AlgoSimBench: Identifying Algorithmically Similar Problems for Competitive Programming

Jierui Li , **Raymond Mooney**
The University of Texas at Austin
{jierui, mooney}@cs.utexas.edu

Abstract

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, 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.

Code — https://github.com/lijierui/AlgoSimBench
Datasets —
https://huggingface.co/datasets/JerryL/AlgoSimBench
Extended version — https://arxiv.org/abs/2507.15378

1 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 paper, 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 1). This notion of similarity goes beyond superficial textual resemblance; it lies in the 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. 5.1). AlgoSimBench can encourage the development of better LLMs to reason independently of superficial textual 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

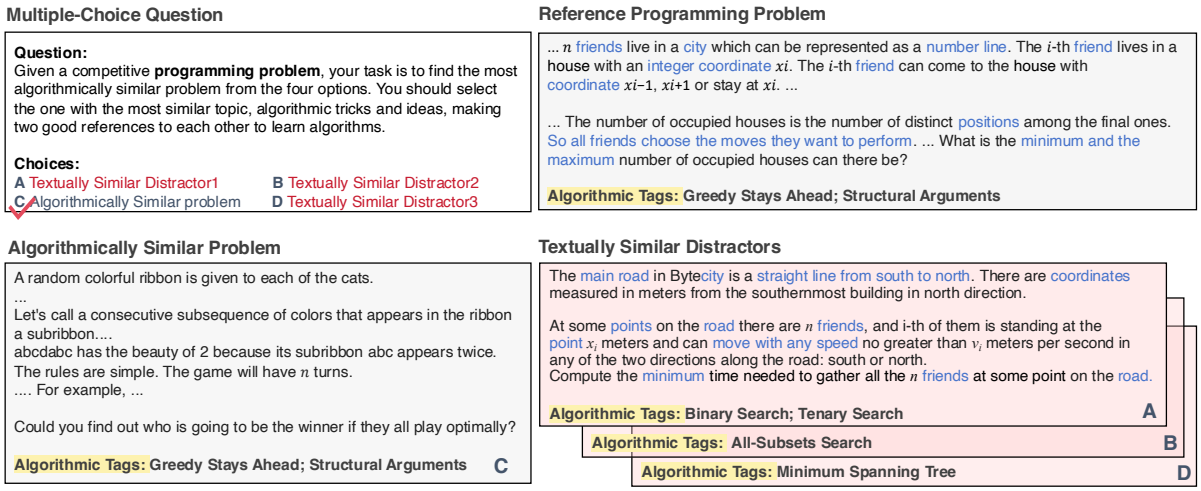


Figure 1: An example from AlgoSimBench, with one algorithmically similar problem and a textually similar distractor as an example. The blue-highlighted text makes the reference problem and the distractor look similar.

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.

2 Related Works

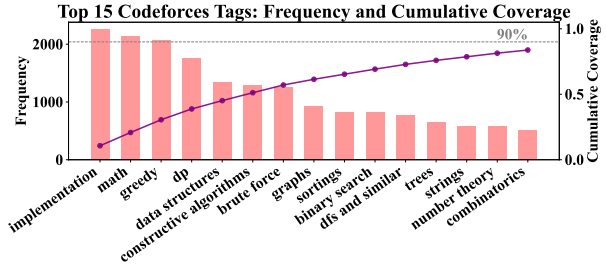
Solving Competitive Programming Problems with LLMs. Competitive programming presents a significantly more challenging landscape than general code generation tasks [Chen *et al.*, 2021; Austin *et al.*, 2021; Liu *et al.*, 2023], as it often demands deep algorithmic reasoning, problem decomposition, and planning before implementation. Li et al.[2022] first framed competitive programming as a benchmark for evaluating language-and-code models and it was later advocated as a way to evaluate LLMs by Huang et al. [2024]. CodeLMs usually pre-train and post-train LLMs on real or synthesized code data, relying on supervised fine-tuning [Rozière *et al.*, 2024; Guo *et al.*, 2024; Hui *et al.*, 2024] or policy optimization with program correctness as the reward [DeepSeek-AI *et al.*, 2025a]. Multi-agent frameworks [Islam *et al.*, 2024; Li *et al.*, 2025] assign separate agents to reasoning, implementation, and debugging, reflecting how humans tackle problems step by step. Shi et al.[2024] shows that selecting exemplar problems and solutions from top human coders for ICL provides significant gains, suggesting that good ICL examples can help LLMs better solve programming problems.

Code/Text Retrieval and Code Similarity. Early code similarity detectors relied on lexical overlap and syntax. Tools like MOSS [Schleimer *et al.*, 2003] used token matching to flag plagiarism. However, lexical and syntax match could be easily evaded by superficial code modifications [Zakeri-Nasrabadi *et al.*, 2023]. To overcome these limitations, semantic code search techniques emerged [Gu *et al.*, 2018; Feng *et al.*, 2020]. Suresh et al. [2025] construct code retrieval contrastive datasets through mining hard negatives. However, Wang et al. [2025] has found that retrievers struggle to fetch relevant and helpful context for code generation. Wei et al.[2025] showed that LLMs can perform poorly on the deceptively simple task of determining functional equivalence between two programs. Unlike prior works on code similarity studying snippet-level functionality (e.g., sort an array of floats), AlgoSimBench focuses on higher-level logical similarity, which lies in the problem structure and potential solutions.

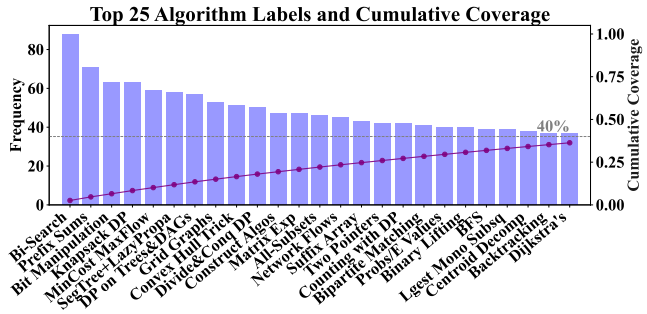
Disentangling Logical Reasoning in Context. Disentangling semantic understanding from logical or formal/symbolic reasoning has been explored across a range of tasks and domains. Zhong et al.[2023] separated semantic representation from multi-step reasoning by decomposing general QA into two dedicated 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.

3 AlgoSimBench Dataset

AlgoSimBench is a challenging benchmark, designed to evaluate LLMs' algorithmic-reasoning ability beyond code gener-



(a) Top 15 Codeforces broad algorithm tags.



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

Figure 2: Comparison between Codeforces tag distribution and our fine-grained tag distribution.

ation, 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.

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 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.

3.2 Adversarial Task Design

As Jain et al. and Wei et al.[2021; 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.*

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.*, 2020].

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 by Shi et al.[2024].

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

	Statement	Solution
BM25 [Robertson *et al.*, 1995]	12.7	32.6
BART [Lewis *et al.*, 2020]	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.*, 2024]	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 1: AlgoSimBench MCQ Retrieval Accuracies (%) across different retrieval methods. Cosine similarity is used for all dense models.

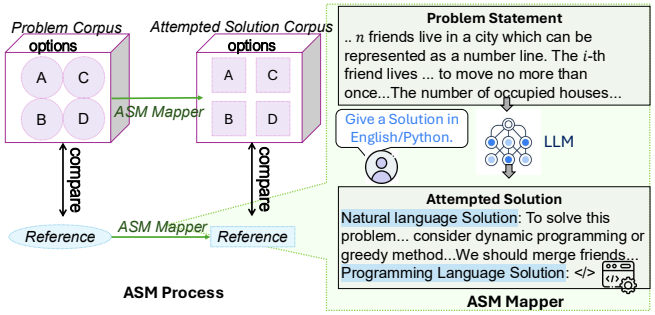


Figure 3: Illustration of Attempted Solution Matching. Both the given reference problem (reference) and options (corpus of problems) are mapped to attempted solutions by instructing an LLM to solve the problems.

sparse retrieval methods were tested using this setting. We use cosine similarity to calculate the similarity scores. As shown in Table 1, 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 3 illustrates the approach.

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.

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.

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.

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.*, 2020], code embedding models GraphCodeBert [Guo *et al.*, 2021], CodeSage-v2 [Zhang *et al.*, 2024], SFR-Code-400M_R [Liu *et al.*, 2024], Jina-Code-V2 [Günther *et al.*, 2024], CodeRank Embedding [Suresh *et al.*, 2025].

[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/

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 2: 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 $p < 0.01$.

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.

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.

Main Results

MCQ accuracies of all seven models using End-to-End selection are given in Table 2. We found that none of the models perform particularly well when only given the original problem Statements. The weaker model *Llama3.1-8B-Instruct* has near-random performance across all methods. Reasoning LLMs like Deepseek-R1 and o3-mini perform better than other models, achieving accuracies around 60%. Given oracle solutions (**Solution**) instead of problem statements, LLMs can identify algorithmic similarity much better, yielding improvements as high as 26.3%.

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.

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?

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.

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 3: 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.

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

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 3, 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.

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 4, 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.

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

5.3 Retrieval-Based Selection

In this setting, we treat AlgoSimBench as a retrieval problem as discussed in section 5.1. Results for retrieval based on problem statements were shown in Table 1. 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 5.

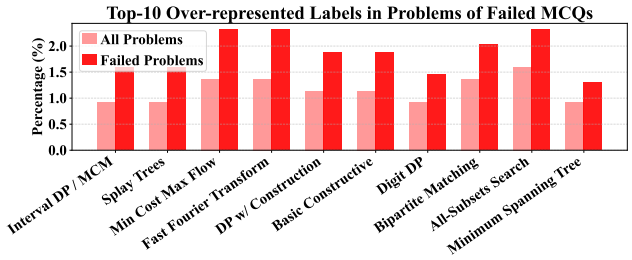
ASM-NL again outperforms other methods. Unlike retrieval based on statements (see Table 1), using a Summary is better than random guessing. Since AlgoSimBench is constructed so that the textual problem-statement similarity is a misleading indicator of ASPs, summarization is helpful but still provides 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 core idea, rather than a full implementation.

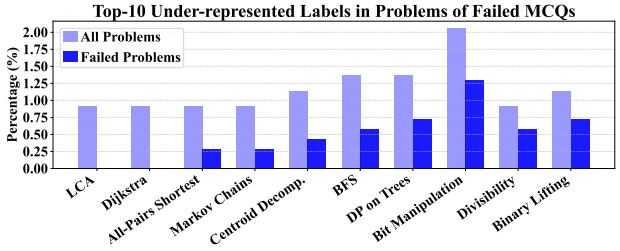
Although Retrieval-Based Selection performs significantly worse than End-to-End Selection, it scales much better to retrieving ASPs from a large corpus. End-to-end selection requires fitting all potential retrievals into an LLM input window and is difficult to scale beyond choosing from only four options 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.*, 2023].

5.4 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



(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 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.

	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 5: 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 1.

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

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 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 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.*, 2023; Patel *et al.*, 2024].

6 Conclusion and Discussion

This paper 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.

References

[Austin *et al.*, 2021] 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.

[Calais *et al.*, 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.

[Chen *et al.*, 2021] 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.

[DeepSeek-AI *et al.*, 2025a] 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, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[DeepSeek-AI *et al.*, 2025b] 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, et al. Deepseek-v3 technical report, 2025.

[Devlin *et al.*, 2019] 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.

[Feng *et al.*, 2020] 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.

[Gao *et al.*, 2023] 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.

[Gu *et al.*, 2018] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th international conference on software engineering*, pages 933–944, 2018.

[Gu *et al.*, 2024] 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.

[Guo *et al.*, 2021] 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.

[Guo *et al.*, 2024] 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.

[Günther *et al.*, 2024] Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, and Mohammad Kalim Akram others. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024.

[Hua *et al.*, 2024] 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.

[Huang *et al.*, 2024] 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, 2024.

[Hui *et al.*, 2024] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, and Bowen Yu others. Qwen2.5-coder technical report, 2024.

[Islam *et al.*, 2024] 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.

[Izacard *et al.*, 2022] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning, 2022.

[Jain *et al.*, 2021] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In *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.

[Jimenez *et al.*, 2024] 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.

[Levesque *et al.*, 2011] Hector J. Levesque, Ernest Davis, and L. Morgenstern. The winograd schema challenge. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011.

[Lewis *et al.*, 2020] 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*, 2020.

[Li *et al.*, 2022] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[Li *et al.*, 2023] 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, 2023.

[Li *et al.*, 2025] 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 (Volume 1: Long Papers)*, pages 3711–3726, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.

[Liu *et al.*, 2023] 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.

[Liu *et al.*, 2024] 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, 2024.

[Patel *et al.*, 2024] Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. Evaluating in-context learning of libraries for code generation, 2024.

[Robertson *et al.*, 1995] 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.

[Rozière *et al.*, 2024] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code, 2024.

[Sakaguchi *et al.*, 2019] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019.

[Schleimer *et al.*, 2003] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

[Shi *et al.*, 2022] 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.

[Shi *et al.*, 2024] Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? In *First Conference on Language Modeling*, 2024.

[Sparck Jones, 1988] Karen Sparck Jones. *A statistical interpretation of term specificity and its application in retrieval*, page 132–142. Taylor Graham Publishing, GBR, 1988.

[Suresh *et al.*, 2025] 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.

[Tang *et al.*, 2023] 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, 2023.

[Trotman *et al.*, 2014] 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.

[Valentino *et al.*, 2024] Marco Valentino, Leonardo Ranaldi, Giulia Pucci, Federico Ranaldi, and Andre Freitas. Semeval-2026 task 11: Disentangling content and formal reasoning in language models, 2024.

[Wang *et al.*, 2025] 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?, 2025.

[Wei *et al.*, 2025] 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.

[Xiong *et al.*, 2026] 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.

[Zakeri-Nasrabadi *et al.*, 2023] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, 204:111796, 2023.

[Zhang *et al.*, 2024] 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.

[Zheng *et al.*, 2024] Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. Large language models are not robust multiple choice selectors, 2024.

[Zhong *et al.*, 2023] 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.

A Appendix

A.1 Dataset Sources & Statistics

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 7. 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.

	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 7: 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 8.

A.2 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 5.

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

Table 8: Statistics of MCQs in AlgoSimBench.

A.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 position: A, B, C or D. The results in Figure 6 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 9

A.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 7.

A.5 Inference Time

We also estimate the efficiency of different methods by evaluating the total number of inference tokens needed to obtain the final data. The results are in Table 10, 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.

A.6 Experimental Settings

Prompts

MCQ prompt To evaluate models' performances in our experiments (i.e., Table 2), we apply a simple prompt for all models and methods, as shown in Figure 8. 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:

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 5: Examples of collected algorithmic tags with their sub-category and category labels.

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 9: Performance under different random seeds for MCQ option layout.

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 10: 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.

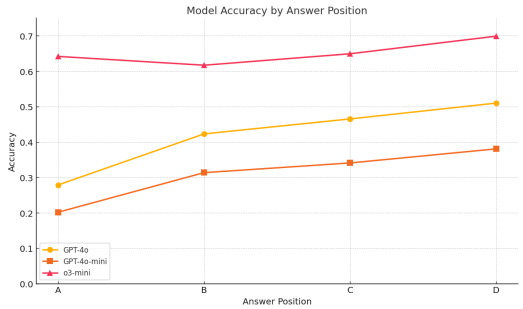


Figure 6: MCQ accuracy if the correct option is fixed to a particular position.

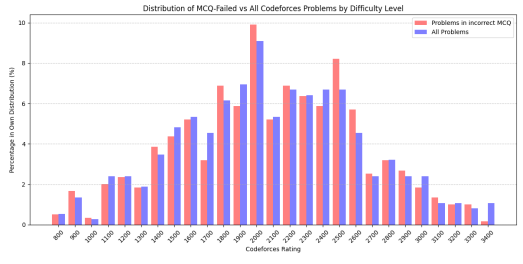


Figure 7: Difficulty-level distribution comparison of failed problems and all problems.

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.}

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].

Model Details

We evaluated all LLMs through their official or hosted API, details of the source models used are:

- GPT-4o-mini: OpenAI's official API gpt-*4o-mini-2024-07-18*
- GPT-4o: OpenAI's official API *gpt-4o-2024-08-06*
- o3-mini: OpenAI's official API *o3-mini-2025-01-31*, reasoning="medium"
- Gemini-2.0-Flash: Google AI Studio official API.
- Deepseek-V3: OpenRouter's hosted API *Deepseek-V3-03-24*
- Deepseek-R1: OpenRouter's hosted API *Deepseek-R1*
- Claude-3.5-sonnet: Anthropic's official API *claude-3-5-sonnet-20241022*

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 8: 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).

- Qwen3--thinking: HuggingFace, *Qwen/Qwen3-0.6B*; *Qwen/Qwen3-4B*; *Qwen/Qwen3-8B*
- Llama3.1-8B-Instruct: HuggingFace, *meta-llama/Llama-3.1-8B-Instruct*
- Qwen3-Coder-480B: OpenRouter, *qwen/qwen3-coder*

Retrieval Method Details 4.1

- BM25 [Trotman *et al.*, 2014], we use BM25Okapi with b=0.75, k1=1.2.
- TF-IDF [Sparck Jones, 1988], we use default parameters without setting thresholds.
- BERT [Devlin *et al.*, 2019], we use *google-bert/bert-base-uncased* from huggingface, model size 110M parameters.
- BART[Lewis *et al.*, 2020], we use *facebook/bart-base* from huggingface, model size 139M parameters.
- CodeBert [Feng *et al.*, 2020], we use *microsoft/codebert-base* from huggingface, model size 125M parameters.
- GraphCodeBert [Guo *et al.*, 2021], we use *microsoft/graphcodebert-base* from huggingface, model size 125M parameters.
- CodeSage-v2 [Zhang *et al.*, 2024], we use ‘codesage/codesage-base-v2‘, which has 356M parameters.
- SFR-Embedding-Code [Liu *et al.*, 2024] is from huggingface *Salesforce/SFR-Embedding-Code-400M_R*, which has 400M parameters.
- Jina-Code-V2 [Günther *et al.*, 2024] is from huggingface *jinaai/jina-embeddings-v2-base-code* model, which has 161M parameters.
- CodeRankEmbed[Suresh *et al.*, 2025] is from huggingface *nomic-ai/CodeRankEmbed*, which has 137M parameters.