible in use: a test can reproduce a bug without containing its cause. Real repositories add dependencies, long-tail languages, configuration, logs, screenshots, and dialogue. Retrieval for coding agents is therefore *conditional relevance*: utility depends on the task, requested target, and context to attend to or ignore.

Systems such as Agentless, LocAgent, and CodeScout identify localization as a distinct bottleneck before repair (Xia et al., 2024; Chen et al., 2025c; Sutawika et al., 2026). Recent CORE-Bench results report a sharp decline from conventional code search to issue-conditioned edit and context retrieval (Zhang et al., 2026). Together, these findings suggest that semantic similarity to an issue is not an adequate definition of what an agent should retrieve.

InstEmbed offers a natural interface for instruction-conditioned retrieval, addressing this failure with task-contrastive training: for one query, an artifact that is positive under one instruction becomes an instruction-violating negative under another. However, it relies primarily on curated benchmark pairs and short artifacts. It does not capture repository state, the different evidence strengths within agent trajectories, or non-code targets encountered during real development.

A growing data ecosystem makes a real-world extension possible. SWE-Gym provides 2,438 executable Python tasks and agent trajectories (Pan et al., 2025); SWE-Smith synthesizes 50,000 verifiable break-and-repair tasks (Yang et al., 2025b); and SWE-reBench V2 scales to more than 32,000 executable tasks across 20 languages, plus 120,000 lighter tasks (Badertdinov et al., 2026). Multi-SWE-bench and SWE-bench Multimodal add multilingual and visual domains (Zan et al., 2025; Yang et al., 2025a). Nebius releases 80,036 trajectories with actions, patches, and test outcomes (Nebius AI, 2024), while the Agent Data Protocol (ADP) standardizes heterogeneous logs (Song et al., 2025). Yet a gold patch is strong edit-location evidence, whereas a read in a failed trajectory may only record exploration. Treating every action as

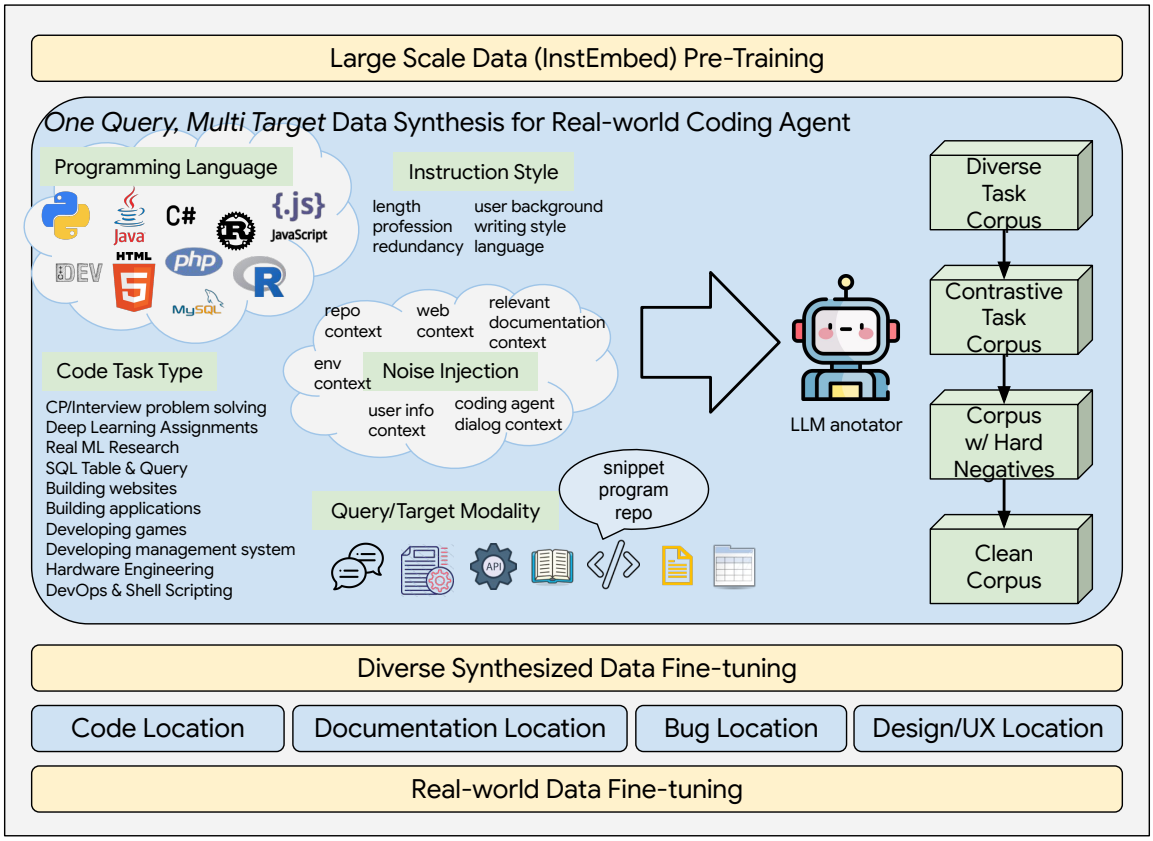


Figure 8.2: Three-stage training to learn instruction-sensitive embeddings for code search in agentic software engineering.

relevant would reproduce agent mistakes.

We could extend InstEmbed through three training stages, as illustrated in Figure 8.2. First, to retain broad pre-training for code semantics and instruction sensitivity. Second, to construct approximately 100,000 synthetic tuples from roughly 40,000 repository events, expanding each event into only the target views supported by its artifacts. Explicit quotas cover code, bug, documentation, stratified by language, repository domain and size, task type, query modality, and difficulty. Instruction style is sampled independently from task semantics, varying length, technicality, user background, redundancy, and language; repository, environment, web, documentation, and dialogue context supply controlled distractors. This prevents target type from becoming tied to one style or language. Missing views are skipped rather than fabricated. Third, to fine-tune on real task and trajectory evidence using source-specific parsers mapped to a canonical event schema.

8.5 Evaluation

The proposed method is designed for situations where the content of gold "helpfulness" is infeasible. Existing benchmarks that are built for RAG-CodeGen can be a great source to train and evaluate our method (Wang et al., 2025; Chen et al., 2025b). In addition, Gupta et al. (2025) curated code generation tasks in a retrieval-augmented setting. Alternatively, performances of benchmarks like (Jimenez et al., 2024) rely heavily on the quality of repo-level retrieval; thus, we might test if our method can improve the performance by retrieving more related information, combined with existing agentic (Antoniades et al., 2025) or Agentless (Xia et al., 2024) frameworks. We should also use the diagnostic evaluation tasks defined in Chapter 7.4.3 to test if the trained models follow instructions correctly while being robust to noise. Potentially, if trained on ICL-enhanced CP programming problems, embedding models are expected to perform better on AlgoSimBench.

Chapter 9: Conclusion

The core focus of this thesis is to enhance competitive-level code generation by utilizing natural language reasoning. The first three works aim at identifying the reasoning gap between natural language and programming language, utilizing natural language reasoning as a bridge towards better CP code generation. The last completed work and future works will focus more on generalizing capacities across different forms of reasoning.

Initially, we evaluated the abilities of Large Language Models (LLMs) in competitive programming. We found that while LLMs struggle to solve problems directly, they are proficient at explaining human-written solutions and translating those verbal explanations back into correct code. Our experiments further revealed that the more detailed the explanation, the more accurately LLMs can implement the corresponding solution.

Based on this observation, we proposed synthesizing problem-solving Chain-of-Thought (CoT) by having LLMs explain human-written code. This method distills a model's explanatory capabilities into the generation of editorial-style CoTs, which outperforms training directly on code and improves problem solve rates. Our experiments also show that training only on the most crucial steps of the CoT yields better performance than training on the entire chain. These findings highlight that natural language can serve as an effective bridge to code generation when used as a preceding reasoning step.

Following the core idea of decomposing symbolic reasoning into natural language reasoning and code implementation, we investigated whether this could be achieved in a training-free framework. We proposed CodeTree, an agentic workflow for code generation. This system separates high-level strategy from implementation, assigning problem reasoning to a "Thinker Agent" that uses natural language to de-

vise distinct strategies. "Coder" and "Debugger" agents then implement the program and fix bugs based on the Thinker's guidance. A "Critic Agent" oversees the process by scoring, expanding, and terminating a tree search for the final solution. Our method, using GPT-4o, yielded better performance than the then-state-of-the-art models while using fewer tokens.

With the emergence of more advanced LLMs and post-training methods, major improvements have occurred in code generation for competitive programming. However, it is less explored whether reasoning learned from problem-solving can generalize to highly relevant tasks. Therefore, we introduced AlgoSimBench, a curated and verified dataset to test a model's ability to identify algorithmically similar problems from a set of semantically similar distractors. The adversarial nature of the benchmark makes this task extremely challenging. To address this, we proposed Attempted Solution Matching (ASM) and found that its application brings performance gains to both end-to-end selection via LLM prompting and to retrieval-based selection methods.

We decided to address the issues in AlgoSimBench, where retrievers are biased to naive semantic matching while overlooking the logic similarity despite what the instruction stressed. We thereby propose InstEmbed, which learns instruction-sensitive code embeddings via task-contrastive training. We utilize existing one-query, multi-target code-domain data and enforce embedding models to learn to distinguish between different tasks as well as to ignore irrelevant context. This is achieved by contrastive learning where the target for another task is treated as a negative example, optimized by InfoNCE loss. The experiments show that InstEmbed can achieve comparable performance with SoTA models of similar sizes, while maintaining large advantages over stress tests like noisy query retrieval or retrieval with mixing modalities of targets.

Three future work directions and potential methods are proposed in the previous chapter. The first future work proposes to learn general embeddings and embed-

ding transformations to shift general embeddings towards any specified task. The second future work aims at exploring co-training for the retriever and generator for retrieval-augmented code generation (RAGCode). The third future work adapts instruction-sensitive training to real-world data and applies it for code search within a modern agentic coding framework.

In conclusion, this thesis investigates how utilizing natural language reasoning can enhance competitive-level code generation and beyond. In this thesis, we present insightful findings in code explanation, generation, and representation, making progress in reasoning distillation, agentic framework for code generation and beyond. We believe this progress can support future research in this domain, and eventually lead to an automatic, intelligent, and controllable software engineering system to assist human engineers.

Appendix: prompts, examples

Explaining Competitive-level Programming Solutions Using LLMs

Case Study Table I.1 presents an example of the input/output of the model, which contains our specific-to-general prompt as well as the comparison between GPT3.5 and GPT4 generated explanations. We can see that both GPT-3.5 and GPT-4 describe the problem and solution very well. Both models correspond the solution logic to the problem situation correctly. In the analysis of why this solution is correct, GPT-3.5 and GPT-4 mention the key idea of decreasing IQ backward means saving IQ for the future. Both models get the key idea to work backward and choose to take the contest greedily. Their analysis of when Doremy should take a contest is correct. One minor issue with GPT-4 is that its analysis does not specify that its description(future, increase) is from a backward perspective.

Table I.2 presents an incorrectly explained example. LLMs can describe the problem and illustrate the printing array operation as "moving the last element to the head of the array". However, both of them fail to understand the purpose of placing the maximum element in the front as they ignore one crucial condition in the problem: all elements are strictly positive. Nevertheless, GPT-4 maintains a better understanding of the problem by noticing the condition that the given array is already sorted in non-decreasing order.

Distilling Algorithmic Reasoning from LLMs via Explaining Solution Programs

Real example with model's responses, as well as detailed prompts are given in Figure I.3 (Explainer); Figure I.4 (Reasoner); Figure I.5 (Coder).

Input You are required to read and try to understand a competitive programming problem statement and interpret its correct solution. Think carefully about why doing this can correctly solve the problem.

Problem:Doremy is asked to test n contests. Contest i can only be tested on day i. The difficulty of contest i is a_i. Initially, Doremy's IQ is q. On day i Doremy will choose whether to test contest i or not. She can only test a contest if her current IQ is strictly greater than 0.
If Doremy chooses to test contest i on day i, the following happens:
if $a_i > q$, Doremy will feel she is not wise enough, so q decreases by 1; otherwise, nothing changes.
Doremy wants to test as many contests as possible. Please give Doremy a solution.
Input
The input consists of multiple test cases. The first line contains a single integer $t(1 \le t \le 10^4)$ — the number of test cases. The description of the test cases follows.
The first line contains two integers n and q $(1 \le n \le 10^5, 1 \le q \le 10^9)$ — the number of contests and Doremy's IQ in the beginning.
The second line contains n integers $a_1, a_2, \cdots, a_n (1 \le a_i \le 10^9)$ — the difficulty of each contest.
It is guaranteed that the sum of n over all test cases does not exceed 10^5.
Output
For each test case, you need to output a binary string s, where $s_i = 1$ if Doremy should choose to test contest i, and $s_i = 0$ otherwise. The number of ones in the string should be maximum possible, and she should never test a contest when her IQ is zero or less.
If there are multiple solutions, you may output any.

Example Input	Example Output
5	1
1 1	11
1	110
2 1	1110
1 2	01111
3 1	
1 2 1	
4 2	
1 4 3 1	
5 2	
5 1 2 4 3	

Note
In the first test case, Doremy tests the only contest. Her IQ doesn't decrease.
In the second test case, Doremy tests both contests. Her IQ decreases by 1 after testing contest 2.
In the third test case, Doremy tests contest 1 and 2. Her IQ decreases to 0 after testing contest 2, so she can't test contest 3.

Hint: Here's an accepted solution from an expert programmer. Can you analyze the code and identify the algorithmic approach used? Please also describe the steps of the algorithm in natural language, explaining what every named variable means, what each step does and how it works. If there is anything non-obvious in the step, explain why it is done. Please notice that in the algorithm, some conclusions might be first deduced from conditions and directly used.

```
def solve():
    n, q =map(int,input().split())
    A=list(map(int,input().split()))
    A.reverse()
    q_now = 0
    record = []
    for a in A:
        if a > q_now and q_now <q: # can
            take off
            record.append(1)
            q_now += 1
        elif a > q_now and q_now >= q:
            record.append(0)
        else: record.append(1)
    print("".join([str(_) for _ in
        reversed(record)]))

t=int(input())
for i in range(t):
    solve()
```

Analyze the problem and the accepted code of the correct solution. Avoid ambiguity in your description as possible. Each part should be independently interpretable, i.e., each point should be clear to readers who have no access to other points.
Answer in the following Format:
1). Brief Problem Summary:
2). Used Algorithm:
3). Step-by-step Solution Description:
4). Explanation of the Solution:
5). Solution in one sentence:
6). Time Complexity:
7). Proof of correctness (Why this is correct):

GPT-3.5's Output:
1). Brief Problem Summary:
Doremy is asked to test n contests. Contest i can only be tested on day i. The difficulty of contest i is a_i. Initially, Doremy's IQ is q. On day i Doremy will choose whether to test contest i or not. She can only test a contest if her current IQ is strictly greater than 0. Doremy wants to test as many contests as possible.
2). Used Algorithm:
Greedy Algorithm
3). Step-by-step Solution Description:
Read the number of test cases.
For each test case:
Read the number of contests and Doremy's IQ in the beginning.
Read the difficulty of each contest.
Reverse the list of difficulties.
Initialize a variable q_{now} to 0 and an empty list record.
For each difficulty a in the reversed list of difficulties:
If a is greater than q_{now} and q_now is less than q, append 1 to the record list and increase q_{now} by 1.
If a is greater than q_{now} and q_{now} is greater than or equal to q, append 0 to the record list.
If a is less than or equal to q_{now}, append 1 to the record list.
Print the binary string obtained by reversing the record list.
4). Explanation of the Solution:
The solution uses a greedy approach to maximize the number of contests Doremy can test. The idea is to start from the last contest and work backwards. For each contest, if its difficulty is greater than Doremy's current IQ and her IQ is less than the maximum allowed IQ, she can choose to take off and decrease her IQ by 1. If the difficulty is greater than her IQ and her IQ is already at the maximum allowed IQ, she cannot take off and must skip the contest. If the difficulty is less than or equal to her IQ, she can test the contest. By working backwards, we ensure that Doremy tests as many contests as possible while maintaining her IQ above 0. *Likert score: 2*
5). Solution in one sentence:
The solution uses a greedy approach to maximize the number of contests Doremy can test by working backwards from the last contest and choosing to test or skip each contest based on its difficulty and Doremy's current IQ.
6). Time Complexity:
The time complexity of the solution is O(n) for each test case, where n is the number of contests.
7). Proof of correctness (Why this is correct):
The solution is correct because it uses a greedy approach that always chooses the optimal solution at each step. By working backwards from the last contest, we ensure that Doremy tests as many contests as possible while maintaining her IQ above 0. The solution is optimal because if we skip a contest that we could have tested, we lose the opportunity to test it later and we may not be able to test as many contests as possible. Therefore, the solution is correct and optimal. *Likert score: 2*

GPT-4's Output:
1). Brief Problem Summary:
Doremy wants to test as many contests as possible without her IQ dropping to 0 or less. The input consists of multiple test cases, each containing the number of contests, Doremy's initial IQ, and the difficulty of each contest. The output should be a binary string indicating which contests Doremy should test.
2). Used Algorithm:
The algorithm is a greedy algorithm that iterates through the contests in reverse order.
3). Step-by-step Solution Description:
Read the number of test cases T.
For each test case:
Read the number of contests n and Doremy's initial IQ q.
Read the list of contest difficulties A and reverse it.
Initialize q_{now} to 0 and create an empty list called record.
Iterate through the reversed list of contest difficulties A:
If the current contest difficulty a is greater than q_{now} and q_{now} is less than q, append 1 to the record list and increment q_{now} by 1.
If the current contest difficulty a is greater than q_{now} and q_{now} is greater than or equal to q, append 0 to the record list.
Otherwise, append 1 to the record list.
Print the record list in reverse order as a string.
4). Explanation of the Solution:
The solution works by iterating through the contests in reverse order and deciding whether Doremy should test the contest or not based on her current IQ (q_{now}) and the contest difficulty. If the contest difficulty is greater than her current IQ and her current IQ is less than her initial IQ, she should test the contest and her IQ will increase. If the contest difficulty is greater than her current IQ and her current IQ is greater than or equal to her initial IQ, she should not test the contest. Otherwise, she should test the contest as her IQ will not decrease. *Likert score: 2*
5). Solution in one sentence:
The solution is a greedy algorithm that iterates through the contests in reverse order and decides whether Doremy should test the contest based on her current IQ and the contest difficulty.
6). Time Complexity:
The time complexity of the solution is $O(n)$ for each test case, where n is the number of contests.
7). Proof of correctness (Why this is correct):
The algorithm is correct because it ensures that Doremy tests as many contests as possible without her IQ dropping to 0 or less. By iterating through the contests in reverse order, the algorithm can make the optimal decision for each contest based on her current IQ and the contest difficulty. If the contest difficulty is greater than her current IQ and her current IQ is less than her initial IQ, she should test the contest and her IQ will increase, allowing her to test more contests in the future. If the contest difficulty is greater than her current IQ and her current IQ is greater than or equal to her initial IQ, she should not test the contest as her IQ will decrease and she may not be able to test future contests. Otherwise, she should test the contest as her IQ will not decrease, ensuring that she can test as many contests as possible. *Likert score: 1*

Figure I.1: A full example of Input prompt, example and GPT's generated explanation(output). Likert scores are provided for points (4) and (7), all other points are scored 2 for both models as it describes the problem and solution well. text color, blue: correct, red: incorrect.

Input
You are required to read and try to understand a competitive programming problem statement and interpret its correct solution. Think carefully about why doing this can correctly solve the problem.

Problem:
Problem: An array a is called ugly if it contains at least one element which is equal to the sum of all elements before it. If the array is not ugly, it is beautiful.
For example: the array [6, 3, 9, 6] is ugly: the element 9 is equal to 6 + 3; the array [5, 5, 7] is ugly: the element 5 (the second one) is equal to 5; the array [8, 4, 10, 14] is beautiful: $8 \neq 0$, $4 \neq 8$, $10 \neq 8 + 4$, $14 \neq 8 + 4 + 10$, so there is no element which is equal to the sum of all elements before it.
You are given an array a such that $1 \leq a_1 \leq a_2 \leq \cdots \leq a_n \leq 100$ You have to reorder the elements of a in such a way that the resulting array is beautiful. Note that you are not allowed to insert new elements or erase existing ones, you can only change the order of elements of a. You are allowed to keep the array a unchanged, if it is beautiful.
Input
The first line contains one integer $t (1 \leq t \leq 2000)$ — the number of test cases.
Each test case consists of two lines. The first line contains one integer n $(2 \leq n \leq 50)$. The second line contains n integers $a_1, a_2, \ldots, a_n$ $(1 \leq a_1 \leq a_2 \leq \cdots \leq a_n \leq 100)$ Output
For each test case, print the answer as follows:
if it is impossible to reorder the elements of a in such a way that it becomes beautiful, print NO; otherwise, in the first line, print YES. In the second line, print n integers — any beautiful array which can be obtained from a by reordering its elements. If there are multiple such arrays, print any of them.

Example Input	Example Output
4	YES
4	3 6 3 6
3 3 6 6	NO
2	YES
10 10	2 4 1 5 3
5	YES
1 2 3 4 5	1 4 4
3	
1 4 4	

Hint: Here's an accepted solution from an expert programmer. Can you analyze the code and identify the algorithmic approach used? Please also describe the steps of the algorithm in natural language, explaining what every named variable means, what each step does and how it works. If there is anything non-obvious in the step, explain why it is done. Please notice that in the algorithm, some conclusions might be first deduced from conditions and directly used.

```
t = int(input())
for _ in range(t):
    __ = input()
    a = list(map(int, input().split()
        ))
    if a[0] == a[-1]:
        print("NO")
    else:
        print("YES")
        print(a[-1], end=" ")
        print(*a[:-1])
```

Analyze the problem and the accepted code of the correct solution. Avoid ambiguity in your description as possible. Each part should be independently interpretable, i.e., each point should be clear to readers who have no access to other points.
Answer in the following Format:
1). Brief Problem Summary:
2). Used Algorithm:
3). Step-by-step Solution Description:
4). Explanation of the Solution:
5). Solution in one sentence:
6). Time Complexity:
7). Proof of correctness (Why this is correct):

GPT-3.5's Output:
1). Brief Problem Summary: Given an array, reorder its elements in such a way that the resulting array is beautiful (i.e., no element is equal to the sum of all elements before it). If it is impossible to reorder the elements to obtain a beautiful array, print "NO".
2). Used Algorithm: Greedy algorithm *Likert score: -2*
3). Step-by-step Solution Description:
a. Read the number of test cases, t.
b. For each test case:
i. Read the size of the array n and the array a.
ii. Check if the first and last elements of the array are equal. If they are, print "NO" and move to the next test case.
iii. Otherwise, print "YES".
iv. Reorder the array by moving the last element to the beginning of the array.
v. Print the reordered array.
4). Explanation of the Solution: The solution uses a greedy approach to reorder the array. If the first and last elements of the array are equal, it is impossible to reorder the array to obtain a beautiful array. Otherwise, we can move the last element to the beginning of the array to obtain a beautiful array. **This is because the last element is not equal to the sum of all elements before it**, and moving it to the beginning ensures that no element is equal to the sum of all elements before it. *Likert score: -1*

5). Solution in One Sentence: Reorder the array by moving the last element to the beginning of the array to obtain a beautiful array, unless the first and last elements of the array are equal.

6). Time Complexity: $O(n)$, where n is the size of the array.

7). Proof of Correctness: If the first and last elements of the array are equal, it is impossible to reorder the array to obtain a beautiful array. Otherwise, moving the last element to the beginning of the array ensures that no element is equal to the sum of all elements before it, **since the last element is not equal to the sum of all elements before it**. Therefore, the solution is correct. *Likert score: -2*

GPT-4's Output:
1). Brief Problem Summary: The problem requires us to reorder the elements of an array to make it beautiful, which means no element in the array should be equal to the sum of all elements before it. We are given multiple test cases, and for each test case, we must determine if it's possible to reorder the array to make it beautiful or not.
2). Used Algorithm: Greedy algorithm *Likert score: -2*
3). Step-by-step Solution Description:
a. Read the number of test cases, t.
b. For each test case:
i. Read the number of elements in the array, n.
ii. Read the elements of the array, a.
iii. Check if the first and last elements of the array are equal. If they are, print "NO".
iv. If not, print "YES" and print the reordered array by placing the last element of the array first followed by the remaining elements in their original order.
4). Explanation of the Solution: **The given solution relies on the fact that if the first and last elements of the sorted array** are equal, it's impossible to reorder the elements in such a way to form a beautiful array. **This is because the first element will always be equal to the sum of all elements before it, which is an empty sum, i.e., zero.** So, in this case, we print "NO". If the first and last elements are not equal, we can reorder the array to make it beautiful by placing the last element (the largest element) first and keeping the remaining elements in their original order. **This ensures that the largest element will never be equal to the sum of all elements before it, as there are no elements before it. Since the array is sorted, the remaining elements will also not satisfy the ugly condition.** *Likert score: -2*
5). Solution in One Sentence: For each test case, print "NO" if the first and last elements of the **sorted array** are equal, otherwise print "YES" and reorder the array by placing the largest element first and keeping the remaining elements in their original order.
6). Time Complexity: $O(t * n)$ where t is the number of test cases and n is the number of elements in the array.
7). Proof of Correctness: As explained in point 4, the solution is correct because it checks the condition that the first and last elements of the array are equal or not. If they are equal or not. If they are equal, it's impossible to make the array beautiful. If they are not equal, the largest element is placed first, ensuring the ugly condition is not satisfied by any element in the reordered array. **Since the array is sorted, the remaining elements will also not satisfy the ugly condition.** *Likert score: -1*

Figure I.2: The lemon-picking example. Likert scores are provided for points (2),(4) and (7), all other points are scored 2 for both models. Unlike GPT3.5, GPT4 notices that the input array is already sorted in non-decreasing order in the problem statement but it still fails to explain why moving the largest element to the first position makes the array beautiful by ignoring the crucial condition that all elements are strictly positive. (see text in bold - blue: correct, red: incorrect).

System Prompt: You are an expert in algorithm and programming.

User Prompt: Analyze the following competitive programming problem and its provided solution. Write an editorial to help students understand the problem-solving approach and strategy. Consider the problem's details, the solution's approach, and the idea behind it.

Problem: In this problem, we only consider strings consisting of lowercase English letters. Strings s and t are said to be isomorphic when the following conditions are satisfied:
* $|s| = |t|$ holds.
* For every pair i, j, one of the following holds:
* $s_i = s_j$ and $t_i = t_j$.
* $s_i \neq s_j$ and $t_i \neq t_j$.
For example, ‘abcac‘ and ‘zyxzx‘ are isomorphic, while ‘abcac‘ and ‘ppppp‘ are not. A string s is said to be in normal form when the following condition is satisfied: * For every string t that is isomorphic to s, $s \leq t$holds. Here $\leq$ denotes lexicographic comparison.
For example, ‘abcac‘ is in normal form, but ‘zyxzx‘ is not since it is isomorphic to ‘abcac‘, which is lexicographically smaller than ‘zyxzx‘.
You are given an integer N. Print all strings of length N that are in normal form, in lexicographically ascending order.
Constraints
* $1 \leq N \leq 10$
* All values in input are integers.
Input
Input is given from Standard Input in the following format:
N
Output
Assume that there are K strings of length N that are in normal form: $w_1, \ldots, w_K$ in lexicographical order. Output should be in the following format:
w_1 : w_K

Below is an accepted solution. Analyze it in the context of the problem.

```
n = list(map(int, input().split(' ')))[0]

def dfs(i, mx, n, res, cur = []):
  if i==n:
    res.append(''.join(cur[::]))
    return

  for v in range(0, mx + 1):
    dfs(i+1, mx+(1 if v==mx else 0), n , res, cur+[chr(v+ord('a'))])
res =[]
dfs(0,0,n,res)
for w in res: print(w)
```

Answer using the following format, with clear, detailed explanations. Use precise terms and avoid ambiguity. Use specific descriptions instead of vague expressions like "rearrange it" or "apply some operation". Specific instructions for each point are as follows in brackets"":
1). Problem Restatement: Understand every aspect of the problem first. Summarize the problem statement to remove the narrative/storytelling or thematic elements like characters or background story, abstract it into a formal statement while describing constraints, input-output specifications.
2). Step-by-Step Solution Explanation: Explain the solution code step-by-step in an algorithm level instead of explaining code line-by-line. Focus on the algorithm rather than implementation details like how to read input.
3). Solution Description: Based on the understanding of both the problem and the solution, describe the solution approach verbally. Explain the solution and the high-level reasoning behind it. Explain the WHY of the core algorithms/data structures/problem modeling.
4). Conceptual Evolution: Given all discussed above, describe how one would arrive at this solution. This can include how to analyze and approach the problem, the choices of type of algorithm used (e.g., dynamic programming, greedy, graph theory), the intuition behind the approach, and why this approach works for this problem. This is a narrative of the problem-solving journey.
5). Common Pitfalls: Pitfalls in the problem description OR common errors that students might make while attempting the problem OR corner/edge cases or offset handled in the solution.
6). Key to Solution:Use one sentence to illustrate the "aha!" steps (key idea or trick) in the solution. Be concise, specific and informative.

Figure I.3: **Explainer Prompt Example**: Example of the prompt we are using to generate Explanations from **Explainer** (GPT-4). The problem and solution is from Li et al. (2022).

System Prompt: You are an expert in algorithm and programming.

User Prompt: Given the algorithmic reasoning problem below, analyze it first, then develop a verbal description of the solution. Focus on the idea rather than the implementation details.
Problem: In this problem, we only consider strings consisting of lowercase English letters. Strings s and t are said to be isomorphic when the following conditions are satisfied:
* $|s| = |t|$ holds.
* For every pair i, j, one of the following holds:
* $s_i = s_j$ and $t_i = t_j$.
* $s_i \neq s_j$ and $t_i \neq t_j$.
For example, ‘abcac‘ and ‘zyxzx‘ are isomorphic, while ‘abcac‘ and ‘ppppp‘ are not. A string s is said to be in normal form when the following condition is satisfied: * For every string t that is isomorphic to s, $s \leq t$holds. Here $\leq$ denotes lexicographic comparison.
For example, ‘abcac‘ is in normal form, but ‘zyxzx‘ is not since it is isomorphic to ‘abcac‘, which is lexicographically smaller than ‘zyxzx‘.
You are given an integer N. Print all strings of length N that are in normal form, in lexicographically ascending order.
Constraints
* $1 \leq N \leq 10$
* All values in input are integers.
Input
Input is given from Standard Input in the following format:
N
Output
Assume that there are K strings of length N that are in normal form: $w_1, \ldots, w_K$ in lexicographical order. Output should be in the following format:
w_1 : w_K

Assistant Prompt Let's walk through the problem and think about how to solve it first.
Conceptual Evolution:
To arrive at this solution, one must understand the definition of isomorphic strings and normal form. The key insight is that a string in normal form will use the earliest possible character ('a') and only introduce a new character when it cannot use an existing one. This insight leads to a recursive generation of strings, where at each step, we either reuse an existing character or introduce the next character in the alphabet. The DFS algorithm is a natural fit for this recursive construction process.
Key to Solution:
The äha!štep is realizing that a string in normal form will always start with 'a' and introduce new characters sequentially only when necessary, which can be efficiently generated using a depth-first search.
Solution Description:
The solution uses a depth-first search (DFS) approach to generate all possible strings in normal form. It builds strings character by character, ensuring that each new character either matches a previous character or is the next unused character in the alphabet. This guarantees that the generated strings are in normal form and lexicographically minimal.
Common Pitfalls:
- Not understanding the definition of isomorphic strings and normal form could lead to incorrect string generation.
- Failing to ensure that the strings are generated in lexicographic order might result in an incorrect output sequence.
- Overlooking the base case in the recursive function could lead to infinite recursion or missing strings.
- Not considering the constraint that N is at most 10 could lead to inefficient solutions that do not scale.
Step-by-Step Solution Explanation:
- Read the integer N from the input.
- Define a recursive function ‘dfs‘ that will build the strings in normal form. This function takes the current position ‘i‘, the maximum character index ‘mx‘ used so far, the string length ‘n‘, a list ‘res‘ to store the results, and the current string ‘cur‘.
- If ‘i‘ equals ‘n‘, it means we have constructed a string of length ‘n‘. Add the current string ‘cur‘ to the results list ‘res‘ and return.
- Iterate over all possible next characters, represented by their index ‘v‘ from 0 to ‘mx + 1‘. The character to be added is determined by converting ‘v‘ to a character starting from 'a'.
- For each character, recursively call ‘dfs‘ with the next position ‘i + 1‘, updating ‘mx‘ to ‘mx + 1‘ if ‘v‘ equals ‘mx‘ (indicating that we are using a new character for the first time), and appending the new character to ‘cur‘.
- Initialize an empty list ‘res‘ to store the resulting strings.
- Call ‘dfs‘ starting from position 0, with ‘mx‘ set to 0 (since we start with 'a').
- After the recursive calls complete, print each string in ‘res‘.

Figure I.4: **Reasoner Prompt Example**: Example of the prompt we are using to Fine-tune the **Reasoner** (GPT-3.5) to generate explanations of the solution. The content from Assistant Prompt is the **Explainer**'s response to the query in Figure I.3

System Prompt: You are an expert in algorithm and programming.

User Prompt: Please solve the given algorithmic reasoning problem. You are also given a Hint on how to solve the problem. Analyze problem and hint, think carefully before implementing. The ultimate goal is to develop a complete Python 3 program to solve the problem. The program should be designed to read input and produce output in the specified format mentioned in problem. Enclose the code within triple backticks (“‘) for clarity. Keep in mind that this code is intended for direct submission to an online judge system, so don’t include test cases.
Here’s the Problem:
A subarray is a continuous part of array. Yarik recently found an array a of n elements and became very interested in finding the maximum sum of a non empty subarray. However, Yarik doesn’t like consecutive integers with the same parity, so the subarray he chooses must have alternating parities for adjacent elements.
For example, [1, 2, 3] is acceptable, but [1, 2, 4] is not, as 2 and 4 are both even and adjacent. You need to help Yarik by finding the maximum sum of such a subarray.
Input
The first line contains an integer $t(1 \leq t \leq 10^4)$— the number of test cases. Each test case is described as follows.
The first line of each test case contains an integer n ($1 \leq n \leq 210^5$)— length of the array.
The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($-10^3 \leq a_i \leq 10^3$)— elements of the array.
It is guaranteed that the sum of n for all test cases does not exceed 210^5.
Output
For each test case, output a single integer— the answer to the problem.

Here’s the Hint on how to solve the problem:
Let’s analyze the problem and think about how to solve it.
Conceptual Evolution:
To arrive at this solution, one might start by considering the brute force approach of checking all possible subarrays, which would be inefficient. Recognizing that the problem exhibits optimal substructure and overlapping subproblems, dynamic programming becomes a natural choice. The intuition is that the maximum sum of a subarray ending at index ‘i‘ can be extended by including ‘a[i]‘ if it maintains the alternating parity condition. This insight leads to the iterative update of ‘dp‘ and the tracking of the maximum sum.
Key to Solution:
The ”aha!” step is realizing that dynamic programming can be used to efficiently track the maximum sum of subarrays ending at each index, while maintaining the alternating parity condition.
Solution Description:
The solution uses dynamic programming to keep track of the maximum sum of subarrays ending at each index, while ensuring the alternating parity condition. The key insight is that the maximum sum of a subarray ending at index ‘i‘ can be derived from the maximum sum of a subarray ending at index ‘i-1‘ if the parity condition is satisfied. The ‘dp‘ array is used to store these intermediate results, and the ‘ans‘ variable is updated to keep track of the maximum sum found so far.
Common Pitfalls:
- Not considering the case where the array has only one element, which should be handled separately.
- Misunderstanding the alternating parity condition and not updating the ‘dp‘ array correctly.
- Forgetting to initialize the ‘dp‘ array with a value that does not affect the maximum sum (e.g., 0).
- Overlooking the possibility of negative numbers in the array, which could lead to incorrect updates of the ‘dp‘ array.
Step-by-Step Solution Explanation:
- Read the number of test cases ‘t‘.
- For each test case:
- Read the length of the array ‘n‘.
- Read the elements of the array ‘a‘.
- Initialize a dynamic programming (DP) array ‘dp‘ with a length of ‘n‘ and fill it with a large negative number (e.g., -10^9) to represent the maximum sum of a subarray ending at each index.
- Initialize a variable ‘ans‘ to 0, which will hold the maximum sum found.
- If the array has only one element, set ‘ans‘ to that element’s value.
- Otherwise, iterate over the array starting from the second element:
- If the current element and the previous element have different parities and they are not the first two elements of the array, update the ‘dp‘ array at the current index with the maximum of the current element and the sum of the current element and the ‘dp‘ value at the previous index.
- If the current element and the previous element have the same parity, update the ‘dp‘ array at the current index with the current element (as the previous subarray cannot be extended).
- Update ‘ans‘ with the maximum of its current value and the ‘dp‘ value at the current index.
- Print the value of ‘ans‘ for the current test case.

Figure I.5: **Coder Prompt Example**: Example of the prompt we are using to instruct the **Coder** to generate code given the problem and the verbal solution(also included in this figure). This is the full version for what’s in Table 4.5

CodeTree: Agent-guided Tree Search for Code Generation with Large Language Models

Qualitative Results

Here we show one real example using GPT-4o-mini to solve the problem: HumanEval-36. We keep the actual prompts and the model's responses in Figure I.6, I.7, and I.8. We find that the initial solution W_1 generated, which passes all visible test cases, is rejected by the critic agent, who also suggests improvements. Debugger follows critic agent's suggestion and implements W_2. We evaluate W_1 and W_2 on the hidden test cases, **where W_2 passes and W_1 fails**. This indicates that the Critic Agent is making correct judgments for false positive solutions (i.e., pass visible but fail on hidden test cases), and effectively guides to the correct solution.

In Figure I.6. The Thinker Agent decides to generate 5 strategies that potentially are the solutions. To start the tree search, the first strategy S_1 was sent to the solver agent to implement. After one solution is generated, it was sent to the Solver to be implemented. W_1 is immediately executed on visible test cases and the outputs are observed.

In Figure I.7, for a W_1 that passes all visible test cases, it will go to the critic agent to be further verified if W_1 can be applied to more general test cases. The Critic Agent gives a negative judgment and suggests improving this solution by considering more situations like zeros and negative integers. We use oracle hidden test cases to evaluate W_1, which is indeed incorrect, endorsing the decision made by Critic Agent.

In Figure I.8, the Debugger Agent will treat the judgment and reasoning from Critic Agent as additional environmental feedback, and then implement a new solution conditioned on: 1. the problem itself, 2. the solution to improve, 3. environmental feedback. Debugger Agent refines W_1 to W_2, which is accepted by the Critic Agent. We evaluate W_2 on hidden test cases; it is correct.

Thinker Agent (Strategy):

<system>:

You are an AI assistant that provides strategy for Python programmers to code. You will be given a function signature and its docstring by the user. Your goal is to think of multiple distinct strategies in English(Not Code) on how to approach this problem and solve it. The number of alternatives should be determined for this specific problem.

Your answer should be formatted as below:
List and Number your strategies line by line using "1. "; "2. "; "3. " and so on.
Describe each strategy with a FEW sentences in a SINGLE Line.

<user>:

[problem description]:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
```

<assistant>:

1. Start by understanding the problem requirements and constraints, focusing on the need to count occurrences of the digit '7' in specific integers.
2. Identify the range of integers to consider, which is from 0 to n-1, and clarify the conditions for divisibility by 11 or 13.
3. Create a loop to iterate through all integers from 0 to n-1, checking each integer for divisibility by 11 or 13.
4. For each integer that meets the divisibility condition, convert it to a string to facilitate counting occurrences of the digit '7'.
5. Use the string method `.count('7')` to count how many times '7' appears in the string representation of the integer.

Solver Agent:

<system>:

You are an AI that only responds with python code, NOT ENGLISH. You will be given a function signature and its docstring by the user. Write your full implementation (restate the function signature). Don't include test cases or printing statements in the code block.

<user>:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
```

To solve the problem, You can follow the hint given by an expert: Start by understanding the problem requirements and constraints, focusing on the need to count occurrences of the digit '7' in specific integers.

<assistant>:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
    count = 0
    for i in range(n):
        if i % 11 == 0 or i % 13 == 0:
            count += str(i).count('7')
    return count
```

Figure I.6: GPT-4o-mini as the Thinker and Solver Agents to solve HumanEval-36. The thinker agent generates 5 distinct strategies and the solver agent implements the first one. By oracle evaluation, the resulting solution can pass visible but fail on hidden test cases.

Critic Agent - Solution Verification:

<system>:

The user will provide a programming task along with a solution that passes all visible test cases. Your task is to further review the solution before it is judged against hidden test cases. Determine whether the solution is robust and general enough to pass unseen, valid test cases. Guideline:

1. Generalization Check: Verify that the solution uses general methods, avoiding hardcoding specific values or cases unless explicitly required. Confirm that the approach logically extends to unseen cases without special assumptions.
2. Boundary Check: Ensure all boundaries are correctly handled, including list indexing, loop start and end points, if-else conditions, and recursion exits. Look for potential off-by-one errors or boundary misses that could cause functional errors.
3. Edge Case Check: Confirm that the solution correctly handles valid edge/corner cases, such as zero, negative, empty, boundary values, or other special problem-specific situations. Note: All unseen test cases are guaranteed to follow stated data types, formats, conditions, and other constraints in the problem, no need to handle unallowed inputs. Do NOT apply redundant handling for cases that the current solution inherently manages, such as empty lists in sorting algorithms (` sorted([]) → []`), unless they explicitly fail (e.g., ` max([]) → error`).
4. Major Efficiency Check: Check if the solution is within polynomial time/space complexity. If not, fail this check.

Response Format:
Firstly, within several sentences, follow the guideline and briefly analyse.
On a new line, respond with "True" if the solution is ACCEPTABLE as-is, or "False" if NECESSARY modifications are required to handle unseen valid test cases.

<user>:

[Problem description]:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
```

[Implementation]:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
    # Start by understanding the problem requirements and constraints, focusing on the need to count occurrences of
the digit '7' in specific integers.
    count = 0
    for i in range(n):
        if i % 11 == 0 or i % 13 == 0:
            count += str(i).count('7')
    return count
```

[Execution Feedback on Visible Test Cases]:
Tested passed:

```
assert fizz_buzz(50) == 0
assert fizz_buzz(78) == 2
assert fizz_buzz(79) == 3
```

Tests failed:

<assistant>:

1. **Generalization Check**: The solution uses a loop to iterate through all integers less than ` n` and checks divisibility by 11 or 13, which is a general approach. It then counts occurrences of the digit '7' in these numbers, which is correctly implemented using ` str(i).count('7')` .
2. **Boundary Check**: The loop iterates from 0 to ` n-1` , which correctly covers all integers less than ` n` . There are no apparent off-by-one errors in the loop or conditions.
3. **Edge Case Check**: The solution should handle edge cases such as ` n = 0` or ` n = 1` . However, it does not handle negative inputs which could lead to incorrect results.
4. **Major Efficiency Check**: The solution runs in O(n) time complexity, which is efficient given the problem constraints. The space complexity is O(1) as it uses a constant amount of extra space.

False

Figure I.7: GPT-4o-mini as the Critic Agent to verify a solution of HumanEval-36 given by Solver Agent. It decides to reject this solution and suggest improvements.

Debugger Agent:

<system>:

You are an AI Python assistant. You will be given a program implementation to improve, a series of unit tests and execution feedback, and a hint to change the implementation appropriately. Write your full implementation (restate the function signature).

<user>:

[Implementation to Improve]:

```
def fizz_buzz(n: int):
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    """
    # Start by understanding the problem requirements and constraints, focusing on the need to count occurrences of
the digit '7' in specific integers.
    count = 0
    for i in range(n):
        if i % 11 == 0 or i % 13 == 0:
            count += str(i).count('7')
    return count
```

[Execution Feedback on Visible Test Cases]:

Tested passed:

```
assert fizz_buzz(50) == 0
assert fizz_buzz(78) == 2
assert fizz_buzz(79) == 3
```

Tests failed:

[AI-generated Feedback]:

1. **Generalization Check**: The solution uses a loop to iterate through all integers less than `n` and checks divisibility by 11 or 13, which is a general approach. It then counts occurrences of the digit '7' in these numbers, which is correctly implemented using `str(i).count('7')`.
2. **Boundary Check**: The loop iterates from 0 to `n-1`, which correctly covers all integers less than `n`. There are no apparent off-by-one errors in the loop or conditions.
3. **Edge Case Check**: The solution should handle edge cases such as `n = 0` or `n = 1`. However, it does not handle negative inputs which could lead to incorrect results.
4. **Major Efficiency Check**: The solution runs in O(n) time complexity, which is efficient given the problem constraints. The space complexity is O(1) as it uses a constant amount of extra space.

<assistant>:

Here is the improved implementation of the `fizz_buzz` function, incorporating the suggested changes:

```python
def fizz_buzz(n: int) -> int:
    """Return the number of times the digit 7 appears in integers less than n which are divisible by 11 or 13.
    >>> fizz_buzz(50)
    0
    >>> fizz_buzz(78)
    2
    >>> fizz_buzz(79)
    3
    >>> fizz_buzz(10)
    0
    >>> fizz_buzz(-5)
    0
    """
    if n <= 0:
        return 0

    count = 0
    for i in range(11, n):
        if i % 11 == 0 or i % 13 == 0:
            count += str(i).count('7')

    return count
```

Changes Made:

1. **Edge Case Handling**: Added a check for `n <= 0` to return 0 immediately for negative inputs and zero.
2. **Loop Starting Point**: The loop now starts from 11, as numbers less than 11 cannot be divisible by 11 or 13.
3. **Documentation**: Updated the docstring to clarify the function's behavior and added additional test cases for edge cases.

This implementation should now correctly handle all specified cases and edge conditions.

Figure I.8: GPT-4o-mini as the Debugger agent to refine a solution of HumanEval-36 given by Solver Agent. It refers to Critic Agent's suggestion and corrects the solution successfully.

Other Prompts

Critic Agent test case scoring, solution scoring, and Thinker Agent Reflection are not included in the above real case(HumanEval-36) solving; we present the detailed prompts used for these agents in Figure I.9.

AlgoSimBench: Identifying Algorithmically Similar Problems for Competitive Programming

Algorithm Tags

AlgoSimBench contains high-quality human-labeled algorithm tags. We show a portion of them with their subcategory and broad category labels in Figure I.10.

Prompts

MCQ prompt To evaluate models' performances in our experiments (i.e., Table 6.4), we apply a simple prompt for all models and methods, as shown in Figure I.11. The red-highlighted keywords correspond to our method as below("keyword"→method): "description"→ statement; "english solution"→ ASM-NL; "solution program"→ solution / ASM-PL. "problem abstract" → summary.

ASM prompt ASM methods requires an initial attempt to each problem given by an LLM. To ask the model to give an NL or PL attempted solution, or summarize the problem, we use the simple yet clear instruction:

Your goal is to give {Problem Restatement/Natural Language Solution/Solution Program} for a competitive programming problem. Requirements are enclosed in {}. You may include any thought process as you like, after which, the actual response should strictly follow the formats below:

Problem Restatement: {Translate and Abstract the problem into pure mathematical and formal descriptions, focus on the structure of problem rather than input/output/background story details.}

Critic Agent - Test Output Scoring:

<system>:
Your task is to evaluate the execution outputs of a code implementation. The statement and code is given by the user, and the output/expected output on a set of test cases. You should analyze the expected outputs and execution outputs. From a 0 to 5 range, give a score on how good the execution outputs are matching the expected ones (higher score means a better match). Standards are below:
0: Errors or time out when executing.
1: No pattern found when comparing pairs of <output, expected_output>, errors are hard to interpret.
2: Results abnormal for a part of cases(e.g., cannot handle negative elements; only half of it sorted).
3: Mismatches have clear patterns to interpret the error. For examples, all elements offset by 1; all elements + 1; reverse all elements etc.,
4: Lack consideration of edge condition/corner cases(e.g., error only when elements are equal), otherwise correct
5: Results exactly matched.

Your answer should be formatted as below:
In the first line, give your brief comparison.
In the second line, give A SINGLE INTEGER NUMBER as your final score(0 to 5)
<user>:
[problem]
[solution]
[execution results on visible test cases]

Critic Agent - Solution Scoring:

<system>: Your task is to evaluate a strategy and corresponding implementation for solving a programming problem. You should score from 1 to 5(higher means better) on how well do the solution implement the strategy and solve the task?

Your answer should be formatted as below:
In the first line, give your brief analysis.
In the second line, give A SINGLE INTEGER NUMBER as your final score(0 to 5)
<user>:
[problem]
[solution]
[execution results on visible test cases]

Thinker Agent Reflection:

<system>: You are a programming assistant. You goal is to help the user to correct their buggy code. You will be given an incorrect function implementation and a series of unit tests & execution results. There could be multiple ways to fix the error, you should provide reflection alternatives using various strategies. The number of reflection alternatives depends on the situation(e.g., if you are certain about where the bug is, you can provide only one). Each self-reflection should be complete and self-contained. If there are more than one bugs, they should be presented in one reflection rather than separately.

Your answer should be formatted as below:
List and Number your strategies line by line using "1. "; "2. "; "3. " and so on.
Describe each strategy with a FEW sentences in a SINGLE Line.

<user>:
[problem]
[solution]
[execution results on visible test cases]
[agent feedback] if any

Figure I.9: Prompts for Critic Agent test cases scoring, solution scoring, as well as Thinker Agent Reflection.

Number Theoretic Transform (NTT)	Convolution	Math
Prime Counting Function	Primes and Divisors	Number Theory
Merge Sort Tree	Segment Tree	Data Structures (DS)
Dynamic Aho Corasick	Aho Corasick	Strings
Finite Field Arithmetic Binary	Miscellaneous	Math
Queue Undo Trick	Techniques	Data Structures (DS)
Steiner Tree Problem	Minimum Spanning Tree (MST)	Graph Theory
Gaussian Elimination	Linear Algebra	Math
Halls Theorem	Matching	Graph Theory
Primitive Root	Modular Arithmetic	Number Theory
Linear Programming / Simplex Algorithm	Mathematical Optimization	Math
Suffix Tree	Suffix Structures	Strings
Lower bound on BIT	Binary Indexed Tree (BIT)	Data Structures (DS)
Difference Array	More Techniques	Basics
Minimum Diameter Spanning Tree	Minimum Spanning Tree (MST)	Graph Theory
DP Optimization using Data Structures	DP Optimizations	Dynamic Programming (DP)
Convex Hull(2D)	Geometry 2D	Geometry
Inverse of a Matrix modulo 2	Linear Algebra	Math
Binary Exponentiation and Basic Modular Arithmetic	Basic Modular Arithmetic	Basics
Euler Tour Tree (ETT)	Link Cut Tree (LCT)	Data Structures (DS)
Dynamic Connectivity Problem (Online) / HDLT Algorithm	Dynamic Connectivity	Data Structures (DS)
Cauchy–Binet formula	Linear Algebra	Math
Knapsack and Subset Sum Optimizations	Intro to DP	Dynamic Programming (DP)
Persistent Segment Tree with Lazy Propagation	Segment Tree	Data Structures (DS)
Link Cut Tree (LCT)	Link Cut Tree (LCT)	Data Structures (DS)
Big Integers	Bigint	Miscellaneous
Sparse Table	Sparse Table	Data Structures (DS)
POSET ft Dilworth's and Mirsky's Theorem	Partially Ordered Set (POSET)	Graph Theory
Extended Li Chao tree	Li Chao Tree	Dynamic Programming (DP)
Linear Diophantine Equation with Two Variables	Linear Diophantine Equations	Number Theory
Eulerian Path and Cycle	Eulerian Path	Graph Theory
Multiplicative Order and Carmichael's Lambda Function	Modular Arithmetic	Number Theory

Figure I.10: Examples of collected algorithmic tags with their sub-category and category labels.

Given a description of a competitive programming problem, your task is to identify **the most algorithmically similar** problem from a list of descriptions of candidate problems. From candidate descriptions, you should decide which has the most similar **topic, algorithmic tricks and ideas** with the given one, making two good references to each other to learn algorithms. For example, both are Lowest common ancestor(LCA) problems.
You can analyze these descriptions first if you'd like. At the end of your response, output your final choice as a single letter (A/B/C/D) on its own line.

[Description of Given Problem]:
For the multiset of positive integers s=\\{s_1,s_2,...,s_k\}, define the Greatest Common Divisor (GCD) and Least Common Multiple (LCM) of s as follow:
  * \gcd(s) is the maximum positive integer x, such that all integers in s are divisible on x.
  * lcm(s) is the minimum positive integer x, that divisible on all integers from s.
For example, \gcd(\{8,12\})=4,\gcd(\{12,18,6\})=6 and lcm(\{4,6\})=12. Note that for any positive integer x, \gcd(\\{x\})=lcm(\\{x\})=x.
Orac has a sequence a with length n. He come up with the multiset t=\{lcm(\\{a_i,a_j\})\ |\ i<j\}, and asked you to find the value of \gcd(t) for him. In other words, you need to calculate the GCD of LCMs of all pairs of elements in the given sequence.

[List of Description Candidates]

[**A**]. Description of the Problem:
Slime and his n friends are at a party. Slime has designed a game for his friends to play.
At the beginning of the game, the i-th player has a_i biscuits. At each second, Slime will choose a biscuit randomly uniformly among all a_1 + a_2 + … + a_n biscuits, and the owner of this biscuit will give it to a random uniform player among n-1 players except himself. The game stops when one person will have all the biscuits.
As the host of the party, Slime wants to know the expected value of the time that the game will last, to hold the next activity on time.
For convenience, as the answer can be represented as a rational number p/q for coprime p and q, you need to find the value of (p · q^{-1})mod 998 244 353. You can prove that qmod 998 244 353 ≠ 0.

[**B**]. Description of the Problem:
Given is a positive integer N. Consider repeatedly applying the operation below on N:
* First, choose a positive integer z satisfying all of the conditions below:
* z can be represented as z=p^e, where p is a prime number and e is a positive integer;
* z divides N;
* z is different from all integers chosen in previous operations.
* Then, replace N with N/z.
Find the maximum number of times the operation can be applied.

[**C**]. Description of the Problem:
Johnny's younger sister Megan had a birthday recently. Her brother has bought her a box signed as "Your beautiful necklace — do it yourself!". It contains many necklace parts and some magic glue.
The necklace part is a chain connecting two pearls. Color of each pearl can be defined by a non-negative integer. The magic glue allows Megan to merge two pearls (possibly from the same necklace part) into one. The beauty of a connection of pearls in colors u and v is defined as follows: let 2^k be the greatest power of two dividing u ⊕ v — [exclusive or](https://en.wikipedia.org/wiki/Exclusive_or#Computer_science) of u and v. Then the beauty equals k. If u = v, you may assume that beauty is equal to 20.
Each pearl can be combined with another at most once. Merging two parts of a necklace connects them. Using the glue multiple times, Megan can finally build the necklace, which is a cycle made from connected necklace parts (so every pearl in the necklace is combined with precisely one other pearl in it). The beauty of such a necklace is the minimum beauty of a single connection in it. The girl wants to use all available necklace parts to build exactly one necklace consisting of all of them with the largest possible beauty. Help her!

[**D**]. Description of the Problem:
In this problem MEX of a certain array is the smallest positive integer not contained in this array.
Everyone knows this definition, including Lesha. But Lesha loves MEX, so he comes up with a new problem involving MEX every day, including today.
You are given an array a of length n. Lesha considers all the non-empty subarrays of the initial array and computes MEX for each of them. Then Lesha computes MEX of the obtained numbers.
An array b is a subarray of an array a, if b can be obtained from a by deletion of several (possible none or all) elements from the beginning and several (possibly none or all) elements from the end. In particular, an array is a subarray of itself.

Lesha understands that the problem is very interesting this time, but he doesn't know how to solve it. Help him and find the MEX of MEXes of all the subarrays!

Figure I.11: MCQ End2End Prompt with an example from the dataset. (Test cases in problems are omited for presentation.) Blue text is the pre-defined prompt, and red text is highlighted keywords to be replaced for different methods(i.e., ASM/Statement/Summary/Solution).

Natural Language Solution: {*Analyze and Describe how to approach and solve this problem using English and equations if needed. (E.g., This problem is to find.... it's a classic subset sum problem, ... use 0-1 knapsack... we first...)*}

Solution Program: {*Implement the Python solution. Wrap your code with "'.*}

To experiment with ICL, we use the original prompts from the USACO work Shi et al. (2024).

Glossary

Jierui Li A hard-working graduate student, nearing the end of this chapter of your education!

Doctor of Philosophy Jierui Li is about to earn this. Congratulations!

Works Cited

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=G7sIFXugTX.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale, 2026. URL https://arxiv.org/abs/2602.23866.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022. URL https://arxiv.org/abs/2212.08073.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643757. URL `https://doi.org/10.1145/3643757`.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=ByldLrqlx`.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. Iterative refinement of project-level code context for precise code generation with compiler feedback. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2336–2353, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.138. URL `https://aclanthology.org/2024.findings-acl.138/`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Rina Diane Caballar and Cole Stryker. What is reasoning in ai?, 2025.

Pedro Calais, Gabriel Franco, Zilu Tang, Themistoklis Nikas, Wagner Meira Jr, Evimaria Terzi, and Mark Crovella. Disentangling text and math in word

problems: Evidence for the bidimensional structure of large language models' reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 12671–12688, 2025.

Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. Bigo(bench) – can llms generate code with controlled time and space complexity?, 2025. URL `https://arxiv.org/abs/2503.15242`.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*, 2023a.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022a.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation, 2025a. URL `https://arxiv.org/abs/2402.03216`.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *arXiv preprint arXiv:2405.20092*, 2024.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse:

Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023b.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2022b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023c.

Yiqun Chen, Lingyong Yan, Weiwei Sun, Xinyu Ma, Yi Zhang, Shuaiqiang Wang, Dawei Yin, Yiming Yang, and Jiaxin Mao. Improving retrieval-augmented generation through multi-agent reinforcement learning, 2025b. URL `https://arxiv.org/abs/2501.15228`.

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization, 2025c. URL `https://arxiv.org/abs/2503.09089`.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022. URL `https://arxiv.org/abs/2009.14794`.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling

instruction-finetuned language models, 2022. URL https://arxiv.org/abs/2210.11416.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020. URL https://arxiv.org/abs/2003.10555.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL https://arxiv.org/abs/1901.02860.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang,

Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025a. URL `https://arxiv.org/abs/2501.12948`.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang,

Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025b. URL `https://arxiv.org/abs/2412.19437`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=wgDcbBMSfh.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. Kto: Model alignment as prospect theoretic optimization, 2024. URL https://arxiv.org/abs/2402.01306.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Jingsheng Gao, Linxu Li, Ke Ji, Weiyuan Li, Yixin Lian, yuzhuo fu, and Bin Dai. SmartRAG: Jointly learn RAG-related tasks from the environment feedback. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OCd3cffulp.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1762–1777, Toronto, Canada, July 2023. Association for Computational Linguistics.

doi: 10.18653/v1/2023.acl-long.99. URL https://aclanthology.org/2023.acl-long.99/.

Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners, 2021a. URL https://arxiv.org/abs/2012.15723.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, November 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL https://aclanthology.org/2021.emnlp-main.552/.

Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners. In Kate Larson, editor, *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 6270–6278. International Joint Conferences on Artificial Intelligence Organization, 8 2024. doi: 10.24963/ijcai.2024/693. URL https://doi.org/10.24963/ijcai.2024/693. Main Track.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

Daya Guo, Shuo Ren, Suyu Lu, Junjie Feng, Duyu Tang, Nan Duan, Ming Zhou, and Sining Liu. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2021.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Dhruv Gupta, Gayathri Ganesh Lakshmy, and Yiqing Xie. Sacl: Understanding and combating textual bias in code retrieval with semantic-augmented reranking and localization, 2025. URL `https://arxiv.org/abs/2506.20081`.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: Retrieval-augmented language model pre-training, 2020. URL `https://arxiv.org/abs/2002.08909`.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024. URL `https://arxiv.org/abs/2310.19923`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL `https://openreview.net/forum?id=sD93GOzH3i5`.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes, 2023.

Wenyue Hua and Yongfeng Zhang. System 1 + system 2 = better world: Neural-symbolic chain of logic reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 601–612, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL `https://aclanthology.org/2022.findings-emnlp.42`.

Wenyue Hua, Kaijie Zhu, Lingyao Li, Lizhou Fan, Shuhang Lin, Mingyu Jin, Haochen Xue, Zelong Li, JinDong Wang, and Yongfeng Zhang. Disentangling logic: The role of context in large language model reasoning capabilities, 2024. URL `https://arxiv.org/abs/2406.02787`.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui.

Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023a.

Jerry Huang, Siddarth Madala, Risham Sidhu, Cheng Niu, Hao Peng, Julia Hockenmaier, and Tong Zhang. Rag-rl: Advancing retrieval-augmented generation via rl and curriculum learning, 2025. URL `https://arxiv.org/abs/2503.12759`.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and Weizhu Chen. Competition-level problems are effective llm evaluators, 2023b.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL `https://arxiv.org/abs/2409.12186`.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL `https://openreview.net/forum?id=LtJMqnbslJe`.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning, 2022. URL https://arxiv.org/abs/2112.09118.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.482. URL https://aclanthology.org/2021.emnlp-main.482/.

Pengcheng Jiang, Jiacheng Lin, Lang Cao, Runchu Tian, SeongKu Kang, Zifeng Wang, Jimeng Sun, and Jiawei Han. Deepretrieval: Hacking real search engines and retrievers with large language models via reinforcement learning. *arXiv preprint arXiv:2503.00223*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and

Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*, 2024.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Daria Kryvosheieva, Saba Sturua, Michael Günther, Scott Martens, and Han Xiao. Efficient code embeddings from code generation models, 2025. URL https://arxiv.org/abs/2508.21290.

Mandar Kulkarni, Praveen Tangarajan, Kyung Kim, and Anusua Trivedi. Reinforcement learning for optimizing rag for domain chatbots, 2024. URL https://arxiv.org/abs/2401.06800.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=vYhglxSj8j.

Tien P. T. Le, Anh M. T. Bui, Huy N. D. Pham, Alessio Bucaioni, and Phuong T. Nguyen. When retriever meets generator: A joint model for code comment generation, 2025. URL https://arxiv.org/abs/2507.12558.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. NV-embed: Improved techniques for training LLMs as generalist embedding models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=lgsyLSsDRe.

Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing code explanations created by students and large language models. *CoRR*, abs/2304.03938, 2023. doi: 10.48550/arXiv.2304.03938. URL https://doi.org/10.48550/arXiv.2304.03938.

Hector J. Levesque, Ernest Davis, and L. Morgenstern. The winograd schema challenge. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011. URL https://api.semanticscholar.org/CorpusID:15710851.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2020a.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020b. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. *CoRR*, abs/2206.14858, 2022. doi: 10.48550/arXiv.2206.14858. URL https://doi.org/10.48550/arXiv.2206.14858.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36: 51991–52008, 2023a.

Jierui Li and Raymond Mooney. Distilling algorithmic reasoning from llms via explaining solution programs, 2024. URL https://arxiv.org/abs/2404.08148.

Jierui Li and Raymond Mooney. Algosimbench: Identifying algorithmically similar problems for competitive programming, 2025. URL https://arxiv.org/abs/2507.15378.

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. Explaining competitive-level programming solutions using llms, 2023b.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. CodeTree: Agent-guided tree search for code generation with large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3711–3726, Albuquerque, New Mexico, April 2025a. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.189. URL https://aclanthology.org/2025.naacl-long.189/.

Mingjie Li, Yuan-Gen Wang, Peng Zhang, Hanpin Wang, Lisheng Fan, Enxia Li, and Wei Wang. Deep learning for approximate nearest neighbour search: A survey and future directions. *IEEE Transactions on Knowledge and Data Engineering*, 35(9):8997–9018, 2023c. doi: 10.1109/TKDE.2022.3220683.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023d.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models, 2025b. URL https://arxiv.org/abs/2407.02883.

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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli,

Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

Zhuofeng Li, Haoxiang Zhang, Cong Wei, Pan Lu, Ping Nie, Yi Lu, Yuyang Bai, Shangbin Feng, Hangxiao Zhu, Ming Zhong, Yuyu Zhang, Jianwen Xie, Yejin Choi, James Zou, Jiawei Han, Wenhu Chen, Jimmy Lin, Dongfu Jiang, and Yu Zhang. Beyond semantic similarity: Rethinking retrieval for agentic search via direct corpus interaction, 2026. URL https://arxiv.org/abs/2605.05242.

Victoria Lin, Xilun Chen, Mingda Chen, Weijia Shi, Maria Lomeli, Richard James, Pedro Rodriguez, Jacob Kahn, Gergely Szilvasy, Mike Lewis, Luke Zettlemoyer, and Scott Yih. Ra-dit: Retrieval-augmented dual instruction tuning. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 19138–19162, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/536d18fbb454f80221465f1a42c6f389-Paper-Conference.pdf.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning, 2024a. URL https://arxiv.org/abs/2402.09664.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 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*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=pPjZIOuQuF.

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval, 2024c. URL https://arxiv.org/abs/2411.12644.

Ye Liu, Rui Meng, Shafiq Joty, silvio savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. CodeXEmbed: A generalist embedding model family for multilingual and multi-task code retrieval. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=z3lG70Azbg.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

Hanzhen Lu and Zhongxin Liu. Improving retrieval-augmented code comment generation by retrieving for generation, 2024. URL https://arxiv.org/abs/2408.03623.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning, 2023.

YINGWEI MA, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At which training stage does code data help LLMs reasoning? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KIPJKST4gw.

Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from using code explanations generated by large language models in a web software development e-book. In Maureen Doyle, Ben Stephenson, Brian Dorn, Leen-Kiat Soh, and Lina Battestilli, editors, *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, pages 931–937. ACM, 2023. doi: 10.1145/3545945.3569785. URL https://doi.org/10.1145/3545945.3569785.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. MTEB: Massive text embedding benchmark. In Andreas Vlachos and Isabelle Augenstein, editors, *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 2014–2037, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.eacl-main.148. URL https://aclanthology.org/2023.eacl-main.148/.

Nebius AI. SWE-agent-trajectories: 80,036 software engineering agent trajectories. Hugging Face dataset, 2024. URL https://huggingface.co/datasets/nebius/SWE-agent-trajectories.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023a.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2023b.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. `https://openai.com/blog/chatgpt`, 2023a.

OpenAI. Gpt-4 technical report, 2023b.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022. URL `https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf`.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL `https://arxiv.org/abs/2412.21139`.

Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. Evaluating in-context learning of libraries for code generation, 2024. URL https://arxiv.org/abs/2311.09635.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/pei23a.html.

Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *CoRR*, abs/1802.04335, 2018. URL http://arxiv.org/abs/1802.04335.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.

Jin Qin, Zihan Liao, Ziyin Zhang, Hang Yu, Peng Di, and Rui Wang. C2llm technical report: A new frontier in code retrieval via adaptive cross-attention pooling, 2025. URL https://arxiv.org/abs/2512.21332.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Preprint.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *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 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410/.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering, 2024.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Stephen E. Robertson, Susan Walker, S. Jones, Micheline M. Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3: Retrieving via probabilistic retrieval. *NIST Special Publication*, 500-225:109–123, 1995.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019. URL `https://arxiv.org/abs/1907.10641`.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020. URL `https://arxiv.org/abs/1910.01108`.

Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=9Vrb9D0WI4`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh*

Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations, 2018. URL https://arxiv.org/abs/1803.02155.

Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kGa4fMtP9l.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.231. URL https://aclanthology.org/2022.emnlp-main.231.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning, 2023.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. Nofuneval: Funny how code LMs falter on requirements beyond functional correctness. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=h5umhm6mzj.

Jialin Song, Jonathan Raiman, and Bryan Catanzaro. Effective large language model debugging with best-first tree search. *arXiv preprint arXiv:2407.19055*, 2024.

Yueqi Song, Ketan Ramaneti, Zaid Sheikh, Ziru Chen, Boyu Gou, Tianbao Xie, Yiheng Xu, Danyang Zhang, Apurva Gandhi, Fan Yang, et al. Agent data protocol: Unifying datasets for diverse, effective fine-tuning of LLM agents, 2025. URL https://arxiv.org/abs/2510.24702.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 3008–3021. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f
pdf.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1102–1121, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.71. URL https://aclanthology.org/2023.findings-acl.71/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023b. URL https://arxiv.org/abs/2104.09864.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for BERT model compression. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *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 4323–4332, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1441. URL https://aclanthology.org/D19-1441/.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025. URL https://arxiv.org/abs/2412.01007.

Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. CodeScout: An effective recipe for reinforcement learning of code search agents, 2026. URL https://arxiv.org/abs/2603.17829.

Ruixiang Tang, Dehan Kong, Longtao Huang, and Hui Xue. Large language models can be lazy learners: Analyze shortcuts in in-context learning. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, 2023a. doi: 10.18653/v1/2023.findings-acl.284. URL http://dx.doi.org/10.18653/v1/2023.findings-acl.284.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. Large language models are in-context semantic reasoners rather than symbolic reasoners, 2023b. URL https://arxiv.org/abs/2305.14825.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Andrew Trotman, Antti Puurula, and Blake Burgess. Improvements to bm25 and language models examined. In *Proceedings of the 19th Australasian Document Computing Symposium*, pages 58–65, 2014.

Marco Valentino, Leonardo Ranaldi, Giulia Pucci, Federico Ranaldi, and Andre Freitas. Semeval-2026 task 11: Disentangling content and formal reasoning in language models, 2024. URL https://sites.google.com/view/semeval-2026-task-11.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2019. URL https://arxiv.org/abs/1807.03748.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15362–15376, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.859. URL https://aclanthology.org/2024.emnlp-main.859/.

Boshi Wang, Xiang Deng, and Huan Sun. Iteratively prompt pre-trained language models for chain of thought. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2714–2730, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.174.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training, 2024a. URL https://arxiv.org/abs/2212.03533.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Improving text embeddings with large language models, 2024b. URL https://arxiv.org/abs/2401.00368.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan Boyd-Graber,

and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL https://aclanthology.org/2023.acl-long.754/.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023c.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-bench: Can retrieval augment code generation? In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.176. URL https://aclanthology.org/2025.findings-naacl.176/.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago S. F. X. Teixeira, Diyi Yang, Ke Wang, and Alex Aiken. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking, 2025. URL https://arxiv.org/abs/2502.12466.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022a. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large

language models. *arXiv preprint arXiv:2201.11903*, 2022b.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hH36JeQZDaO.

Orion Weller, Benjamin Chang, Sean MacAvaney, Kyle Lo, Arman Cohan, Benjamin Van Durme, Dawn Lawrie, and Luca Soldaini. Followir: Evaluating and teaching information retrieval models to follow instructions, 2024. URL https://arxiv.org/abs/2403.15246.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Jing Xiong, Zixuan Li, Chuanyang Zheng, Zhijiang Guo, Yichun Yin, Enze Xie, Zhicheng Yang, Qingxing Cao, Haiming Wang, Xiongwei Han, Jing Tang, Chengming Li, and Xiaodan Liang. Dq-lore: Dual queries with low rank approximation re-ranking for in-context learning, 2026. URL https://arxiv.org/abs/2310.02954.

Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. A survey on knowledge distillation of large language models, 2024. URL https://arxiv.org/abs/2402.13116.

Shi-Qi Yan and Zhen-Hua Ling. Rpo: Retrieval preference optimization for robust retrieval-augmented generation, 2025. URL https://arxiv.org/abs/2501.13726.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual*

Conference on Neural Information Processing Systems, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench Multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=riTiq3i21b.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents, 2025b. URL https://arxiv.org/abs/2504.21798.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow, 2018.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In

Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623316. URL https://doi.org/10.1145/3597503.3623316.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. Mammoth: Building math generalist models through hybrid instruction tuning, 2023.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: A (de-)compositional framework for algorithmic reasoning with language models, 2023.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=vfzRRjumpX.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor,

Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore, December 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

Fuwei Zhang, Yanzhao Zhang, Mingxin Li, Dingkun Long, Lexiang Hu, Pengjun Xie, Zhao Zhang, and Fuzhen Zhuang. CORE-Bench: A comprehensive benchmark for code retrieval in the era of agentic coding, 2026. URL https://arxiv.org/abs/2606.11864.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*, 2023b.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with llm-generated oracle verifiers, 2023c.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swebench goes live!, 2025a. URL https://arxiv.org/abs/2505.23419.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR, 2023d.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025b. URL https://arxiv.org/abs/2506.05176.

Ziyin Zhang, Zihan Liao, Hang Yu, Peng Di, and Rui Wang. F2llm technical report: Matching sota embedding performance with 6 million open-source data, 2025c. URL https://arxiv.org/abs/2510.02294.

Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. Large language models are not robust multiple choice selectors, 2024. URL https://arxiv.org/abs/2309.03882.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.49. URL https://aclanthology.org/2024.findings-acl.49/.

Wanjun Zhong, Tingting Ma, Jiahai Wang, Jian Yin, Tiejun Zhao, Chin-Yew Lin, and Nan Duan. Disentangling reasoning capabilities from language models with compositional reasoning transformers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 7587–7600, 2023.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WZH7099tgfM.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence, 2022. URL https://arxiv.org/abs/2206.08474.

Xuekai Zhu, Biqing Qi, Kaiyan Zhang, Xingwei Long, and Bowen Zhou. Pad: Program-aided distillation specializes large models in reasoning, 2023.

Vita

Jierui Li is going to graduate soon!

Address: jierui@cs.utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.