

Idea First, Code Later: Disentangling Problem Solving from Code Generation in Evaluating LLMs for Competitive Programming

Sama Hadhoud¹ Alaa Elsetohy¹ Frederikus Hudi²
 Jan Christian Blaise Cruz¹ Steven Halim³ Alham Fikri Aji¹

¹MBZUAI ²NAIST ³National University of Singapore

{sama.hadhoud, alaa.elsetohy, jan.cruz, alham.fikri}@mbzuai.ac.ae
 frederikus.hudi.fe7@naist.ac.jp dcssh@nus.edu.sg

Abstract

Large Language Models (LLMs) increasingly succeed on competitive programming problems, yet existing evaluations conflate algorithmic reasoning with code-level implementation. We argue that competitive programming is fundamentally a problem-solving task and propose centering natural-language editorials in both solution generation and evaluation. Generating an editorial prior to code improves solve rates for some LLMs, with substantially larger gains when using expertly written gold editorials. However, even with gold editorials, models continue to struggle with implementation, while the gap between generated and gold editorials reveals a persistent problem-solving bottleneck in specifying correct and complete algorithms. Beyond pass/fail metrics, we diagnose reasoning errors by comparing model-generated editorials to gold standards using expert annotations and validate an LLM-as-a-judge protocol for scalable evaluation. We introduce a dataset of 83 ICPC-style problems with gold editorials and full test suites, and evaluate 19 LLMs, arguing that future benchmarks should explicitly separate problem solving from implementation.

1 Introduction

Competitive Programming (CP) is framed as a coding contest, but it primarily evaluates *algorithmic problem solving* under strict time and memory limits. For human contestants, the core deliverable is an *algorithmic plan*: key observations, data structures, invariants, and complexity arguments, while code is a downstream translation into an executable program (Halim et al., 2020). This workflow is reflected by contest *editorials*: natural-language explanations that describe the algorithm and justify correctness, and which serve as the solution reference after a contest.¹

¹See Appendix A for an example gold editorial.

LLMs have rapidly improved on CP-style tasks. Early systems such as AlphaCode demonstrated strong contest performance via large-scale sampling (Li et al., 2022), and more recent reasoning-oriented models achieve substantially higher accuracy on difficult competitive benchmarks (OpenAI et al., 2025; DeepSeek-AI et al., 2025a), with frontier results approaching elite human levels (International Collegiate Programming Contest (ICPC), 2025; Lin and Cheng, 2025). Despite this progress, most evaluations still treat CP as a single *problem-to-code* mapping (Chen et al., 2021; Hendrycks et al., 2021; Jain et al., 2025; Shi et al., 2024; Hossain et al., 2025; Quan et al., 2025), scoring only the final program. This end-to-end protocol conflates two distinct capabilities: **problem solving**—deriving a correct and efficient algorithm—and **implementation**—translating that plan into correct and efficient code. When a submission fails, standard metrics cannot distinguish between these failure modes.

We revisit CP evaluation by making *editorials* an explicit intermediate artifact. Figure 1 introduces an **editorial-centric evaluation pipeline** with three conditions: **w/oEd** (problem → code), **w/GenEd** (problem → model editorial → code), and **w/GoldEd** (problem + gold editorial → code). Crucially, **w/GoldEd approximates performance under perfect problem solving**, isolating implementation limitations, while the gap between w/GenEd and w/GoldEd directly reflects the reliability of model-generated reasoning.

To support this evaluation, we curate a dataset of 83 ICPC-style problems from seven contests (2017–2025), each packaged with its original statement, an expert-written gold editorial, and the full official test suite. We evaluate 19 contemporary LLMs under all three conditions using standard ICPC-style judging. Beyond pass/fail, we analyze generated editorials directly: we annotate a qualitative subset with an expert competitive programmer and

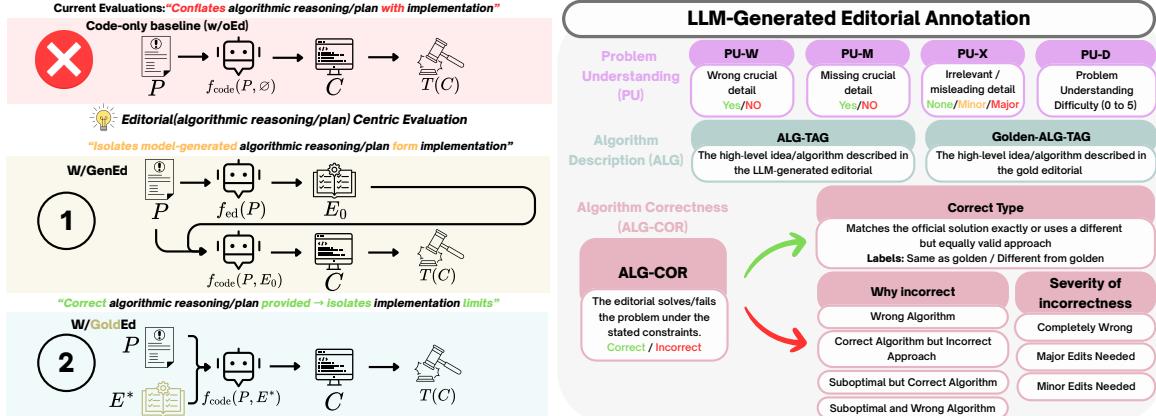


Figure 1: Overview of our evaluation pipeline and editorial annotation scheme. Left: three settings, **w/oEd** (problem → code, baseline), **w/GenEd** (problem → generated editorial → code), and **w/GoldEd** (problem plus gold editorial → code). Right: the LLM-generated editorial annotation rubric used to diagnose reasoning quality, covering *Problem Understanding* (PU-W, PU-M, PU-X, PU-D), *Algorithm Description* (ALG-TAG vs. Golden-ALG-TAG), and *Algorithm Correctness* (ALG-COR, correctness type, error type, and severity).

validate a scalable *LLM-as-a-judge* protocol that labels editorial quality against gold references (e.g., problem understanding, algorithm description, and algorithmic correctness).

Overall, gold editorials yield large and consistent gains, but performance remains far from saturated even under gold guidance, highlighting a substantial implementation bottleneck. Self-generated editorials provide smaller and less reliable gains and can even degrade performance when reasoning is misleading. These results suggest that CP benchmarks should move beyond problem-to-code scoring and explicitly evaluate reasoning and implementation as separate, measurable components, with editorials serving as the bridge.

To summarize, our contributions are as follows:

- We decompose competitive programming evaluation into a **Problem → Editorial → Code** pipeline that isolates problem solving from implementation.
- We quantify a **problem-solving gap** by comparing **w/GenEd** against **w/GoldEd**, showing that generated editorials often fail to specify correct and efficient algorithms.
- We identify an **implementation gap** by showing that models fail even under **w/GoldEd**, where the correct algorithm is given.
- We provide expert competitive-programmer annotations on editorial quality and validate an **LLM-as-a-judge** protocol that aligns with expert judgments.
- We show that editorials can transfer across models, enabling “writer–coder” composi-

tions where a strong planner improves a different model’s implementations.

2 Editorial-Centric Competitive Programming Evaluation

2.1 Problem setup

Let P denote a CP problem statement. Each contest problem in our dataset follows the usual structure: a natural-language statement, explicit time and memory limits, an *Input* section describing the input format, an *Output* section specifying the required output format, and one or more sample input/output pairs. We preserve this layout in Markdown and include one full example in Appendix A.

We model solution generation using two objects: an *editorial* E , which represents algorithmic reasoning and planning, and a *program* C , which represents executable code. Program correctness is evaluated using a standard ICPC-style compile-and-run judging pipeline on the official contest test suites (see Appendix C for full details), yielding an outcome $T(C) \in \{\text{PASS}, \text{Compile Error(CE)}, \text{Wrong Answer(WA)}, \text{Time Limit Exceeded(TLE)}, \text{MLE Memory Limit Exceeded(MLE)}, \text{Runtime Error(RTE)}\}$,

We explicitly separate reasoning and coding through two generation operators:

$$E = f_{\text{ed}}(P), \quad C = f_{\text{code}}(P, E),$$

where f_{ed} produces an editorial from the problem statement, and f_{code} produces code conditioned on both the problem and a given editorial. When no editorial is provided, $f_{\text{code}}(P, \emptyset)$ corresponds to direct problem-to-code generation.

2.2 Editorial-Centric Generation

Code-only baseline (w/oEd). In the baseline setting, the model is asked to solve the problem directly: $C = f_{\text{code}}(P, \emptyset)$. This mirrors standard CP benchmarks and conflates reasoning and implementation into a single step. Any failure may stem from incorrect planning, incorrect coding, or both.

Generated editorial (w/GenEd). To isolate model-generated reasoning, we first ask the model to write an editorial and then generate code conditioned on it: $E_0 = f_{\text{ed}}(P)$, $C = f_{\text{code}}(P, E_0)$. The editorial E_0 is generated once and kept fixed. This allows us to distinguish two failure modes: (i) the editorial itself describes an incorrect or incomplete algorithm, or (ii) the editorial is correct, but the model fails to faithfully implement it.

Gold editorial (w/GoldEd). To estimate an upper bound given correct reasoning, we provide the model with an expert-written gold editorial E^* : $C = f_{\text{code}}(P, E^*)$. Any remaining errors therefore arise from implementation limitations.

Throughout, we use a *single-shot* setting with a minimal prompting scheme (see Appendix E).

2.3 Contest Dataset and Evaluation Metrics

We curate 83 problems from seven contests that are not hosted on major public CP platforms (e.g., Codeforces, AtCoder). Some problem statements may be publicly accessible (e.g., contest PDFs or course materials), but we expect contamination risk to be lower than for widely scraped judge platforms. The problems come from regional ICPC contests and CS3233 (Competitive Programming course) examinations at the National University of Singapore, spanning 2017–2025.²

Each problem package includes the original problem statement, a gold editorial written by the problem setter or tester, and the full official test suite. We rely on complete judge test suites to avoid false positives.

To account for variation in problem difficulty, we group problems using solve rates³ from official scoreboards. Within each contest, problems are ranked by solve rate and partitioned into three contest-relative tertiles: T1 (easiest), T2 (middle), and T3 (hardest). Pooling across contests yields approximately balanced difficulty groups while preserving each contest’s intrinsic difficulty profile.

We report two complementary metrics. *pass@1* measures the fraction of problems for which a model’s first submission passes the full test suite. We also report a *virtual rank percentile* to contextualize performance relative to human teams: for each contest and setting, ignoring time-based penalties, the model is treated as an additional team whose score equals the number of problems solved and is inserted into the official scoreboard (1.0 = top team, 0.5 = median, 0.0 = last place). Averaging these percentiles across contests yields a single measure of human competitiveness.

2.4 Generated Editorial Evaluation by Experts

We perform an expert evaluation to understand the reasoning behaviors exhibited in model-generated editorials and how these behaviors relate to the results of the downstream execution. The study is performed on one representative contest (CS3233 2025 Midterm). For each problem, annotators review the gold editorial and the model-generated editorial from the w/GenEd condition.

Annotators Annotations were performed by an experienced IOI medalist competitive programmer with ICPC World Final participation; the annotator was blind to the identity of the model.

The pool of qualified annotators is very small; typically restricted to elite competitive programmers such as ICPC World Finalists and producing editorial level annotation under our rubric can take several hours.

Annotation Procedure Editorials are evaluated along three dimensions:

- **Problem Understanding (PU):** whether the editorial correctly captures the task, constraints, and corner cases without introducing misleading information.
- **Algorithm Description (ALG):** Records the algorithmic technique(s) and summaries for both the generated and golden editorials.
- **Algorithm Correctness (ALG-COR):** whether the described method solves the problem under the stated constraints.

Figure 1 (right) summarizes the annotation rubric; full definitions appear in Appendix D.

3 Results

We evaluate 19 contemporary LLMs spanning three groups: (i) proprietary reasoning-oriented

²See Appendix B for dataset and release details.

³The proportion of teams that solved the problem.

Model	Overall (83 problems)			T1 (26 problems)			T2 (28 problems)			T3 (29 problems)		
	w/oEd	w/GenEd (%/Δ)	w/GoldEd (%/Δ)	w/oEd	w/GenEd (%/Δ)	w/GoldEd (%/Δ)	w/oEd	w/GenEd (%/Δ)	w/GoldEd (%/Δ)	w/oEd	w/GenEd (%/Δ)	w/GoldEd (%/Δ)
	Closed Source Models											
GPT-5	67.5%	68.7% / +1.2%	83.1% / +15.7%	92.3%	88.5% / -3.8%	96.2% / +3.8%	75.0%	75.0% / +0.0%	92.9% / +17.9%	37.9%	44.8% / +6.9%	62.1% / +24.1%
O3	51.8%	45.8% / -6.0%	63.9% / +12.0%	69.2%	76.9% / +7.7%	80.8% / +11.5%	60.7%	46.4% / -14.3%	75.0% / +14.3%	27.6%	17.2% / -10.3%	37.9% / +10.3%
Gemini 2.5 Pro	43.4%	45.8% / +2.4%	72.3% / +28.9%	69.2%	73.1% / +3.8%	92.3% / +23.1%	42.9%	50.0% / +7.1%	78.6% / +35.7%	20.7%	17.2% / -3.4%	48.3% / +27.6%
Gemini 2.5 Flash	38.6%	37.3% / -1.2%	54.2% / +15.7%	65.4%	73.1% / +7.7%	80.8% / +15.4%	39.3%	32.1% / -7.1%	53.6% / +14.3%	13.8%	10.3% / -3.4%	31.0% / +17.2%
Claude Opus 4	21.7%	30.1% / +8.4%	47.0% / +25.3%	42.3%	57.7% / +15.4%	76.9% / +34.6%	21.4%	32.1% / +10.7%	57.1% / +35.7%	3.4%	3.4% / +0.0%	10.3% / +6.9%
Claude Sonnet 4	16.9%	19.3% / +2.4%	48.2% / +31.3%	34.6%	38.5% / +3.8%	73.1% / +38.5%	17.9%	17.9% / +0.0%	57.1% / +39.3%	0.0%	3.4% / +3.4%	17.2% / +17.2%
GPT-4.1	13.3%	17.1% / +3.8%	33.7% / +20.5%	26.9%	34.6% / +7.7%	61.5% / +34.6%	3.6%	14.8% / +11.2%	32.1% / +28.6%	10.3%	3.4% / -6.9%	10.3% / +0.0%
GPT-4o	7.2%	3.6% / -3.6%	13.3% / +6.0%	19.2%	7.7% / -11.5%	38.5% / +19.2%	0.0%	3.6% / +3.6%	3.6% / +3.6%	3.4%	0.0% / -3.4%	0.0% / -3.4%
<i>Closed Source Avg</i>	32.5%	33.5% / +0.9%	52.0% / +19.4%	52.4%	56.2% / +3.8%	75.0% / +22.6%	32.6%	34.0% / +1.4%	56.2% / +23.7%	14.7%	12.5% / -2.2%	27.2% / +12.5%
Open Source Models												
GPT-OSS-120B	41.0%	31.3% / -9.6%	59.0% / +18.1%	61.5%	61.5% / +0.0%	76.9% / +15.4%	50.0%	32.1% / -17.9%	75.0% / +25.0%	13.8%	3.4% / -10.3%	27.6% / +13.8%
GPT-OSS-20B	33.7%	27.7% / -6.0%	47.0% / +13.3%	57.7%	53.8% / -3.8%	65.4% / +7.7%	39.3%	28.6% / -10.7%	53.6% / +14.3%	6.9%	3.4% / -3.4%	24.1% / +17.2%
DeepSeekR1	28.9%	45.8% / +16.9%	43.4% / +14.5%	61.5%	88.5% / +26.9%	65.4% / +3.8%	21.4%	39.3% / +17.9%	53.6% / +32.1%	6.9%	13.8% / +6.9%	13.8% / +6.9%
Qwen-3-8B	15.7%	13.3% / -2.4%	24.1% / +8.4%	34.6%	34.6% / +0.0%	42.3% / +7.7%	14.3%	7.1% / -7.1%	25.0% / +10.7%	0.0%	0.0% / +0.0%	6.9% / +6.9%
DeepSeekV3	14.5%	9.6% / -4.8%	28.9% / +14.5%	34.6%	26.9% / -7.7%	61.5% / +26.9%	10.7%	3.6% / -7.1%	25.0% / +14.3%	0.0%	0.0% / +0.0%	3.4% / +3.4%
Qwen-3-Coder-480B-A35B	13.3%	10.8% / -2.4%	28.9% / +15.7%	23.1%	19.2% / -3.8%	61.5% / +38.5%	14.3%	10.7% / -3.6%	17.9% / +3.6%	3.4%	3.4% / +0.0%	10.3% / +6.9%
Kimi-K2	13.3%	13.3% / +0.0%	26.5% / +13.3%	34.6%	34.6% / +0.0%	50.0% / +15.4%	3.6%	7.1% / +3.6%	28.6% / +25.0%	3.4%	0.0% / -3.4%	3.4% / +0.0%
OlympicCoder-7B	6.0%	8.4% / +2.4%	10.8% / +4.8%	19.2%	26.9% / +7.7%	26.9% / +7.7%	0.0%	0.0% / +0.0%	7.1% / +7.1%	0.0%	0.0% / +0.0%	0.0% / +0.0%
Llama-3-1-40B	6.0%	2.4% / -3.6%	15.7% / +9.6%	19.2%	7.7% / -11.5%	42.3% / +23.1%	0.0%	0.0% / +0.0%	3.6% / +3.6%	0.0%	0.0% / +0.0%	3.4% / +3.4%
Llama-3-3.7B	4.8%	6.0% / +1.2%	8.4% / +3.6%	11.5%	11.5% / +0.0%	23.1% / +11.5%	3.6%	7.1% / +3.6%	0.0% / -3.6%	0.0%	0.0% / +0.0%	3.4% / +3.4%
Gemma-3-27B	3.6%	2.4% / -1.2%	8.4% / +4.8%	7.7%	3.8% / -3.8%	23.1% / +15.4%	3.6%	3.6% / +0.0%	3.6% / +0.0%	0.0%	0.0% / +0.0%	0.0% / +0.0%
<i>Open Source Avg</i>	16.4%	15.6% / -0.9%	27.4% / +11.0%	33.2%	33.6% / +0.3%	49.0% / +15.7%	14.6%	12.7% / -1.9%	26.6% / +12.0%	3.1%	2.2% / -0.9%	8.8% / +5.6%
<i>Overall Avg</i>	23.2%	23.1% / -0.1%	37.7% / +14.5%	41.3%	43.1% / +1.8%	59.9% / +18.6%	22.2%	21.6% / -0.5%	39.1% / +16.9%	8.0%	6.5% / -1.5%	16.5% / +8.5%

Table 1: Pass@1 by difficulty tertile (T1 easiest–T3 hardest). Gold editorials substantially improve performance across all difficulties (up to $\sim 30\%$), but hard problems remain challenging, indicating a residual **implementation bottleneck**. Self-generated editorials yield smaller, model-dependent gains (up to $\sim 15\%$) and sometimes hurt performance, highlighting a persistent **problem-solving gap** in model-generated reasoning.

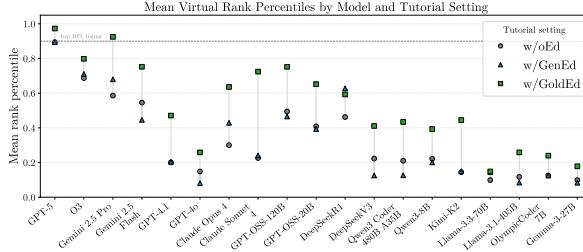


Figure 2: Mean virtual rank percentile under w/oEd, w/GenEd, and w/GoldEd (higher is better). Gold editorials yield large and consistent improvements (up to ~ 0.4), yet even under gold guidance only a small number of models attain high rank percentiles (above ~ 0.8), with only a handful exceeding ~ 0.7 .

systems, (ii) proprietary general-purpose chat models, and (iii) open-weight models over a wide range of scales, including both frontier and open models.⁴ We report results in C++, the standard language for ICPC-style contests; **Python generally performs worse** (Appendix G). Table 1 reports pass@1 across all problems, stratified by difficulty tertiles (T1 easiest–T3 hardest). We compare three conditions (see Section 2.2): a code-only baseline (w/oEd), a two-stage setting with a model-generated editorial (w/GenEd), and a setting with a gold editorial (w/GoldEd).

3.1 Overall Performance and Editorial Effects

Gold editorials isolate an implementation gap. Table 1 shows a clear asymmetry between generated and gold editorials. On average, providing a gold editorial yields a large absolute gain, in-

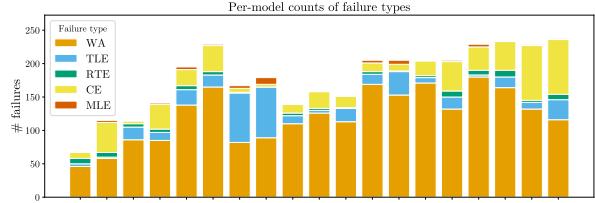


Figure 3: Aggregate failure verdict distribution across *all* editorial settings: Wrong Answer (WA), Time Limit Exceeded (TLE), Runtime Error (RTE), Compile Error (CE), and Memory Limit Exceeded (MLE). Remaining failures are dominated by WA, while TLE becomes more salient for some stronger models (notably Claude)

dicating that many baseline failures are problem-solving-limited and can be removed by supplying a correct plan. However, performance remains far from saturated even with gold guidance, especially on T3 (rising only from 8.0% to 16.5% on average), which isolates a substantial residual implementation gap: models often struggle to translate a high-level algorithm into correct and efficient code.

The generated-gold editorial gap reflects a problem-solving limitation. In contrast, self-generated editorials yield much smaller and less reliable changes, and can even degrade performance for some models. A few stronger closed-source models, along with DeepSeek R1, obtain gains of up to $\sim 15\%$, but these improvements are concentrated in T1 and T2 and largely disappear on T3. The gap between w/GenEd and w/GoldEd therefore directly reflects a problem-solving limitation: many models do not reliably produce editorials

⁴Model identifiers and inference details are in Appendix F.

that fully specify a correct algorithm under the stated constraints. When the generated editorial is incomplete, overly complex, or subtly wrong, conditioning on it can lock the model into a flawed plan, explaining why some models benefit from self-editorials while others regress.

Only a few models become strongly human-competitive even with gold editorials. Figure 2 places each model as a *virtual team* on official contest scoreboards and reports mean rank percentiles across contests. w/GoldEd consistently improve rankings (up to ~ 0.4 absolute percentile points), yet most models remain far from top-team performance, despite receiving full solution guidance, whereas human teams solved the same problems under strict contest time limits. Only a small number of models exceed the mean rank percentile ~ 0.8 and only slightly more surpass ~ 0.7 , largely only with gold editorials; w/GenEd instead produces smaller (up to ~ 0.2) and higher-variance shifts. Overall, correct plans alone are insufficient for most models to match strong human teams, revealing persistent gaps in implementation fidelity and efficiency under contest-style evaluation. Per-contest ladders are shown in Appendix H; while absolute percentiles vary across contests, the qualitative editorial effects are consistent.

WA dominates, while TLE rises for stronger models (notably Claude). Figure 3 is WA-heavy for nearly all models, showing that most errors are still correctness-limited—either the model does not reach a correct algorithm or it fails to implement a correct plan robustly. However, TLE is disproportionately common for a subset of stronger models, most clearly the Claude variants, suggesting a shift toward problem-solving failures at the complexity level: the approach is often plausible, but misses the key optimization needed to meet contest limits. The remaining CE/RTE mass is smaller but points to residual implementation fragility. A breakdown per (model, setting) is reported in Appendix I.

3.2 Qualitative analysis of editorial behavior

While the quantitative results show that editorials affect pass@1, they do not explain *why* or where failures occur. To probe model reasoning more directly, we inspect 22 editorials (11 problems \times 2 models) from a single contest (CS3233 2025 Midterm) using the rubric in Appendix D. This small but indicative case study focuses on two models, the open-weight reasoner **DeepSeekR1** and the

closed-source frontier model **GPT-5**. Each editorial is annotated for problem understanding, algorithm description, and correctness, and paired with the execution outcome of its w/GenEd code. Full per-editorial annotations appear in Appendix K.

Severe problem understanding errors are rare; the main risk is hallucination. Most editorials accurately capture the task requirements: 21 of 22 are not flagged with any wrong or missing crucial details. Only one case exhibits both wrong and missing crucial details with a *major* level of misleading content. In this instance, the model hallucinates an additional constraint absent from the problem statement (see excerpt below).

Example: problem understanding error

Problem statement (excerpt): “A chain slide is a continuous sequence of slides such that $Y_{i_j} = X_{i_{j+1}}$ and ends at the button the chain slide started from.”

DeepSeekR1 editorial (excerpt): “The selected edges must form an *Eulerian subgraph*, meaning that for every node, the in-degree equals the out-degree.”

Explanation. In CHAINED MAIMAI SLIDES, the model editorial hallucinates a global conservation constraint (Eulerian in-degree equals out-degree) that is not implied by the problem statement, thereby ruling out valid solutions composed of multiple disjoint cycles.

That said, minor misleading or imprecise statements do appear, usually reflecting algorithmic assumptions or complexity oversights rather than task misunderstandings. In general, the difficulty of understanding the problem is low, with all problems rated as easy or medium.

Reasoning failures often involve wrong plans, missing invariants, or untightened complexity. Among the editorials, 15 are labeled algorithmically correct and 7 Incorrect. The incorrect editorials cluster into a small number of recurring patterns. Five are labeled *Wrong algorithm*, where the high-level idea does not solve the full problem, often due to a subtle but decisive reasoning gap. For example, in ARTS AND COMPUTING STUDENTS (DeepSeekR1), the editorial treats a constrained *shift* as free *movement*, invalidating the construction.

The remaining two incorrect editorials are labeled “Suboptimal (Likely TLE or MLE), but correct algorithm”: the idea is sound, but the editorial fails to tighten the complexity, and resulting code is asymptotically too slow, leading to timeouts.

Example: missing algorithmic invariant

Gold editorial (excerpt): “Let f_i be the maximum path length ending at i , and let b_i be the maximum path length starting from i .¹”

GPT-5 editorial (excerpt): “For each node, we propagate the accumulated value from the root downwards and record the result at that node.”

Explanation. In DEPENDENCY FLOOD, the model preserves the correct problem formulation but tracks only forward propagation and omits the backward aggregation, yielding an incomplete algorithmic specification and incorrect results.

Example: suboptimal despite correct idea

Gold editorial (excerpt): “A convex polyomino in a board with m columns and n rows has at most $\mathcal{O}(m + n)$ boundary cells.”

DeepSeekR1 editorial (excerpt): “For each rotation and translation, we count the overlapping border cells by traversing the grid cell by cell.”

Explanation. In FICKETTS CONJECTURE FOR POLYOMINOES, the editorial correctly enumerates rotations and translations but performs an $\mathcal{O}(RC)$ scan per placement, ignoring the boundary-size bound that reduces the intended solution to $\mathcal{O}(R + C)$, leading to likely timeouts.

Correctness \neq clarity: gold editorials assume CP familiarity, while LLMs spell out steps/definitions. The annotations reveal a gap between *correctness* and *clarity*. Gold editorials serve as a reference for algorithmic correctness but are not always the most operationally clear. In DEPENDENCY FLOOD (DeepSeek-R1), the model editorial is labeled “Same as golden” yet judged easier to grasp than the official explanation, despite implementing the same idea. A similar pattern appears in HUNGRY PIPLUPS (GPT-5), where the model’s editorial explains each component step by step, while the gold editorial assumes familiarity with standard CP abstractions (see Appendix O). Consistent with this, model-generated editorials are substantially longer than gold editorials across models (Appendix J), suggesting a trade-off between concision and explicit reasoning.

Paradigm tags are only weakly informative: mismatches and false positives both occur. Algorithmic tags provide a coarse check of whether a model editorial operates in the same broad paradigm as the gold editorial. On the annotated

set, DeepSeek R1 matches the gold tags on 6 of 11 problems, and GPT-5 on 7 of 11. When tags differ, models often omit a key paradigm, for example treating HUNGRY PIPLUPS as simulation or greedy reasoning rather than the segment-tree and binary-search approach in the gold editorial. However, tag mismatch does not necessarily imply incorrectness, as in EASYGOING WORKPLACE (both), while tag agreement does not guarantee correctness, as in DEPENDENCY FLOOD (GPT-5). Overall, tag alignment is only a weak signal of editorial correctness.

Algorithmic correctness is necessary, not sufficient: implementation remains a gap. Algorithmic correctness at the editorial level strongly predicts downstream results. Of the 15 editorials labeled algorithmically correct, 12 lead to accepted submissions; the remaining cases fail due to efficiency or robustness issues (two TLEs, one RTE). None of the 7 editorials labeled Incorrect pass. Correct reasoning is therefore necessary but not sufficient: even with a sound plan, models may fail to produce efficient or robust implementations

Overall, the qualitative analysis supports four conclusions: (i) editorial-level algorithmic correctness strongly predicts success; (ii) most reasoning failures arise from incorrect or incomplete plans rather than problem misunderstanding; (iii) implementation efficiency and robustness remain major bottlenecks even with correct editorials; and (iv) while gold editorials provide a reliable correctness reference, model-generated editorials can sometimes be clearer and equally effective.

3.3 LLM-as-a-Judge Editorial Diagnostics

Expert editorial annotations provide high-fidelity insight into algorithmic reasoning failures, but are expensive to scale. We therefore adopt an *LLM-as-a-judge* to label *all* generated editorials using the same rubric as the expert annotator (Section 2.4), guided by a gold editorial that provides a consistent reference for correctness and error diagnosis.

Judge setup. For each w/GenEd run, the judge is given the problem statement P , the gold editorial E^* , and the model-generated editorial E , and outputs structured labels. We use [google/gemini-3-pro-preview](#) (Google DeepMind, 2025) as the judge for reliable long-context rubric following and structured outputs, and because it is not part of the evaluated LLMs (Table 1). The full LLM-as-a-judge prompt and output schema are provided in Appendix L.1.

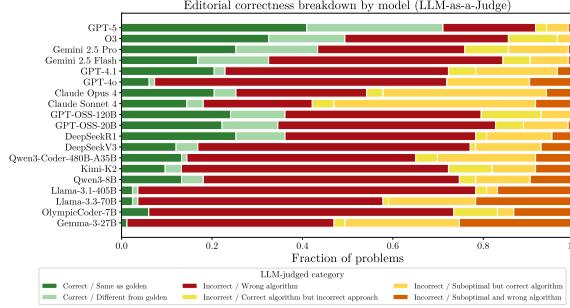


Figure 4: Six-way editorial correctness breakdown labeled by the LLM-as-a-judge. Frontier models produce more judge-CORRECT plans, but *wrong algorithm*—i.e., incorrect problem solving—remains the dominant error across many models.

Rubric field	n	Agr.	κ
ALG-COR (Correct vs. Incorrect)	22	0.818	0.611
Correct type (given expert=Correct)	15	0.733	0.474
Why incorrect (given expert=Incorrect)	7	0.714	0.500
Severity (given expert=Incorrect)	7	0.429	0.152

Table 2: Agreement between expert annotator and Gemini 3 Pro judge on the validation subset. The LLM judge is reliable for overall correctness and for identifying why an editorial is incorrect; severity is less consistent.

Validation against expert labels. We validate the judge on an expert-annotated subset of the CS3233 2025 Midterm. Table 2 reports agreement on the core correctness signal (ALG-COR) and the conditional diagnostics used in our analysis. We report accuracy and Cohen’s κ , with conditionally-defined fields evaluated only where applicable under the expert label. The judge shows strong agreement on overall correctness and moderate agreement on both *correct type* and *why incorrect*, indicating reliable identification of incorrectness and its primary cause. Agreement on *severity* is weaker; we therefore treat severity as exploratory. See Appendix L.2 for results on auxiliary rubric fields.

A persistent problem-solving gap across models. Figure 4 decomposes w/GenEd editorials by the six-way ALG-COR taxonomy. Frontier models produce a substantially larger share of editorials judged CORRECT (both *same as* and *different from* the gold approach), yet even for these models a large fraction of editorials remain incorrect. Among incorrect cases, *wrong algorithm* dominates for most models—particularly open-weight systems—indicating that failures under w/GenEd are primarily due to incorrect problem-solving plans rather than purely implementation errors. Claude variants form a notable exception, allocat-

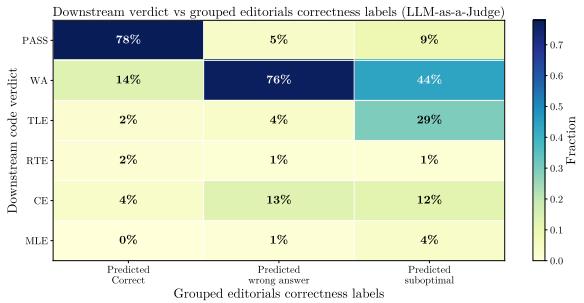


Figure 5: Downstream verdict distribution (PASS/WA/TLE/RTE/CE/MLE) conditioned on editorial correctness labels. Editorial correctness labels meaningfully stratify downstream outcomes.

ing comparatively more mass to *suboptimal but correct* editorials, consistent with reasoning that captures the right idea but fails to tighten complexity or efficiency arguments.

Judge labels predict downstream failure modes. Figure 5 (right) shows that judge labels translate into distinct execution outcomes for the downstream w/GenEd code. For readability, we collapse the six ALG-COR categories into three groups: *Correct* (same/different), *Predicted WA* (wrong algorithm or incorrect approach, suboptimal and wrong), and *Predicted suboptimal* (suboptimal but correct). When the editorial is judged *Correct*, the downstream program is accepted most of the time, but the remaining WA/CE/RTE/TLE tail directly exposes an **implementation gap** even under a sound plan. When the editorial is *Predicted WA*, failures are overwhelmingly WA (with a small PASS tail), indicating that editorial-level reasoning mistakes largely propagate into incorrect outputs. When the editorial is *Predicted suboptimal*, TLE is strongly enriched (while WA remains common), providing a sanity check that the judge’s efficiency-related diagnoses align with runtime failures. See Appendix L.3 for more analysis: problem-understanding errors are relatively rare but consequential, since correct understanding is the bare minimum in competitive programming. Algorithm-tag alignment is a weak proxy for success compared to fine-grained ALG-COR labels.

3.4 Cross-model editorial transfer

Our annotations suggest that model-generated editorials are often easier to follow than the accompanying gold editorials, motivating a transfer question: can an editorial written by one model benefit a different model that only acts as a coder?

So far, models were evaluated in a *matched* configuration, where the same system supplies both f_{ed} and f_{code} . Here, we decouple these roles by treating models as interchangeable *writers* and *coders*. Given a problem P , a writer m_w produces an editorial $E^{(m_w)}$, and a coder m_c generates code conditioned on it: $C^{(m_w \rightarrow m_c)} = f_{\text{code}}^{(m_c)}(P, E^{(m_w)})$. When $m_w = m_c$, this recovers w/GenEd; replacing $E^{(m_w)}$ with a gold editorial yields w/GoldEd.

We evaluate this cross-model setting using three open-weight coders (Qwen3-Coder-480B-A35B, GPT-OSS-20B, Qwen3-8B) and several strong writers (DeepSeek-R1, Claude Opus 4, Gemini 2.5 Pro, GPT-5, GPT-OSS-120B). Figure 6 compares each coder’s self-editorial and gold-editorial baselines to cross-model compositions.

Editorial transfer improves coding. Across all coders, every cross-model configuration performs at least as well as the coder’s w/GenEd, and many substantially outperform it. For larger coders, editorials from the strongest writers often recover much of the writer’s own end-to-end performance and can even exceed the coder’s w/GoldEd score; in some cases, a weaker coder implements a stronger model’s plan more effectively than the writer itself. Overall, these results show that reasoning and implementation can be modularized: pairing a strong writer with a competent coder can outperform either model used end-to-end, with editorials serving as a simple, model-agnostic interface.

4 Related Work

Competitive Programming Benchmarks Programming LLMs are often evaluated end-to-end on unit tests (APPS (Hendrycks et al., 2021), HumanEval (Chen et al., 2021)) and, more recently, in contest-style settings (AlphaCode/CodeContests (Li et al., 2022), LiveCodeBench (Jain et al., 2025), USACO (Shi et al., 2024), LLM-ProS (Hossain et al., 2025), CodeElo (Quan et al., 2025)). Closest to our goal of disentangling steps, Coding Triangle evaluates editorials, code, and tests (Zhang et al., 2025), and Yang et al. (2025b) studies staged competitive-programming workflows. We focus on editorials as an explicit, transferable plan separating reasoning from implementation.

Code LLMs and Reasoning LLMs Progress spans code-specialized models (e.g., CodeGeeX (Zheng et al., 2024), StarCoder (Li et al., 2023),

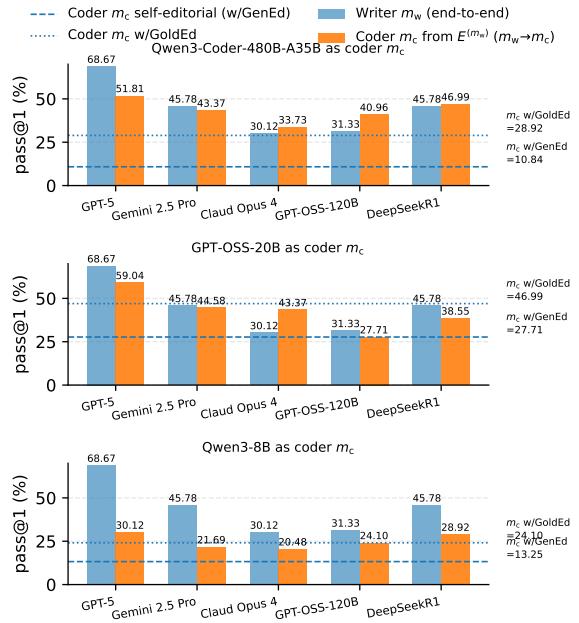


Figure 6: Cross-model editorial transfer. Each bar reports pass@1 when a fixed coder implements an editorial written by a different model. Using editorials from stronger writers often improves weaker coders and can occasionally yield performance competitive with or exceeding the writer’s own end-to-end results.

Code Llama (Rozière et al., 2024)) and reasoning-oriented systems (OpenAI’s O-series (OpenAI et al., 2025), DeepSeek-R1 (DeepSeek-AI et al., 2025a)). Our editorial-centric pipeline evaluates plan quality and implementation fidelity independently and enables writer–coder composition. Appendix M provides additional related work.

5 Conclusion

We revisited competitive-programming evaluation for large language models by treating editorials as explicit artifacts that separate algorithmic reasoning from implementation. Across 83 ICPC-style problems and 19 models, gold editorials consistently yield large gains in pass@1 and virtual rank, while self-generated editorials have smaller and less reliable effects. Even with correct editorials, performance remains far from saturated, with failures dominated by wrong answers and timeouts, underscoring persistent implementation bottlenecks.

We also show that editorials transfer effectively across models: editorials from strong reasoners can boost weaker coders and sometimes outperform the writer’s own end-to-end performance. Together, these results suggest that reasoning and implementation can be modularized, with editorials serving as a model-agnostic interface.

6 Limitations

We focus on a constrained but high-quality evaluation setting. All main experiments are conducted in C++, the dominant ICPC contest language; Python results are reported separately in Appendix G and exhibit systematically lower pass rates. The dataset is relatively small, spanning seven contests and 83 problems, and may not generalize to other contests, programming languages, or problem formats. However, this scale is a deliberate tradeoff: curating competitive-programming problems with *complete official test suites* and *expert-written gold editorials*, while minimizing contamination risk from widely scraped online judges, substantially constrains dataset size.

Expert annotation further limits coverage. The pool of qualified annotators is very small—typically restricted to elite competitive programmers such as ICPC World Finalists—and producing editorial-level annotation under our rubric can take several hours. As a result, expert annotations are limited to a single contest and a single annotator, and we treat one gold editorial per problem as the reference despite the existence of multiple valid solution strategies.

All evaluations use a single-shot deterministic setup with minimal prompting, without sampling, tool use, or editorial-aware fine-tuning, aside from a small appendix-level exploration of test-time feedback (Appendix N). Consequently, our results should be interpreted as evidence for the value of editorial-centric evaluation and modular reasoning–implementation separation, rather than as a definitive ranking of models across all competitive-programming settings.

7 Ethical Considerations

This paper studies large language models (LLMs) in competitive programming (CP) via an editorial-centric evaluation protocol that separates *problem solving* (deriving an algorithm) from *implementation* (writing correct and efficient code). The work is methodological and diagnostic: we do not deploy models in real-world decision-making.

Academic integrity and misuse. Improved CP capability may be misused to cheat in coursework or contests. Our benchmark is built from *past* contests and course materials and is intended for research and post-hoc benchmarking, not for use in active competitions. We do not provide tooling

designed to bypass contest rules. Because releasing complete test suites can reduce the future usefulness of these problems for hidden-test assessments, we recommend that educators avoid reusing included problems as undisclosed evaluation items.

Dataset provenance, permissions, and privacy.

Our dataset consists of problem statements, gold editorials, and official test suites from seven sources (Appendix B). The CS3233 (NUS) portion includes course assessment materials; we obtained permission from the course instructor to redistribute the problem statements and gold editorials. Other contests were sourced from publicly accessible task repositories; we preserve attribution and provide provenance metadata for each problem. The dataset contains no personal data about participants, students, or annotators; we report aggregate statistics and do not attempt to infer or disclose private information.

Safety of executing model-generated code and evaluation limitations.

We compile and run untrusted, model-generated programs in a sandboxed ICPC-style judging pipeline with strict time and memory limits (Appendix C) and do not deploy generated code in production systems. Our benchmark reflects ICPC-style constraints and is primarily evaluated in C++ (Appendix G); conclusions should be interpreted as CP-style reasoning and implementation performance rather than general-purpose software engineering ability.

8 Use of Generative AI

We used generative AI tools for language-level assistance (e.g., copy-editing, rephrasing, and improving clarity) on portions of the manuscript. All research ideas, experimental design, dataset construction, implementation of the evaluation pipeline, quantitative analyses, and conclusions were developed by the authors. We did not use generative AI to create or alter experimental results, to fabricate citations, or to replace expert validation; all citations and factual claims were checked by the authors.

References

- Anthropic. 2025. [Claude 4 system card: Claude opus 4 & claude sonnet 4](#). Technical Report 4263b940cabb546aa0e3283f35b686f4f3b2ff47, Anthropic.

- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. [Codet: Code generation with generated tests](#). In *The Eleventh International Conference on Learning Representations*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#).
- Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marrs, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3290 others. 2025. [Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agetic capabilities](#). Preprint, arXiv:2507.06261.
- 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, Zhi-hong Shao, Zhusu Li, Ziyi Gao, and 181 others. 2025a. [Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning](#). Preprint, arXiv: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, and 181 others. 2025b. [Deepseek-v3 technical report](#). Preprint, arXiv:2412.19437.
- Google DeepMind. 2025. [Gemini 3 Pro Model Card](#). Model card, Google DeepMind.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. [The llama 3 herd of models](#). Preprint, arXiv:2407.21783.
- S. Halim, F. Halim, and S. Effendy. 2020. [Competitive Programming 4 - Book 2: The Lower Bound of Programming Contests in the 2020s](#). bk. 2.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Md Sifat Hossain, Anika Tabassum, Md. Fahim Arefin, and Tarannum Shaila Zaman. 2025. [Llm-pros: Analyzing large language models' performance in competitive problem solving](#). In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 80–87.
- International Collegiate Programming Contest (ICPC). 2025. Openai joins inaugural ai tools experiment at the 2025 icpc world finals. <https://worldfinals.icpc.global/2025/openai.html>.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. [MapCoder: Multi-agent code generation for competitive problem solving](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations*.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. [Starcoder: may the source be with you!](#) Preprint, arXiv:2305.06161.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweis, 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, and 7 others. 2022. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- Hanzhao (Maggie) Lin and Heng-Tze Cheng. 2025. [Gemini achieves gold-medal-level performance at the international collegiate programming contest world finals](#). Google DeepMind Blog, 17 Sep 2025.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. [Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Thirty-seventh Conference on Neural Information Processing Systems*.
- OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz

- Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, and 7 others. 2025. [Competitive programming with large reasoning models](#). *Preprint*, arXiv:2502.06807.
- OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, and 401 others. 2024. [Gpt-4o system card](#). *Preprint*, arXiv:2410.21276.
- OpenAI. 2025a. Introducing gpt-4.1 in the api. <https://openai.com/index/gpt-4-1/>.
- OpenAI. 2025b. Introducing gpt-5. <https://openai.com/index/introducing-gpt-5/>.
- Guilherme Penedo, Lewis Tunstall, Anton Lozhkov, Hynek Kydlicek, Edward Beeching, Loubna Ben Allal, Quentin Gallouédec, Leandro von Werra, Agustín Piqueres Lajarín, Nathan Habib, and 1 others. 2025. Open r1: Update #3. <https://huggingface.co/blog/open-r1/update-3>. Hugging Face Blog.
- Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. [Code llama: Open foundation models for code](#). *Preprint*, arXiv:2308.12950.
- Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. 2024. [Can language models solve olympiad programming?](#) In *First Conference on Language Modeling*.
- Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, and 197 others. 2025a. [Gemma 3 technical report](#). *Preprint*, arXiv:2503.19786.
- Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chen-zhuang Du, Dikang Du, Yulun Du, Yu Fan, and 150 others. 2025b. [Kimi k2: Open agentic intelligence](#). *Preprint*, arXiv:2507.20534.
- Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025. [CodeContests+: High-quality test case generation for competitive programming](#). In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.
- Xinwei Yang, Zhao Feng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025b. [ELABORATION: A comprehensive benchmark on human-LLM competitive programming](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 59–104, Vienna, Austria. Association for Computational Linguistics.
- Taolin Zhang, Zihan Ma, Maosong Cao, Junnan Liu, Songyang Zhang, and Kai Chen. 2025. [Coding triangle: How does large language model understand code?](#) *Preprint*, arXiv:2507.06138.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. [Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x](#). *Preprint*, arXiv:2303.17568.

A Example Problem and Editorial

This appendix shows an example problem and its corresponding gold editorial from our dataset. Figure 7 shows a representative competitive-programming problem and illustrates the problem format used throughout our dataset. Figure 8 shows an example gold editorial, illustrating the level of algorithmic detail expected from expert-written editorials.

B Dataset Details and Metadata

Provenance and contest coverage. Our dataset contains 83 ICPC-style problems drawn from seven contests spanning 2017–2025: three CS3233 midterm contests hosted on NUS (course instance) and four regional ICPC contests distributed via public task repositories. Table 3 summarizes contest-level statistics.

Copyright and permissions. The CS3233 portion of the dataset consists of course assessment materials from the National University of Singapore; we requested copyright permission from the course instructor to include and redistribute these

Example Problem Statement

Contest: CS3233 Midterm Contest 2025

Problem title: Arts and Computing Students

Time limit: 1 second

Memory limit: 1024 MB

Problem Statement.

To encourage interdisciplinary collaboration, the School of Computing has decided to hold a dinner event for students in School of Computing and the Faculty of Arts and Social Sciences. The event will be held in a large hall with a long table that can seat $2N$ students. The $2N$ seats are arranged on a single line, with N pairs of seats. Seats belonging to the same pair are adjacent in the line. Each seat will either contain a student from the School of Computing, a student from the Faculty of Arts and Social Sciences, or be empty. Two students sitting next to each other within the same pair of seats are called “buddies”. There are at most N pairs of buddies by definition.

The Dean of the School of Computing suspects that School of Computing students will start Leetcoding on their laptops if they do not have a student from the Faculty of Arts and Social Sciences as a buddy. To prevent this, the Dean wants to rearrange the students such that each student from the School of Computing has a buddy from the Faculty of Arts and Social Sciences.

The Dean can perform the following operation: he can choose a student sitting next to an empty seat and move that student to the empty seat. These two seats need not belong to the same pair.

Given the initial arrangement of students, determine whether it is possible to rearrange the students such that each student from the School of Computing has a buddy from the Faculty of Arts and Social Sciences.

Input.

The first line contains an integer N ($1 \leq N \leq 2 \times 10^5$), the number of pairs of seats.

The second line contains a string of length $2N$ consisting of the characters C, A, and . The i -th character is C if the student in the i -th seat is from the School of Computing, A if the student is from the Faculty of Arts and Social Sciences, and . if the seat is empty.

Output.

If it is not possible to rearrange the students such that each C has an adjacent A in its pair, output:

NO

Otherwise, output:

YES

<final-arrangement-string>

The second line must be a string of length $2N$ over {C, A, .} representing a final valid arrangement after allowed moves. Any valid final arrangement may be output.

Sample Input #1

```
3
C.AAAC
```

Sample Output #1

```
YES
CAA.AC
```

Sample Input #2

```
4
AAA.CCAC
```

Sample Output #2

NO

Sample Input #3

```
4
CA.A.C.A
```

Sample Output #3

```
YES
CA..ACA.
```

Figure 7: Example competitive-programming problem from our dataset (ARTS AND COMPUTING STUDENTS).

Example Gold Editorial: *Arts and Computing Students*

There are two solutions to this problem:

- **Dynamic Programming:** Observe that our ultimate goal is to pair up the students as much as possible, as we risk running out of space (there are only N pairs of seats) if we do not pair them up sufficiently. We can let $dp[i]$ be the minimum number of pairs needed to pair up students $1, 2, \dots, i$ such that the constraint is satisfied. To compute $dp[i]$, it suffices to condition on whether student i is paired with student $i - 1$:

$$dp[i] = \min \begin{cases} dp[i - 2] + 1 & \text{if student } i \text{ and } i - 1 \text{ forms a valid pair} \\ dp[i - 1] + 1 & \text{if student } i \text{ can sit alone (i.e. arts student)} \end{cases}$$

A solution exists if $dp[\# \text{ students}] \leq N$, and we can backtrack to find the construction.

- **Greedy:** Notice that if we have a computing student at the beginning / end of the array or we have two consecutive computing students, then their seating arrangements are fixed. (For example, if we have ACCA as a substring, we know that the pairs must be AC and CA.) We first split the string whenever we see ACCA. For the remaining chunks without two consecutive computing students, we can greedily seat them using $\lfloor \frac{\# \text{ students}}{2} \rfloor$ pairs (if we have more computing students than arts students, then this is clearly impossible). It is easy to see that this greedy strategy minimizes the number of pairs needed to seat all students.

P.S. Many teams attempted to code a (wrong) greedy solution right away. It's a better idea to (informally) justify the correctness of your greedy approach or try coming up with counter-examples before coding. When in doubt, dynamic programming is the safe choice to go with, as long as it's fast enough!

Figure 8: Gold editorial for ARTS AND COMPUTING STUDENTS.

materials (problem statements, gold editorials) as part of our dataset release. The CS3233 gold editorials are private course materials that were not publicly released prior to this work.

Release plan. We will release the dataset upon publication, including problem statements, gold editorials, and the full official test suites / judging harness.

C Judging Protocol Details

We evaluated all generated programs using a standard ICPC-style compile-and-run judging pipeline in the official contest test suites.

For each submission, we first attempt to compile the program (or perform a syntax check for interpreted languages). Compilation failures are labeled CE (Compile Error). Otherwise, the resulting executable is run in each test case under the specified time and memory limits. Executions that exceed the time or memory budget are labeled TLE (Time Limit Exceeded) or MLE (Memory Limit Exceeded), respectively, and abnormal termination (e.g., segmentation faults or runtime exceptions) is labeled RTE (Runtime Error).

If the execution is completed successfully, the program's output is compared against the reference output provided by the contest judges; any mismatch yields WA (Wrong Answer). The first

failure verdict encountered during the testing is reported as the result $T(C)$. A submission is labeled PASS only if it compiles successfully and produces correct output for all test cases within the prescribed resource limits.

Submissions that do not yield a usable program (e.g., explicit refusals, incomplete code, or outputs in the wrong programming language) are conservatively treated as CE or RTE, consistent with standard contest judging practice. These rare failure modes are analyzed separately in the Appendix I.

D Editorial Annotation Rubric

Figure 9 provides the full rubric used for annotating model-generated editorials. The rubric is organized into three components: Problem Understanding (PU), Algorithm Description (ALG), and Algorithm Correctness (ALG-COR). Each component includes detailed fields and rating options used by annotators.

E Prompt templates

Figures 10 and 11 show the exact prompts used for code generation and editorial generation in our experiments. The wording below is preserved verbatim; only formatting has been changed to improve readability.

Editorial Annotation Rubric

1. Problem Understanding (PU)

Purpose: Verify that the editorial accurately captures every essential detail of the problem statement, without misinterpreting constraints, omitting crucial subtleties, or introducing misleading additions.

PU-W — Wrong crucial detail Does the editorial assert something that changes the problem's meaning? Options: *Yes / No*. If Yes, specify whether the misinformation is *explicit* or *implicit*.

PU-M — Missing crucial detail Does the editorial omit a constraint or subtlety that affects the meaning of the problem? Options: *Yes / No*. If Yes, note whether the missing information is *explicit* or *implicit* in the statement.

PU-X — Irrelevant or misleading detail Extra statements that do not affect correctness but muddy understanding. Options: *None / Minor / Major*.

PU-D — Problem Understanding Difficulty Annotator's assessment of how difficult the problem is to understand. Options: scale 0–5, where 0 = very clear and 5 = extremely difficult.

2. Algorithm Description (ALG)

Purpose: Assess whether the editorial presents a coherent and appropriate high-level algorithmic idea.

ALG-TAG Algorithmic paradigm(s) described in the model-generated editorial. Annotators select one or more from: *DP, Greedy, DFS/BFS, Dijkstra, Segment Tree, Binary Lifting, FFT, Flow, Geometry, Math/Number Theory, Other*.

ALG-FREE A concise (≈ 40 words) free-text summary of the core idea in the model-generated editorial.

Golden-ALG-TAG Algorithmic paradigm(s) described in the gold-standard editorial, using the same tag set as above.

Golden-ALG-FREE A concise (≈ 40 words) summary of the corresponding idea in the gold editorial.

3. Algorithm Correctness (ALG-COR)

Purpose: Determine whether the editorial's algorithm—*as described*—solves the problem correctly and efficiently under the stated constraints.

ALG-COR Options: *Correct / Incorrect*.

If Correct **Correct Type** Options:

- **Same as golden** — matches the official solution exactly. Example: identifies the correct DP recurrence just as in the gold editorial.
- **Different from golden** — uses a different but equally valid algorithm. Example: employs a greedy method instead of DP, but still satisfies all constraints.

If Incorrect Annotators diagnose why the approach fails.

Why Incorrect Options:

1. **Wrong algorithm (WA)** The high-level idea is unsuitable; it cannot solve the full problem. Example: tries to solve a graph problem using DP on sequences.
2. **Correct algorithm but incorrect approach (WA)** The algorithmic idea is sound, but the reasoning includes a mistake that breaks correctness. Example: binary search is intended but the midpoint update never converges.
3. **Suboptimal (Likely TLE or MLE), but correct algorithm** The method returns correct answers but violates resource limits in worst-case scenarios. Example: uses an $O(n^2)$ technique where $O(n \log n)$ is required.
4. **Suboptimal (Likely TLE or MLE), and wrong algorithm** The algorithm is both inefficient and structurally unsuitable. Example: brute force or an unnecessary DP when a linear greedy method is required.

Severity of Incorrectness Options:

- **Completely wrong** No part of the solution can be salvaged; the idea is unrelated to a valid strategy.
- **Major edits needed** The structure almost works but requires a significant reworking of key components.
- **Minor edits needed** Small adjustments (e.g., off-by-one fix, missing boundary case, or initialization) restore correctness.

Figure 9: Editorial annotation rubric used for both expert evaluation. The rubric decomposes reasoning quality into problem understanding (PU), algorithm description (ALG), and algorithm correctness (ALG-COR).

Contest	Year	Source	#Teams	#Problems
CS3233 Midterm Contest	2023	NUS	25	11
CS3233 Midterm Contest	2024	NUS	15	12
CS3233 Midterm Contest	2025	NUS	16	11
ICPC Asia Pacific Championship	2024	GitHub	65	13
ICPC Asia Jakarta Regional	2017	GitHub	80	12
ICPC Asia Jakarta Regional	2018	GitHub	75	12
ICPC Asia Jakarta Regional	2019	GitHub	80	12
Total	-	-	-	83

Table 3: Contest-level composition of the dataset.

Code-generation prompts (w/oEd and w/Ed)

Without editorial (w/oEd):

System message.
You are an expert competitive programmer: write clean, efficient, and correct code that solves the problem statement within its constraints.

User message.
Here is the problem statement with time limit: <time limit> and memory limit: <memory limit>
<problem statement>
Please generate an optimized and correct solution in <programming language>.

With editorial (w/GenEd and w/GoldEd).

System message.
You are an expert competitive programmer: write clean, efficient, and correct code that solves the problem statement within its constraints and strictly follows the editorial instructions.

User message.
Here is the problem statement with time limit: <time limit> and memory limit: <memory limit>
<problem statement>
Editorial guidelines (follow these exactly):
<editorial text>
Please generate an optimized and correct solution in <programming language>.

Figure 10: Code-generation prompts used in our evaluation. The top prompt corresponds to direct problem-to-code generation (**w/oEd**), while the bottom prompt conditions code generation on an editorial (**w/GenEd** and **w/GoldEd**).

Editorial-generation prompt

System message.
You are an expert competitive programmer. Your task is to write a clear and detailed editorial for the given problem. Begin by restating the problem in your own words. Then, walk through the key ideas and algorithmic approach needed to solve it. Explain each step carefully — think step by step, and highlight any non-trivial reasoning. Avoid writing actual code. If helpful, include brief pseudocode to illustrate the logic. Focus on understanding and explanation. Your goal is to teach the reader how to approach and solve the problem, not just provide an answer. Make sure the solution respects the given time and memory limits. Finally, explain why your solution is correct and how it handles all possible cases.

User message.
Here is a competitive programming problem with time limit: <time limit> and memory limit: <memory limit>
<problem statement>
Please generate a detailed editorial.

Figure 11: Editorial-generation prompt used to elicit model-written editorials. Models are instructed to explain the algorithmic solution in natural language without producing code.

F Model Cards and Inference Setup

We evaluate a total of 19 language models spanning proprietary and open-weight systems. Unless otherwise noted, all models are evaluated in a strictly single-shot regime: each model produces at most one editorial (when applicable) and one code submission per problem and setting. We do not perform test-time sampling, majority voting, or iterative refinement; reported pass@1 results therefore reflect a single shot completion per model, problem, and condition.

Proprietary models are queried via their official APIs, using provider-recommended or default inference settings wherever possible. Open-weight models are served either locally (via vLLM or HuggingFace Transformers) or through hosted providers (OpenRouter, DeepInfra). To ensure comparability across heterogeneous backends, we favor deterministic decoding (e.g., greedy decoding or zero temperature) unless the model authors explicitly recommend otherwise.

For some proprietary APIs, exact maximum token limits are not publicly specified or are managed dynamically by the provider; in these cases, the corresponding entries are marked as “–” in Table 4. Throughout the table, token limits are reported in thousands ($k = 1,000$) for compactness.

Table 4 summarizes the exact model identifiers, serving backends, public HuggingFace endpoints (when applicable), citations, token limits, and inference configurations used in our experiments.

G Python vs. C++ Performance

Although competitive programming is overwhelmingly conducted in **C++** due to its performance and memory guarantees, we additionally evaluate a subset of models in **Python** to assess language sensitivity.

Figure 12 compares pass@1 for Python and C++ across all 83 problems under the three editorial settings (w/oEd, w/GenEd, w/GoldEd). Across nearly all models and settings, **C++ consistently outperforms Python**.

This pattern is expected in ICPC-style problems, which frequently rely on tight constant factors, low-level data structures, and explicit memory control. Python implementations often incur TLE or excessive overhead even when the underlying algorithm is correct. Accordingly, we treat **C++** as the primary evaluation language and report Python results only as supplementary analysis.

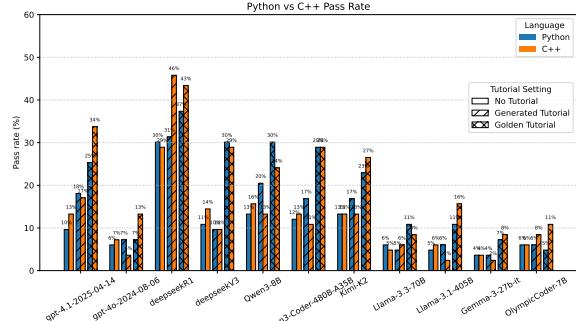


Figure 12: Pass@1 comparison between Python and C++ across editorial settings.

H Per-Contest Virtual Rank Percentiles

Figures 13–16 report virtual rank percentiles separately for each contest. Absolute rank percentiles vary across contests due to differences in the problem set, number of teams, so percentiles should be interpreted relative to each contest’s field. Despite this heterogeneity, the qualitative pattern is consistent: w/GoldEd yields the largest and most reliable upward rank shifts relative to w/oEd, w/GenEd produces smaller and higher-variance changes, and only a small subset of models attains high human-relative rank in any single contest.

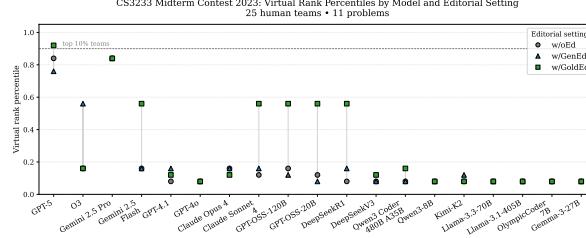
I Detailed Failure Analysis by Error Type

Overall failure mix. Across all models and settings, Wrong Answer (WA) is the single largest failure type in Figure 17. For almost every bar, the WA segment is the biggest component, which indicates that many failed submissions do compile and run but implement an incorrect or incomplete algorithm. Compile Error (CE) is also substantial: for several models, especially O3, GEMINI 2.5 FLASH, QWEN3-8B, and OLYMPICCODER-7B, CE is the second largest or even co-dominant mode. These CEs include both ordinary C++ syntax and type errors and the no-code behaviors described below. Time Limit Exceeded (TLE) and Runtime Error (RTE) account for a smaller share of failures, while Memory Limit Exceeded (MLE) is rare.

Effect of editorials on failure types. Changing the editorial setting affects both how often models fail and how they fail. Moving from **w/oEd** to **w/GenEd** typically shifts some mass from CE and RTE into WA: more runs produce code that compiles and runs, but the resulting programs still fail on hidden tests. In the **w/GoldEd** setting, total failure counts drop noticeably for most models and

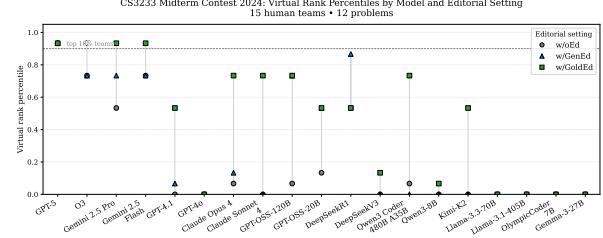
Short name(s)	Backend	HuggingFace endpoint(s)	Citation(s)	Max tokens	Configurations
Closed-source models					
GPT-5 (2025-08-07), O3 (2025-04-16)	OpenAI API	–	(OpenAI, 2025b; OpenAI et al., 2025)	–	GPT-5 uses high-effort reasoning, O3 uses medium effort; other parameters at provider defaults.
Gemini-2.5-Pro, Gemini-2.5-Flash	Google Gemini API	–	(Comanici et al., 2025)	–	Deterministic decoding with temperature = 0; thinking_budget = -1.
GPT-4.1 (2025-04-14), GPT-4o (2024-08-06)	OpenAI API	–	(OpenAI, 2025a; OpenAI et al., 2024)	–	Provider default decoding and safety settings.
Claude-Opus-4, Claude-Sonnet-4	Anthropic API	–	(Anthropic, 2025)	32k	thinking_budget = 22k; other parameters at provider defaults.
Open-weight models					
GPT-OSS-120B, GPT-OSS-20B	OpenRouter (120B); local vLLM (20B)	gpt-oss/gpt-oss-120b; gpt-oss/gpt-oss-20b	–	65k	temperature = 0.0; reasoning_effort = "medium".
DeepSeek-R1, DeepSeek-V3	DeepSeek official API	deepseek-ai/DeepSeek-R1; deepseek-ai/DeepSeek-V3	(DeepSeek-AI et al., 2025a,b)	–	Recommended reasoning (R1) and base (V3) endpoints with provider defaults.
Qwen3-Coder- 480B-A35B; Kimi-K2; Llama-3.1-405B; Llama-3.3-70B; Gemma-3-27B-it	DeepInfra API	Qwen/Qwen3-Coder-480B- A35B-Instruct; moonshotai/Kimi-K2- Instruct; meta-llama/Meta-Llama-3.1- 405B-Instruct; meta-llama/Llama-3.3-70B- Instruct; google/gemma-3-27b-it	(Yang et al., 2025a; Team et al., 2025b; Grattafiori et al., 2024; Team et al., 2025a)	65k	Shared config; temperature = 0.0; do_sample = False (greedy).
Qwen3-8B	Local (Transformers)	Qwen/Qwen3-8B	(Yang et al., 2025a)	32k	Transformers defaults; deterministic decoding.
OlympicCoder- 7B	Local	open-r1/OlympicCoder-7B	(Penedo et al., 2025)	32k	Authors' recommended settings: do_sample = True, top_k = 50, top_p = 0.95, temperature = 0.7.

Table 4: Model cards and inference configuration for all systems in Table 1. Token limits are reported in thousands (k = 1,000). All models are evaluated with a single completion per problem and editorial setting.

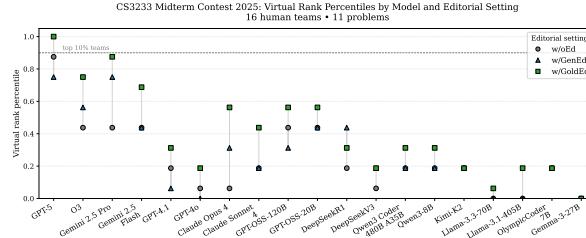


(a) CS3233 Midterm 2023 (25 teams, 11 problems).

Figure 13: Per-contest virtual rank percentiles by model and editorial setting (CS3233 midterms).

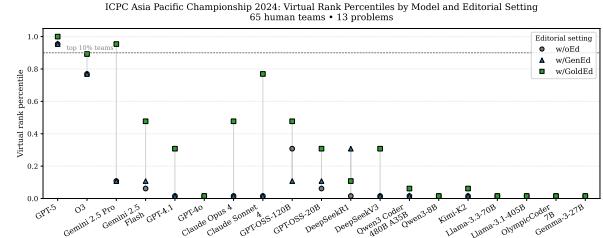


(b) CS3233 Midterm 2024 (15 teams, 12 problems).



(a) CS3233 Midterm 2025 (16 teams, 11 problems).

Figure 14: Per-contest virtual rank percentiles (CS3233 and ICPC APAC).



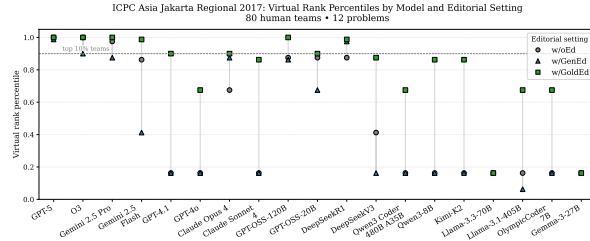
(b) ICPC APAC 2024 (65 teams, 13 problems).

WA segments shrink, which shows that correct human editorials remove many misplanned solutions. However, CE remains non-trivial, especially for models that already struggle with compilation such as O3 and GEMINI 2.5 FLASH. For these systems, even gold editorials do not reliably prevent syntax errors, truncated outputs, or off-spec completions.

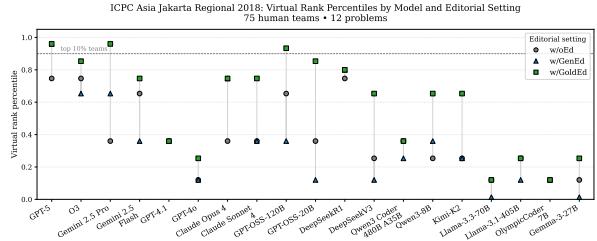
Reasoning-limited versus implementation-limited models. The per-model stacks in Figure 17 also reveal systematic differences between model families. Many open-weight coders, such as GPT-OSS-20B/120B, Qwen3-8B, Kimi-K2, Llama-3.1/3.3, and Gemma-3-27B, have failures where WA is the largest segment but CE is also a sizable fraction in all three settings. These models usually manage to reach compilation, yet a large share of runs still encode an incorrect algorithm or miss important edge cases, and a non-trivial number fail already at compile time. In contrast, the strongest closed models such as GPT-5, O3, and Gemini 2.5 Pro have relatively small CE segments and a mix of WA and TLE, which suggests that their main residual bottlenecks are logical errors and efficiency rather than basic syntax. O3 and GEMINI 2.5 FLASH are notable outliers: their bars contain unusually large CE blocks in every setting, indicating that unstable or incomplete code generation is itself a major source of failure.

No-code and no-output pathologies. Among all 83 problems, 20 models, and 3 editorial settings ($83 \times 19 \times 3 = 4,731$ model–problem–setting runs), we observe a small but qualitatively distinct set of failures where the model never really submits a candidate program. Using the raw logs, we identify 91 completions that are labeled COMPILER ERROR but contain no compilable C++ at all. In these cases the model either explicitly declines to solve the problem (for example, “The problem is very hard. A correct solution that works within the limits is not feasible to provide in this format.”), produces an editorial-style explanation that stops just before the “Reference implementation” section, or outputs a full solution in another language such as Python while the prompt requests C++. These no-code CEs are heavily concentrated in O3 and GEMINI 2.5 FLASH, with smaller numbers for GPT-OSS-20B, GPT-OSS-120B, GPT-5, QWEN3-8B, and OLYMPICCODER-7B, and they occur disproportionately on the hardest contests (ICPC APAC 2024 and CS3233 midterms).

We also find 12 runs that compile but terminate with a runtime error without producing any output on our judge, effectively giving no answer at execution time. These cases arise mainly for GPT-5, O3, GEMINI 2.5 FLASH, and OLYMPICCODER-7B on the most challenging problems. Qualitatively, they behave similarly to the no-code CEs: the model never produces a testable solution, as opposed to producing a plausible but wrong pro-



(a) ICPC Jakarta 2017 (80 teams, 12 problems).



(b) ICPC Jakarta 2018 (75 teams, 12 problems).

Figure 15: Per-contest virtual rank percentiles (ICPC Jakarta regionals).

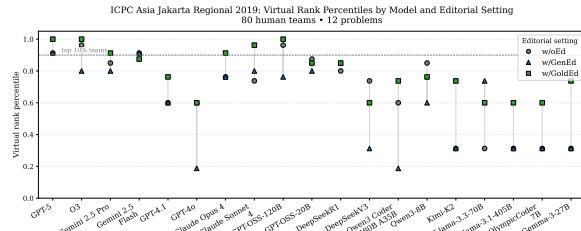


Figure 16: ICPC Jakarta 2019: Virtual rank percentiles by model and editorial setting (80 teams, 12 problems).

gram. In the main figures we conservatively count such behaviors under the standard CE or RTE buckets, but they highlight that a small share of failures correspond to genuine non-attempts (or off-spec attempts) rather than buggy implementations. Models such as QWEN3-8B and QWEN3-CODER-30B-A3B occasionally exhibit related behavior by exhausting their token budgets on long “thinking” traces or hallucinated analyses without ever emitting code, which our pass@1 metric naturally penalizes.

J Editorial length statistics

To quantify how model-generated editorials differ from human-written gold editorials in level of detail, we compare their word counts across all problems and models.

Across all 19 models, model-generated editorials are systematically longer than the corresponding gold editorials. In other words, when models are asked to “write an editorial”, they typically produce more expansive and didactic explanations than the contest editorials themselves, even when both describe essentially the same algorithmic plan. This quantitative pattern complements our qualitative observation that model editorials often trade concision for more step-by-step reasoning.

K CS3233 2025 Midterm Contest Full Annotations

This appendix provides the complete human annotations for the CS3233 2025 Midterm qualitative case study. Annotations follow the rubric in Appendix D and are applied to model-generated editorials from the generated-editorial setting. Table 5 summarizes problem understanding, including wrong or missing crucial details, misinformation type, misleading severity, and annotator comments. Tables 6 and its continuation report the algorithmic paradigms assigned to each editorial and the corresponding freeform summaries for both the model and gold editorials. Table 7 reports editorial-level algorithmic correctness, diagnostic failure categories, and the final judge verdicts of the generated code, enabling direct comparison between editorial reasoning quality and downstream execution outcomes.

L LLM-as-a-Judge

L.1 LLM-as-a-Judge Prompts

We evaluate LLM-generated editorials using an LLM-based judge with a fixed system prompt and a templated user prompt. Figures 19–22 show the full prompt used for LLM-based editorial evaluation. We instantiate the template by inserting the problem statement, the gold editorial, and the model-generated editorial.

All editorial evaluations are performed using `google/gemini-3-pro-preview` accessed through OpenRouter⁵, with a maximum generation budget of 65,536 tokens and high reasoning effort enabled.

L.2 Additional agreement results

For completeness, we additionally evaluate agreement between the expert annotator and the Gemini 3 Pro judge on auxiliary rubric fields not used

⁵<https://openrouter.ai>

Problem	Model	PU-W	Misinformation Type	PU-M	Misinformation Type	PU-X	PU-D	PU-Comments
ARTS AND COMPUTING STUDENTS								
DeepSeek R1	No	–	No	–	Minor	2	–	The understanding becomes wrong under “Key Insights”, it assumes the student can be moved rather than shifted
GPT 5	No	–	No	–	None	2	–	
BRILLIANCE OF WINGS								
DeepSeek R1	No	–	No	–	None	1	–	
GPT 5	No	–	No	–	None	1	–	
CHAINED MAIMAI SLIDES								
DeepSeek R1	Yes	Explicit	Yes	Explicit	Major	2	–	
GPT 5	No	–	No	–	None	2	–	
DEPENDENCY FLOOD								
DeepSeek R1	No	–	No	–	Minor	2	–	
GPT 5	No	–	No	–	None	2	–	
EASYGOING WORKPLACE								
DeepSeek R1	No	–	No	–	None	0	–	missing the $B_i < i$, i.e. superior is always the smaller number
GPT 5	No	–	No	–	None	0	–	missing the $B_i < i$ written in the I/O section, i.e. superior is always the smaller number
FICKETTS CONJECTURE FOR POLYOMINOES								
DeepSeek R1	No	–	No	–	Minor	2	–	it says: “max fraction of overlapping border cells between A and A_1 ”. it should be: “max fraction of border cells between A and A_1 , within overlapping cells”
GPT 5	No	–	No	–	None	2	–	
GEORGETTE ME GEORGETTE YOU								
DeepSeek R1	No	–	No	–	None	0	–	
GPT 5	No	–	No	–	None	0	–	
HUNGRY PIPLUPS								
DeepSeek R1	No	–	No	–	None	2	–	
GPT 5	No	–	No	–	None	2	–	
IMPERFECTION								
DeepSeek R1	No	–	No	–	None	0	–	
GPT 5	No	–	No	–	None	0	–	
JAUNT THROUGH THE GARDEN								
DeepSeek R1	No	–	No	–	None	0	–	
GPT 5	No	–	No	–	None	0	–	
KEEP THE ORDERING								
DeepSeek R1	No	–	No	–	None	0	–	
GPT 5	No	–	No	–	None	0	–	

Table 5: Problem understanding annotations for the CS3233 2025 Midterm case study. PU W and PU M indicate the presence of wrong or missing crucial problem details, with misinformation type recorded when applicable. PU X records misleading severity and PU D is the annotator-rated understanding difficulty. PU Comments provides brief annotator notes.

Problem	Model	ALG-TAG	ALG-TAG-OTHER	ALG-FREE	Gold-ALG-TAG	Gold-ALG-TAG-OTHER	Gold-ALG-FREE	ALG-Comments
ARTS AND COMPUTING STUDENTS								
DeepSeek R1	Greedy			Count the A and C for general feasibility. Scan left-to-right and right-to-left tracking the count difference. Constructing by re-arranging the whole seating with AC pair first, followed by remaining As and empty seats.	DP, Greedy		DP with possibility to either pair with next student or no pairing. Then check whether the possible arrangement has enough seating. Alternatively, split the ACCA arrangement, the remaining can be picked greedily for any adjacent AC and CA. The possible arrangement has the minimum number of seatings needed.	–
GPT 5	Greedy			Count A, C and ‘.’ for feasibility. Remove ‘.’, greedily pair the different consecutive chars from left-to-right. Re-iterate to pair the unpaired chars: A with A, unfeasible for C.			The generated editorial does not consider cases where picking arbitrarily one of 2 forms (AC or CA) would end up in a longer arrangement	
BRILLIANCE OF WINGS								
DeepSeek R1	Other	Set Manipulation (inclusion-exclusion)		Total number of edges that are present in the first tree but not in second tree	Other	Graph Theory (Tree), Set Manipulation (inclusion-exclusion)	Each step require edge deletion from tree 1 and edge insertion from tree 2. Hence, minimum step is total edge - shared edges.	–
GPT 5	Other	Set Manipulation (inclusion-exclusion)		Create hash set or balanced BST for tree-2, then iterate edges of tree-1 and check if it is in the set. Output (N-1) - edge_in_set.			–	
CHAINED MAIMAI SLIDES								
DeepSeek R1	DP, Two-Pointer / Sliding Window			–	Other	MCMF	–	–
GPT 5	Other	MCMF		–			–	–
DEPENDENCY FLOOD								
DeepSeek R1	DFS/BFS (Graph Traversal), Other	Graph Theory		Graph traversal for: (1) constructing prefix and suffix count, (2) updating accepted query. The observation shows that there is an upper bound of this update, K, so not all the queries are update-operation.	DFS/BFS (Graph Traversal), Other	Graph Theory	Graph traversal for: (1) constructing prefix and suffix count, (2) updating accepted query. The observation shows that there is an upper bound of this update, K, so not all the queries are update-operation.	LLM-editorial says DP but only the memoization part, no relation in the dimension; surprisingly, the generated editorial is easier to grasp
GPT 5	DFS/BFS (Graph Traversal), Other	Graph Theory		Graph traversal for: (1) constructing prefix count from root to particular node, (2) updating accepted query.			–	–
EASYGOING WORKPLACE								
DeepSeek R1	DFS/BFS (Graph Traversal), Greedy			Two traversals where the first one computes the range of feasible K for each subtree and also each node's feasibility to slack off. Second traversal to assign so that only K workers work, the answer construction follows this decision.	Binary Search, Math/Number Theory		If K is feasible, there should exist an x where $(x, \dots, N, x-1, \dots, 1)$ is the answer. x could be found by binary-search or iterative construction.	got WA by following the golden editorial.
GPT 5	DP, DFS/BFS (Graph Traversal), Other			DP + Disjoint Set Union, using graph traversal.			–	–
FICKETTS CONJECTURE FOR POLYOMINOES								
DeepSeek R1	Other	Brute Force		Try all possible rotations and translations of A1, then count by traversing each cell one-by-one.	Geometry		Pre-compute the boundary cells, then consider all rotations and translations, i.e., all possible A1. When counting the value, use this precomputed coordinates, ensuring $O(m+n)$ per A1.	–
GPT 5	Other	Geometry		Build row and column intervals, then count the border cells via erosion of 3x3 blocks. Translations are then considered exhaustively for computing the best ratio.			–	–
GEORGETTE ME GEORGETTE YOU								
DeepSeek R1	DP			Same with golden, but the DP memoization is referred as involutions.	DP		The permutation P equals inverse permutation Q when all elements satisfy $P_i = k$ and $P_k = i$ for all (i, k) . By counting the $P=Q$ permutations using DP, then the answer is all permutations minus $P=Q$ permutations.	LLM-editorial uses 1-dimensional DP, like prefix-sum, but never mention this paradigm is DP.
GPT 5	DP			Precompute the factorial and the involutions of 1-cycle and 2-cycles. Answer each query as factorial minus involutions.			–	–

Table 6: Algorithmic paradigm annotations for the CS3233 2025 Midterm case study. ALG TAG and ALG TAG OTHER list the paradigms attributed to each model-generated editorial, and ALG FREE provides a freeform summary of the described approach. Gold columns report the corresponding gold editorial tags and summary. ALG Comments records brief annotator notes on the algorithm description.

Problem	Model	ALG-TAG	ALG-TAG- OTHER	ALG-FREE	Gold- ALG-TAG	Gold- ALG-TAG- OTHER	Gold-ALG-FREE	ALG-Comments	
HUNGRY PIPLUPS									
DeepSeek R1	Greedy			Simulating the problem statement, with small tweaks without any effect to the overall solution complexity	Segment Tree, Binary Search, Greedy		Construct a segment tree of N, where each node stores value count of A_i , exploiting the limits of $1 \leq A_i \leq 500$. Thus, each query (update and merging), only cost $\mathcal{O}(\max A_i \log N)$.	Model is the expected naive or BF version	
GPT 5	Binary Search, Greedy, Segment Tree			Segment-tree like approach by managing $\max A_i$ (500) ordered sets, each query is done by $\mathcal{O}(\log N)$ operation. The output is counted greedily.				–	
IMPERFECTION									
DeepSeek R1	Segment Tree, Bit Manipulation, Other	Fenwick Tree (Binary Indexed Tree)		Precompute fixed cost of deleting all elements greater than any given x, and maintain variable cost to efficiently delete elements smaller than x. The answer is the minimum of fixed + variable cost for all x.	Segment Tree, Bit Manipulation, Other	Binary Indexed Tree (Fenwick Tree)	Segment Tree, Bit Manipulation, Other	Finding two algorithms to: (1) compute the sum of costs to delete K cheapest elements, and (2) remove the maximum element from consideration. Both can be solved using BIT or Segment Tree data structures.	feels like different algo but actually similar
GPT 5	Segment Tree, Bit Manipulation, Other	Binary Indexed Tree & Fenwick Tree		Group by value to precompute cost to delete for a fixed value, iterate the unique values in increasing order, saving the best answer. BIT/Fenwick tree is required to maintain frequency and cost.				–	
JAUNT THROUGH THE GARDEN									
DeepSeek R1	DP			1-dimensional DP	DP		2-dimensional DP	The generated editorial suddenly changed to 1-dimensional DP while previously already mentioning the DP state is 2 dimensions. The gold editorial mentions an $\mathcal{O}(N)$ solution rather than the $\mathcal{O}(N^2)$ DP; this formula seems to be what the generated editorial tries to achieve.	
GPT 5	Math/Number Theory, DP			DP for precomputation of combinatorics, base-2 powers, and the inversion. Iterate in linear time to compute the answer based on the formula.				–	
KEEP THE ORDERING									
DeepSeek R1	Other, Binary Search			Precompute Construct valid N, compute and check N^2 , (brute-force/BFS) found by binary searching the index in this precomputed list.	Other, Binary Search	Precompute (brute-force/DFS)	Construct valid N, compute and check N^2 , save in precomputation. The answer is found by binary searching the index in this precomputed list.	–	
GPT 5	Other, Binary Search			Precompute Construct valid N, compute and check N^2 , (brute-force) save in precomputation. Answer is binary search of finding index.				–	

Table 6: Algorithmic paradigm annotations for the CS3233 2025 Midterm case study (continued).

Problem	Model	ALG COR	Correct type	Why incorrect	Severity	Annotator comments	Final verdict
ARTS AND COMPUTING STUDENTS							
DeepSeek R1		INCORRECT	–	Wrong algorithm	Completely wrong	Shift ≠ Move	WA
GPT 5		INCORRECT	–	Wrong algorithm	Major edits needed	Need to reconsider a valid case of: "A. AC AA CC AA"	WA
BRILLIANCE OF WINGS							
DeepSeek R1		CORRECT	Same as golden	–	–	–	PASS
GPT 5		CORRECT	Same as golden	–	–	–	PASS
CHAINED MAIMAI SLIDES							
DeepSeek R1		INCORRECT	–	Wrong algorithm	Completely wrong	Hallucinated	RTE
GPT 5		CORRECT	Same as golden	–	–	Seems to have same core idea despite differing in the complexity analysis.	PASS
DEPENDENCY FLOOD							
DeepSeek R1		CORRECT	Same as golden	–	–	–	WA
GPT 5		INCORRECT	–	Wrong algorithm	Minor edits needed	The model only keeps track of one count from root to particular node but not from particular node to leaves.	WA
EASYGOING WORKPLACE							
DeepSeek R1		CORRECT	Different from golden	–	–	–	TLE
GPT 5		CORRECT	Different from golden	–	–	The solution does not rely on the fact that subordinate always have higher index.	PASS
FICKETTS CONJECTURE FOR POLYOMINOES							
DeepSeek R1		INCORRECT	–	Suboptimal (Likely TLE or MLE), but correct algorithm	Major edits needed	should be TLE as it considers $O(R \cdot C)$, rather than $O(R+C)$, i.e. there can only be 2 border-cells per row or per column	TLE
GPT 5		CORRECT	Different from golden	–	–	Most parts are similar; rather than straightforward counting, the model proposes a count through slope arrays.	RTE
GEORGETTE ME GEORGETTE YOU							
DeepSeek R1		CORRECT	Same as golden	–	–	–	PASS
GPT 5		CORRECT	Same as golden	–	–	–	PASS
HUNGRY PIPLUPS							
DeepSeek R1		INCORRECT	–	Suboptimal (Likely TLE or MLE), but correct algorithm	Major edits needed	Does not make use or ensure the range could be calculated only from the range of A_i values, yielding $O(N)$ instead of expected $O(\log N)$	TLE
GPT 5		CORRECT	Same as golden	–	–	–	PASS
IMPERFECTION							
DeepSeek R1		CORRECT	Same as golden	–	–	–	NA
GPT 5		CORRECT	Same as golden	–	–	–	PASS
JAUNT THROUGH THE GARDEN							
DeepSeek R1		INCORRECT	–	Wrong algorithm	Minor edits needed	The golden editorial mention there exist an $O(N)$ solution rather than $O(N^2)$ DP detailed in the editorial. This formula seems to be what the generated editorial tries to achieve.	WA
GPT 5		CORRECT	Different from golden	–	–	The gold editorial leaves the $O(N)$ solution for exercise to the reader.	PASS
KEEP THE ORDERING							
DeepSeek R1		CORRECT	Same as golden	–	–	Implementation inefficiency as it involves conversion to string for every check	TLE
GPT 5		CORRECT	Same as golden	–	–	–	PASS

Table 7: Algorithmic correctness annotations and execution outcomes for the CS3233 2025 Midterm case study. ALG COR indicates whether the editorial-level algorithm is correct under contest constraints. Correct type distinguishes editorials that match the gold approach from those that use a different but valid approach. For incorrect editorials, we report the diagnosed failure mode and severity. Final verdict is the judge outcome of the code generated after the editorial in the generated-editorial setting.

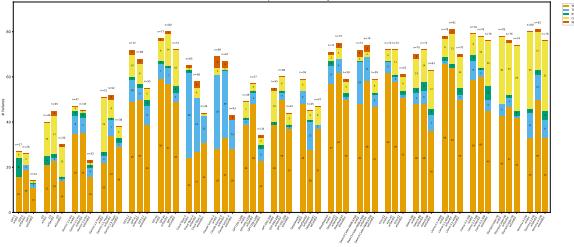


Figure 17: Failure counts by model and editorial setting. For each (model, setting) pair we show the number of failed submissions decomposed into Wrong Answer (WA), Time Limit Exceeded (TLE), Runtime Error (RTE), Compile Error (CE), and Memory Limit Exceeded (MLE). The label above each bar indicates the total number of failures n in that condition.

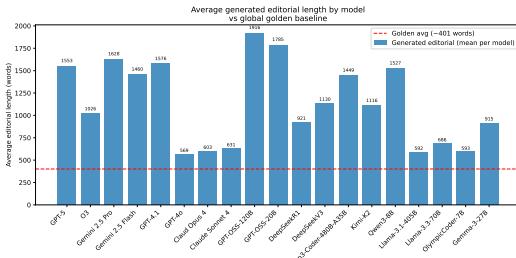


Figure 18: Word-count comparison between human-written gold editorials and model-generated editorials, aggregated over all problems. For each model we plot the distribution of word counts for its w/GenEd editorials and the corresponding gold editorials. Across all models, model-generated editorials are consistently longer than the gold ones, indicating that they tend to provide more verbose, step-by-step explanations.

as primary signals in our analysis, including the PU dimensions.

Table 8 reports raw agreement and Cohen’s κ for these auxiliary fields. While several PU fields exhibit high raw agreement, their chance-corrected agreement is low or near zero. This behavior reflects a combination of strong class imbalance and limited sample size, under which Cohen’s κ is known to be unstable. Accordingly, we do not rely on these fields in our main analysis and report them here for completeness.

L.3 Additional judge-based diagnostics

The main paper uses the judge primarily for editorial-level correctness (ALG-COR) and its re-

Table 8: Agreement between the expert annotator and the Gemini 3 Pro judge on auxiliary rubric fields.

Rubric field	n	Agr.	κ
PU-W (Yes vs. No)	22	0.909	-0.048
PU-M (Yes vs. No)	22	0.955	0.000
PU-X (None/Minor/Major)	22	0.818	0.000
PU-D (0–5)	22	0.318	-0.006

lationship to downstream outcomes. Here we report additional judge fields that help contextualize *why* editorials fail, and how often failures stem from problem understanding versus algorithmic reasoning.

Problem-understanding errors are uncommon compared to algorithmic errors. Figure 23 (top-left) shows that most generated editorials do not exhibit *crucial* wrong or missing problem details, indicating that gross misinterpretation is relatively rare. However, the fraction of such errors increases for open-weight models. In competitive programming, correct problem understanding is a strict prerequisite rather than an optional skill; even a small error rate at this stage is severe, as it renders all downstream reasoning and implementation invalid. Thus, while less frequent than algorithmic failures, problem-understanding errors remain a non-trivial and important limitation.

Most problems are rated as medium to understand. Figure 23 (bottom-left) shows the distribution of PU-D scores, with most mass concentrated at intermediate values. This supports our interpretation that the dominant difficulty in this benchmark is deriving a correct and efficient algorithm, not parsing the statement.

Tag alignment is a coarse but imperfect proxy for correctness. Figure 23 (bottom-right) reports the overlap between generated and gold algorithm tags. While exact matches are common for stronger models, partial overlap is also frequent and “no overlap” remains non-trivial for weaker models. Consistent with Appendix L.5, tag overlap is a weaker predictor of downstream success than ALG-COR.

LLM-as-a-judge prompt (editorial evaluation)

System message.

You are an expert competitive-programming judge and annotation assistant.

Your job is to:

- 1) Read the problem statement, the gold editorial, and the LLM-generated editorial.
- 2) Compare the LLM-generated editorial against the problem statement and the gold editorial.
- 3) Assign labels and short explanations according to the rubric and JSON schema provided in the user message.

STRICT FORMAT RULES (MUST OBEY):

- You MUST output exactly ONE JSON object.
- The JSON MUST be syntactically valid and parseable by json.loads in Python.
- The JSON MUST match the schema in the user message (all keys present).
- Use only the specified categorical values (case-sensitive).
- Use null where a field does not apply.
- Do NOT include any extra commentary, markdown, backticks, or text outside the JSON.

User message.

Goal

Your task is to evaluate editorials, not code. An editorial is a written explanation of the algorithm and the reasoning behind to solve a competitive-programming problem.

For each competitive-programming problem, you will assess TWO write-ups:

- 1) The gold editorial provided by the problem setter or tester (reference solution).
- 2) An LLM-generated editorial (the one you are evaluating).

You will assign ratings/labels in three categories:

- 1) Problem Understanding (PU)
- 2) Algorithm Description (ALG)
- 3) Algorithm Correctness (ALG-COR)

Your primary evaluation target is the LLM-generated editorial. The gold editorial is used only as a reference for the intended solution.

INPUTS

Problem statement:

«PROBLEM_STATEMENT»

Gold editorial:

«GOLD_EDITORIAL»

LLM-generated editorial (the one you are evaluating):

«LLM_EDITORIAL»

1. PROBLEM UNDERSTANDING (PU)

Purpose: Decide whether the LLM editorial correctly understands the problem statement.

You are checking only the LLM-generated editorial here.

• PU-W – Wrong crucial detail

- **Question:** Does the LLM editorial assert something that changes the problem's meaning?
 - "value": "Yes" or "No"
 - If "value" = "Yes", then:
 - * "type":
 - "explicit" if the wrong detail is directly stated
 - "implicit" if it is strongly implied but not literally stated
 - * "notes": Short explanation describing the wrong detail and why it is crucial
 - If "value" = "No":
 - * "type": null
 - * "notes": Short confirmation that there are no crucial wrong details

• PU-M – Missing crucial detail

- **Question:** Does the LLM editorial omit a constraint or subtlety that affects correctness?
 - "value": "Yes" or "No"
 - If "value" = "Yes", then:
 - * "type":
 - "explicit" if the missing info is explicitly present in the statement
 - "implicit" if it is only implied but important
 - * "notes": Short explanation of the missing detail and why it matters
 - If "value" = "No":
 - * "type": null
 - * "notes": Short confirmation that no crucial detail is missing

Figure 19: LLM-as-a-judge prompt for editorial evaluation. (part 1)

- **PU-X – Irrelevant / misleading detail**
 - **Question:** Does the LLM editorial add extra statements that do not change correctness but muddy understanding?
 - "value": one of "None", "Minor", "Major"
 - "notes": Short explanation describing the extra or misleading material (if any)

- **PU-D – Problem understanding difficulty**

- **Question:** How difficult is the *original problem statement* to understand (not the editorial)?
 - "value": integer in [0, 5]
 - * 0 = very clear — no difficulty
 - * 5 = extremely difficult
 - "rationale": Short explanation of why this difficulty rating was chosen
-

2. ALGORITHM DESCRIPTION (ALG)

Purpose: Describe the high-level idea/algorithm in both editorials.
You must label the LLM editorial and the gold editorial separately.

For the LLM-generated editorial:

- **ALG-TAG**

- **Purpose:** High-level paradigms used in the LLM-generated editorial
- "ALG-TAG": array of one or more strings
- Each element must be chosen from the following fixed list (case-sensitive):
 - * "DP"
 - * "Greedy"
 - * "DFS/BFS"
 - * "Dijkstra"
 - * "Segment Tree"
 - * "Binary Lifting"
 - * "FFT"
 - * "Flow"
 - * "Geometry"
 - * "Math/Number Theory"
 - * "Other"
- If and only if "Other" is included, you MUST also provide extra tags in "ALG-TAG-OTHER"

- **ALG-TAG-OTHER**

- "ALG-TAG-OTHER": array of strings (possibly empty)
- If "Other" is in "ALG-TAG", then:
 - Include one or more extra algorithm tags, for example:
 - "Meet-in-the-middle"
 - "Bitmask brute force"
 - "2-SAT via implication graph"
 - "Game theory / Grundy numbers"

- If "Other" is NOT in "ALG-TAG", then:
 - Set this field to []

- **ALG-FREE**

- "ALG-FREE": one or two concise sentences (approximately 40 words total)
- Purpose: summarise the core idea of the LLM-generated editorial

- **Gold editorial (reference solution)**

- **Golden-ALG-TAG**

- High-level paradigms used in the gold editorial
- Same fixed tag list as "ALG-TAG":
 - "DP"
 - "Greedy"
 - "DFS/BFS"
 - "Dijkstra"
 - "Segment Tree"
 - "Binary Lifting"
 - "FFT"
 - "Flow"
 - "Geometry"
 - "Math/Number Theory"
 - "Other"

- If and only if "Other" is included, you MUST also provide extra tags in "Golden-ALG-TAG-OTHER"

Figure 20: LLM-as-a-judge prompt for editorial evaluation. (part 2)

- **Golden-ALG-TAG-OTHER**
 - "Golden-ALG-TAG-OTHER": array of strings (possibly empty)
 - If "Other" is in "Golden-ALG-TAG", include one or more extra algorithm tags
 - Otherwise, set this field to []
 - **Golden-ALG-FREE**
 - "Golden-ALG-FREE": one or two concise sentences (approximately 40 words total)
 - Purpose: summarise the core idea of the gold editorial
-
- 3. ALGORITHM CORRECTNESS (ALG-COR)**
-
- Purpose: Evaluate whether the LLM editorial's algorithm, as described, solves the problem correctly and efficiently under the given constraints.
You are judging the LLM-generated editorial here.
- **ALG-COR.overall (overall correctness)**
 - "overall": "Correct" or "Incorrect"
 - Interpret "Correct" as: the described algorithm solves all valid inputs within time/memory limits.
 - Interpret "Incorrect" as: there exists some valid input where the algorithm fails (wrong answer or infeasible resource usage).
 - **Case 1: If "overall" = "Correct"**
 - **if_correct.correct_type**
 - * "correct_type": "Same as golden" or "Different from golden"
 - * "notes": short explanation of this choice (e.g., why it matches or differs from the gold editorial).
 - * Use null for "correct_type" and "notes" only if "overall" = "Incorrect".
 - "Same as golden":
 - * The LLM editorial matches the official approach (same core algorithmic idea).
 - "Different from golden":
 - * The LLM editorial uses a different but still valid algorithm.
 - **Case 2: If "overall" = "Incorrect"**
 - **if_incorrect.why_incorrect**
 - * "why_incorrect": EXACTLY one of:
 - "Wrong algorithm"
 - "Correct algorithm but incorrect approach"
 - "Suboptimal but correct algorithm"
 - "Suboptimal and wrong algorithm"
 - null (only if "overall" = "Correct")
 - **Meaning**
 - * "Wrong algorithm": The high-level algorithmic idea itself cannot solve the full problem.
 - * "Correct algorithm but incorrect approach": The idea is sound, but there is a clear implementation or reasoning bug.
 - * "Suboptimal but correct algorithm": The algorithm would produce correct answers but is too slow or memory-heavy for the stated constraints (likely TLE/MLE).
 - * "Suboptimal and wrong algorithm": The algorithm is both incorrect on some inputs and too slow / resource-heavy.
 - **if_incorrect.severity**
 - * "severity": one of:
 - "Completely wrong"
 - "Major edits needed"
 - "Minor edits needed"
 - null (only if "overall" = "Correct")
 - **Meaning**
 - * "Completely wrong": No substantial part of the solution is reusable.
 - * "Major edits needed": Core structure is somewhat aligned but requires substantial changes.
 - * "Minor edits needed": A small fix (e.g., off-by-one, missing boundary case) would make it correct.
 - **if_incorrect.diagnosis**
 - * "diagnosis": short natural-language explanation of why the algorithm fails (e.g., specific counterexample pattern, asymptotic complexity issue).
 - If "overall" = "Correct", then:
 - Set `if_incorrect.why_incorrect`, `if_incorrect.severity`, and `if_incorrect.diagnosis` to null.
 - If "overall" = "Incorrect", then:
 - Set `if_correct.correct_type` and `if_correct.notes` to null.
- You MUST now apply these guidelines to the given problem statement, gold editorial, and LLM editorial and produce labels accordingly.

Figure 21: LLM-as-a-judge prompt for editorial evaluation. (part 3)

REQUIRED JSON OUTPUT SCHEMA

Return ONLY a single valid JSON object with EXACTLY this structure:

```
{  
    "PU": {  
        "PU-W": { "value": "Yes" or "No", "type": "explicit" or "implicit" or null, "notes": "..." },  
        "PU-M": { "value": "Yes" or "No", "type": "explicit" or "implicit" or null, "notes": "..." },  
        "PU-X": { "value": "None" or "Minor" or "Major", "notes": "..." },  
        "PU-D": { "value": 0 or 1 or 2 or 3 or 4 or 5, "rationale": "..." }  
    },  
    "ALG": {  
        "ALG-TAG": [...],  
        "ALG-TAG-OTHER": [...],  
        "ALG-FREE": "...",  
        "Golden-ALG-TAG": [...],  
        "Golden-ALG-TAG-OTHER": [...],  
        "Golden-ALG-FREE": "..."  
    },  
    "ALG-COR": {  
        "overall": "Correct" or "Incorrect",  
        "if_correct": {  
            "correct_type": "Same as golden" or "Different from golden" or null,  
            "notes": "..." or null  
        },  
        "if_incorrect": {  
            "why_incorrect": "Wrong algorithm" or "Correct algorithm but incorrect approach" or  
            "Suboptimal but correct algorithm" or "Suboptimal and wrong algorithm" or null,  
            "severity": "Completely wrong" or "Major edits needed" or "Minor edits needed" or null,  
            "diagnosis": "..." or null  
        }  
    }  
}
```

IMPORTANT: - Every key shown above MUST be present. - Use null exactly where a field does not apply, as explained.
- Do NOT output any text outside the JSON object.

Figure 22: LLM-as-a-judge prompt for editorial evaluation. (part 4)

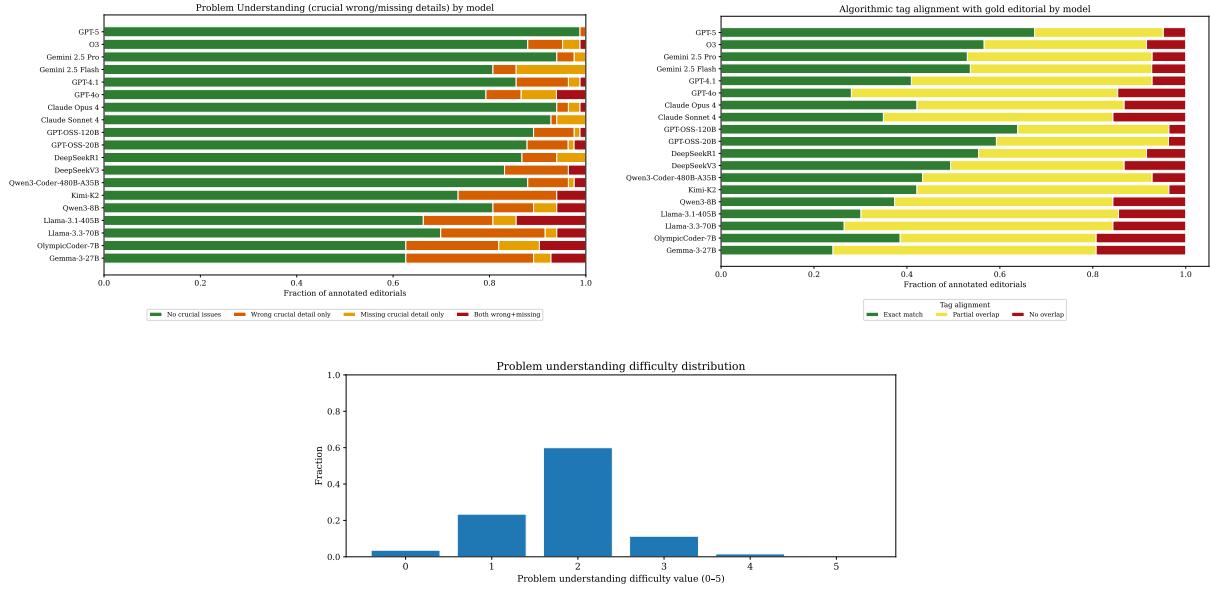


Figure 23: Additional LLM-judge diagnostics for **w/GenEd** editorials. Top-left: prevalence of crucial problem-understanding issues (wrong/missing details). Top-right: algorithm-tag alignment between generated and gold editorials (exact match / partial overlap / no overlap).

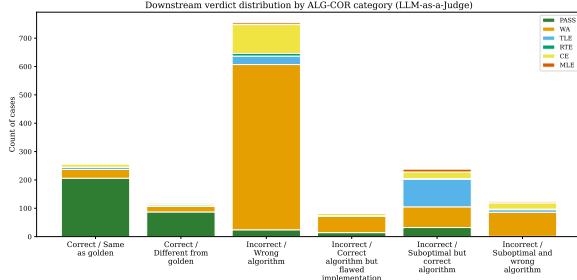


Figure 24: Downstream verdict counts (PASS/WA/TLE/RTE/CE/MLE) for **w/GenEd** code, stratified by the full six-way ALG-COR label. This complements Figure 5 by showing absolute sample sizes per category.

L.4 Full verdict breakdown by six-way ALG-COR

In Figure 5 we collapse ALG-COR into three groups for readability. Figure 24 provides the full six-way breakdown, which makes the mapping from fine-grained judge diagnoses to runtime outcomes explicit.

L.5 Correlates of downstream success under generated editorials

To complement the qualitative validation of the LLM judge (Table 2). For each model, across the 83 problems, we correlate each LLM-judge rubric field for the generated editorial with the binary indicator $\mathbb{1}[T(C) = \text{PASS}]$ for the resulting code. Binary fields use the ϕ coefficient; ordinal/numeric fields use Spearman's ρ . Stars denote two-sided

significance: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$. “–” indicates the field had no variance for that model (correlation undefined).

Overall, ALG-COR_overall is the strongest and most consistent correlate of passing for *every* model (all significant), supporting our use of judge-labeled correctness as a scalable proxy for reasoning quality. Tag alignment (tag_equal, tag_jaccard) is a weaker but often significant signal, while PU-D_value (annotator-rated understanding difficulty) is frequently negatively associated with success. Length features (editorial_words, code_lines) are negatively correlated for several models, which we interpret primarily as a difficulty confound rather than a causal effect of verbosity.

M Extended Related Work

M.1 Competitive Programming Benchmarks

End-to-end code generation and test-suite quality. A large fraction of LLM-for-code evaluation uses an end-to-end setup where models translate problem statements into code and are scored by unit tests. APPS (Hendrycks et al., 2021) and HumanEval (Chen et al., 2021) are canonical benchmarks in this paradigm. Because unit tests can be incomplete, multiple works aim to strengthen correctness evaluation by augmenting tests; EvalPlus (Liu et al., 2023) expands HumanEval/MBPP test suites and shows that stronger tests can change

Table 9: Per-model correlations between LLM-judge editorial diagnostics and downstream success under **w/GenEd**. Top row reports each model’s **w/GenEd pass@1** for context; subsequent rows report correlation with $\mathbb{W}[T(C) = \text{PASS}]$. Binary fields use ϕ ; ordinal/numeric fields use Spearman’s ρ . Stars denote two-sided significance: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$. “–” indicates undefined correlation due to zero variance.

	Claude Sonnet 4 (medium)	O3 8B	Owen3 480B-A35B	Owen3-Coder Opus 4	Claude DeepSeek V3	DeepSeek R1	Gemini 2.5 Flash	Gemini-3 2.5 Pro	GPT 27B	GPT 4.1	GPT 40	GPT 5	Llama 3.3-70B	Llama 3.1-405B	Kimi K2	Olympic Coder-7B	GPT-OSS 120B	GPT-OSS 20B	
Pass@1	0.193	0.458	0.130	0.110	0.309	0.100	0.475	0.384	0.450	0.024	0.175	0.037	0.679	0.060	0.024	0.133	0.086	0.313	0.295
ALG-COR_overall1	0.644***	0.640***	0.719***	0.738***	0.581***	0.614***	0.659***	0.708***	0.772***	0.703***	0.671***	0.444***	0.654***	0.765***	0.391***	0.790***	0.469***	0.790***	0.711***
PU_X_ord	0.133	0.171	0.145	0.053	-0.101	0.180	–	-0.149	0.016	-0.038	0.040	0.075	–	-0.079	-0.066	0.010	-0.108	0.063	0.051
PU-W_value	-0.054	-0.279*	-0.141	-0.123	-0.131	-0.140	-0.176	-0.066	-0.179	-0.114	-0.174	-0.077	-0.163	-0.157	-0.100	-0.235*	-0.200	-0.221*	-0.203
PU-D_value	-0.315**	-0.221*	-0.514***	-0.235*	-0.379***	-0.250*	-0.102	-0.194	-0.187	-0.319**	-0.165	-0.298**	-0.146	-0.228*	-0.189	-0.186	-0.282*	-0.180	-0.257*
tag_equal	0.347**	0.121	0.417***	0.240*	0.298**	0.167	0.330**	0.258*	0.234*	0.101	0.403***	0.168	0.150	0.077	-0.103	0.314**	0.210	0.184	0.326**
tag_jaccard	0.322**	0.195	0.333***	0.240*	0.225*	0.069	0.282*	0.297*	0.225*	-0.000	0.365***	0.168	0.078	-0.013	-0.147	0.303**	0.173	0.209	0.323**
editorial_words	-0.086	0.002	-0.086	0.034	-0.309**	-0.033	-0.197	-0.139	-0.146	-0.242*	-0.303**	0.005	-0.338**	-0.117	-0.277*	-0.335**	-0.061	-0.243*	-0.139
code_lines	-0.213	0.074	-0.064	-0.195	-0.283*	-0.285*	-0.348**	-0.178	-0.309**	-0.114	-0.186	-0.223*	-0.183	-0.230*	-0.227*	-0.152	-0.263*	-0.201	-0.217

absolute scores and even model rankings.

Contest-style benchmarks and competitive programming protocols. Contest-oriented evaluation draws problems from competitive platforms and uses contest-style judging. AlphaCode introduced competition-level generation and released the CodeContests dataset (Li et al., 2022). LiveCodeBench (Jain et al., 2025) improves temporal realism by continuously collecting fresh problems and explicitly discussing contamination. USACO (Shi et al., 2024) and LLM-ProS (Hossain et al., 2025) focus on higher difficulty distributions (Olympiad/ICPC-style), and CodeElo (Quan et al., 2025) evaluates via Codeforces-like submissions and maps results to human-comparable Elo ratings. Since evaluation quality depends critically on test coverage, CodeContests+ constructs higher-quality/verified test cases for the CodeContests problem set (Wang et al., 2025).

Decomposed evaluation and process benchmarks. Most benchmarks still primarily score the final program, making it difficult to localize failures to reasoning vs. implementation. Coding Triangle explicitly evaluates multiple dimensions of programming capability—editorial analysis, code implementation, and test-case generation (Zhang et al., 2025). ELABORATION benchmarks the broader human–LLM competitive programming process, introducing a taxonomy of human feedback and a protocol/dataset for staged interaction (Yang et al., 2025b). Our work shares the goal of decomposing the pipeline but focuses on a fully automated editorial-centric setting: editorials act as explicit intermediate plans that can be independently evaluated and transferred across models.

M.2 Code LLMs, Reasoning LLMs, and Multi-stage Pipelines

AlphaCode (Li et al., 2022) demonstrated that large-scale sampling and selection can yield strong con-

test performance, while code-specialized models (e.g., CodeGeeX (Zheng et al., 2024), StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2024)) improved implementation quality on standard code-generation benchmarks. Reasoning-oriented models such as OpenAI’s O-series (OpenAI et al., 2025) and DeepSeek-R1 (DeepSeek-AI et al., 2025a) further highlight the importance of multi-step reasoning for harder tasks.

Multi-stage methods aim to improve reliability by introducing explicit intermediate steps or feedback. MapCoder structures generation into retrieval, planning, coding, and debugging agents (Islam et al., 2024). CodeT selects among multiple candidate programs using generated tests (Chen et al., 2023). Our approach is complementary: we treat editorials as first-class intermediate artifacts, directly measuring plan quality and implementation fidelity, and enabling modular writer–coder composition through editorial transfer.

N Test-Time Feedback on Generated Editorials (Exploratory)

All results so far use a strictly single-shot setup, where a model generates one editorial E and one program C . We introduce a small ablation that adds limited test-time feedback *only in the w/GenEd setting*, allowing the model to revise its outputs using coarse signals.

Starting from $E^{(0)} = f_{\text{ed}}(P)$, $C^{(0)} = f_{\text{code}}(P, E^{(0)})$, we allow a bounded number of revisions to the editorial and/or the code. No test inputs or reference outputs are revealed; feedback is limited to judge verdicts or editorial-level self-assessment. We evaluate this setting on four representative open-weight models. Editorial refinement uses the same rubric as Appendix D. Since editorial-level algorithmic correctness strongly predicts downstream outcomes, it provides a natural signal for self-refinement.

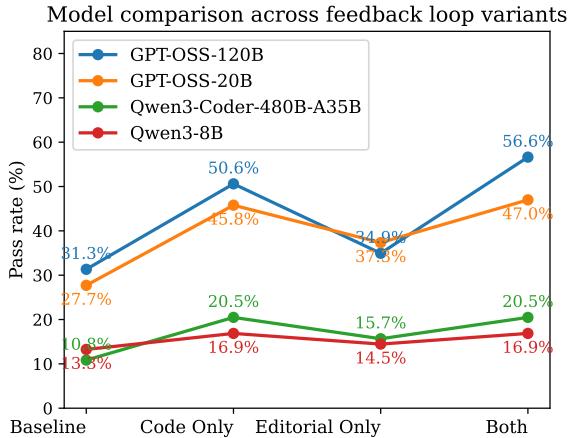


Figure 25: Pass@1 comparison across test-time feedback variants for four open-weight models. Baseline corresponds to w/GenEd without feedback.

Formalization. At iteration t , the model maintains a pair $(E^{(t)}, C^{(t)})$. Let $T(C)$ denote the judge verdict for a program, and $A(E)$ the self-assessment of an editorial. Updates differ by feedback variant.

Code feedback only. The editorial is fixed, and only the code is revised using verdict feedback: $E^{(t+1)} = E^{(0)}, C^{(t+1)} = f_{\text{code}}(P, E^{(0)}, C^{(t)}, T(C^{(t)}))$.

Editorial refinement only. The editorial is iteratively refined, and code is generated once at the end: $E^{(t+1)} = f_{\text{ed}}(P, E^{(t)}, A(E^{(t)})), C = f_{\text{code}}(P, E^{(T_E)})$.

Editorial refinement + code feedback. We first refine the editorial, then revise the code: $E^{(t+1)} = f_{\text{ed}}(P, E^{(t)}, A(E^{(t)})), C^{(0)} = f_{\text{code}}(P, E^{(T_E)}), C^{(k+1)} = f_{\text{code}}(P, E^{(T_E)}, C^{(k)}, T(C^{(k)}))$.

We use small fixed budgets, with $T_E, T_C \leq 5$.

Figure 25 reports pass@1 across variants. Code-only feedback yields the largest gains over w/GenEd, showing that coarse verdicts are effective for repairing implementations. Editorial refinement alone provides smaller but generally positive improvements. Combining both yields additional gains for GPT-OSS models but little benefit over code-only feedback for Qwen variants.

Findings. Limited test-time feedback can substantially improve performance. Most gains come from iterative code repair, while editorial refinement helps correct flawed plans and steer revisions toward viable solution strategies.

O Gold vs. Model-Generated Editorials Examples

Figures 26– 28 provides a detailed comparison between gold and model-generated editorials for a representative problem (DEPENDENCY FLOOD). Both editorials describe the same algorithm and satisfy the same correctness invariant. However, the model editorial is judged easier to understand because it explicitly linearizes the reasoning process: it first reframes the problem as longest-path maintenance, then isolates the accept/reject condition, and finally explains how local updates propagate and why they terminate within the given constraints.

Figures 29– 31 provides a detailed comparison between gold and model-generated editorials for a representative problem (HUNGRY PIPLUPS). Both the gold and model editorials describe the same algorithmic approach and are labeled **ALG-COR = Correct (Same as golden)**. The model editorial is easier to operationalize due to its explicit left-to-right backlog interpretation and step-by-step reasoning, whereas the gold editorial assumes familiarity with standard competitive-programming abstractions.

DEPENDENCY FLOOD: Full text with highlighted diagnostic evidence

Problem statement.

A university offers N courses, numbered from 1 to N in increasing order of difficulty.

To ensure that students acquire the necessary knowledge before taking advanced courses, the university may impose **course dependencies**. Each course dependency is represented by a pair of integers (u, v) ($1 \leq u < v \leq N$), meaning that students must complete course u before enrolling in course v .

A set of course dependencies is called **acceptable** if all N courses can be completed within K semesters, assuming that a student can take any number of courses (possibly zero) in a semester.

Formally, a set of course dependencies is acceptable if and only if there exists a sequence of N numbers a_1, a_2, \dots, a_N ($1 \leq a_i \leq K$) such that $a_u < a_v$ for every (u, v) in the set.

Let S be the set of course dependencies. Initially, S consists of M course dependencies $(A_1, B_1), (A_2, B_2), \dots, (A_M, B_M)$, and **it is guaranteed that S is acceptable**.

You need to process Q queries sequentially. The i -th query ($1 \leq i \leq Q$) is as follows:

- Given two integers C_i, D_i ($1 \leq C_i < D_i \leq N$), determine whether adding a new dependency (C_i, D_i) to S maintains its acceptability. If adding this dependency makes S unacceptable, output **reject** and leave S unchanged. Otherwise, output **accept** and add (C_i, D_i) to S .

Input. The first line of input contains three integers N , M and K ($2 \leq N \leq 2 \times 10^5$, $0 \leq M \leq 2 \times 10^5$, $1 \leq K \leq 100$), representing the number of courses, the number of initial course dependencies, and the number of semesters, respectively.

The i -th line ($1 \leq i \leq M$) of the next M lines contain two integers A_i and B_i ($1 \leq A_i < B_i \leq N$), denoting the i -th initial course dependency.

The following line contains an integer Q ($1 \leq Q \leq 2 \times 10^5$), the number of queries.

The i -th line ($1 \leq i \leq Q$) of the next Q lines contain two integers C_i and D_i ($1 \leq C_i < D_i \leq N$), representing the i -th query.

Output. Output Q lines. The i -th line ($1 \leq i \leq Q$) of the output should contain the result for the i -th query: Print **accept** if adding the dependency (C_i, D_i) keeps S acceptable, or **reject** if it makes S unacceptable.

Sample Input #1

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

Sample Output #1

```
reject
accept
reject
```

Sample Input #2

```
6 4 3
2 5
1 3
1 4
4 6
8
2 6
3 4
4 5
1 3
2 4
3 6
2 3
5 6
```

Sample Output #2

```
accept
reject
accept
accept
accept
accept
accept
reject
```

Figure 26: Full problem, gold editorial, and model-generated editorial for DEPENDENCY FLOOD, with highlighted diagnostic evidence. (part 1)

Gold editorial.

For simplicity, we will use graph theory terminology instead of the definitions given in the problem statement. Note that acceptability is equivalent to ensuring that there is no path of length K .

Let f_i ($1 \leq i \leq N$) be the maximum path length ending at vertex i , and let b_i ($1 \leq i \leq N$) be the maximum path length starting from vertex i . The values of f_i satisfy the following recurrence relation, where E denotes the set of edges:

$$f_i = \begin{cases} 1 + \max_{(j,i) \in E} f_j & \exists(j,i) \in E \\ 0 & \text{otherwise} \end{cases}$$

Thus, the initial values of f_i can be computed in $\mathcal{O}(N + M)$ time by processing vertices in ascending order. Similarly, the initial values of b_i can be computed in descending order.

If we can efficiently maintain the values of f_i and b_i , checking whether a new edge can be added becomes straightforward using the following lemma:

Lemma. A new edge (u, v) can be added if and only if $f_u + b_v + 1 < K$.

Proof. Exercise.

The key challenge is efficiently updating f_i and b_i when a new edge (u, v) is added. We describe the update process for f_i (the update for b_i follows analogously).

Given the way f_i is initially computed, we can derive an update algorithm using a simple DFS, as shown in the following pseudocode:

```
Function update(i, val):
    f[i] = val
    for each edge (i, j):
        if f[j] < f[i] + 1:
            update(j, f[i] + 1)
Function add_new_edge(u, v):
    if f[v] < f[u] + 1:
        update(v, f[u] + 1)
```

This recursive algorithm appears to require $\mathcal{O}(N + M)$ time per edge addition. However, we observe that the function `update` is called at most $(K - 1)N$ times in total. This is because:

- Every call of the function increases some f_i by at least one.
- The sum of all f_i values never exceeds $(K - 1) \times N$.
- The values of f_i never decrease.

Hence we can solve this problem in the overall time complexity of $\mathcal{O}(Q + K(N + M))$.

Model editorial (DeepSeek-R1)

Problem Analysis. The problem involves managing a set of course dependencies in a directed acyclic graph (DAG) where nodes represent courses and edges represent dependencies (an edge from u to v means course u must be completed before v). The goal is to process queries that add new dependencies while ensuring that the entire set of courses can be scheduled within K semesters.

The scheduling constraint requires that the longest chain of dependencies (a path of consecutive dependencies) does not exceed K courses, as each course in the chain must be in a distinct semester.

Key Insight. The core insight is that the acceptability of the dependency set is determined by the length of the longest chain (in terms of the number of nodes). A chain of length L requires at least L semesters. Thus, the dependency set is acceptable only if the longest chain in the DAG has length at most K .

For each query to add a new edge (C, D) , the critical check is whether the new chain formed by concatenating the longest chain ending at C and the longest chain starting at D (via the new edge) exceeds K nodes. Specifically, if the sum of the lengths of these two chains is at least $K + 1$, adding the edge creates a chain that is too long, making the set unacceptable.

Figure 27: Full problem, gold editorial, and model-generated editorial for DEPENDENCY FLOOD, with highlighted diagnostic evidence. (part 2)

Algorithm Selection.

- **Precomputation:**

- **Initial Graph Setup:** Store the initial dependencies in adjacency lists for outgoing and incoming edges.
- **Dynamic Programming Arrays:**
 - * $dp_in[u]$ represents the length of the longest chain ending at node u .
 - * $dp_out[u]$ represents the length of the longest chain starting at node u .
- Compute dp_in by processing nodes in increasing order (since edges go from lower to higher indices, this is a topological order). For each node u , $dp_in[u]$ is the maximum of 1 and $dp_in[v] + 1$ for all incoming edges (v, u) .
- Compute dp_out by processing nodes in decreasing order. For each node u , $dp_out[u]$ is the maximum of 1 and $dp_out[w] + 1$ for all outgoing edges (u, w) .

- **Query Processing:**

- For each query (C, D) :
 - * If the edge (C, D) already exists, output “accept” (no change).
 - * Otherwise, check if $dp_in[C] + dp_out[D] \geq K + 1$. If true, output “reject” (adding the edge would create a chain longer than K).
 - * If false, output “accept” and add the edge to the graph. Then update:
 - **Update dp_in for D :** Set $dp_in[D] = \max(dp_in[D], dp_in[C] + 1)$ if this new value is $\leq K$. Propagate this update to all descendants of D using BFS: for each descendant w , if $dp_in[u] + 1$ (where u is the current node) is greater than $dp_in[w]$ and $\leq K$, update $dp_in[w]$ and continue propagating.
 - **Update dp_out for C :** Set $dp_out[C] = \max(dp_out[C], dp_out[D] + 1)$ if this new value is $\leq K$. Propagate this update to all ancestors of C using BFS: for each ancestor w , if $dp_out[u] + 1$ is greater than $dp_out[w]$ and $\leq K$, update $dp_out[w]$ and continue propagating.

- **Complexity Analysis:**

- **Precomputation:**

- * Building adjacency lists: $O(N + M)$.
- * Computing dp_in and dp_out : $O(N + M)$ each.

- **Query Processing:**

- * Each query check: $O(1)$.
- * Edge addition and propagation: Each node can be updated at most K times (since chain lengths are bounded by K). Each update processes the node’s outgoing or incoming edges. The total work over all queries is $O(K \cdot (M + Q))$, where $M + Q$ is the total number of edges after all queries.
- **Overall Complexity:** $O(N + M + Q + K \cdot (M + Q))$. Given $K \leq 100$ and $N, M, Q \leq 2 \times 10^5$, this is efficient.

Explanation

- **Initial Setup:** The initial graph is built from input, storing edges in adjacency lists and a set for quick lookup.
- **Precomputation:** The dp_in array is computed by processing nodes from 1 to N , updating each node’s value based on incoming edges. The dp_out array is computed by processing nodes from N down to 1, updating each node’s value based on outgoing edges.
- **Query Handling:** For each query:
 - If the edge exists, “accept” is outputted immediately.
 - Otherwise, the sum $dp_in[C] + dp_out[D]$ is checked. If it exceeds K , the edge is rejected.
 - If accepted, the edge is added to the graph, and BFS propagates updates to dp_in for D ’s descendants and dp_out for C ’s ancestors, ensuring all affected nodes reflect the new longest chains within the K constraint.
- **Efficiency:** The BFS propagation is bounded by K , ensuring each node is updated at most K times, making the solution efficient for large inputs. The overall approach efficiently maintains the DAG properties and checks the scheduling constraint dynamically.

Figure 28: Full problem, gold editorial, and model-generated editorial for DEPENDENCY FLOOD, with highlighted diagnostic evidence. (part 3)

HUNGRY_PIPLUPS: Full problem, editorials, and highlighted diagnosis

Problem Statement.

Near Snowpoint City, food has become scarce, and the local Piplups have been fighting over fishing rights on the many icebergs in the area. In Snowpoint Bay, there are an infinite number of icebergs arranged in a line. Every day, exactly one Piplup journeys from the mainland to one of these icebergs, making it their home and fishing ground.

Piplups are very territorial. If a piplup swims to an iceberg X that has already been claimed, he will be chased out by the piplup already on the island and will attempt to swim to iceberg $X + 1$ instead. This continues until he reaches an empty iceberg.

Professor Piplup, who has eaten enough gummis to maximize his IQ, has been observing and recording these Piplup migration patterns. Over N days, he has recorded, for the i^{th} day, exactly one piplup left for iceberg A_i . Some of his graduate students managed to steal some of his records to plan for their migration trip and have the following queries for you.

For each query the student only have records from days L_i to R_i , they assume that no Piplups left on days outside this range. As a group of X_i Piplups, they will start leaving one by one, all heading for iceberg 1. Not wanting to drift too far downstream, they wish to determine the furthest iceberg (i.e., the largest numbered iceberg) that any of them will end up at with the information they have. Note that despite being friends, **there must still be at most one piplup per iceberg.**

There are also updates as the professor occasionally remembers that he recorded the P_i^{th} Piplup going to a different iceberg, X_i , prompting an update to his records.

The students are unsure of the correct answer and seek your help in answering their queries. Please assist the students in determining the answer to their migration dilemma!

Task Summary.

Notably, the task is as such. Define an operation \oplus as

$$S \oplus x = S \cup \{\text{Smallest } y : y \notin S \text{ and } y \geq x\}.$$

Given N numbers $\{A_1, A_2, \dots, A_N\}$. Handle M queries:

- **Update:** Change $A_{P_i} = X_i$.
- **Query:** Find the X_i^{th} smallest number not in $(\dots((\emptyset \oplus A_{L_i}) \oplus \dots) \oplus A_{R_i})$.

Input.

The first line of input contains two integers N, M , as mentioned above.

The second line of input contains N integers A_1, A_2, \dots, A_N ($0 \leq A_i \leq N - 1$).

The third line of input contains M queries of two possible types:

- Type 0: 0 Pi Xi
- Type 1: 1 Li Ri Xi

For type 0 query, the value at P_i is changed to X_i (i.e. $A_{P_i} = X_i$).

For type 1 query, you are asked to output the position of the X_i^{th} piplup only considering piplups from day L_i to R_i inclusive.

Limits.

- $1 \leq N \leq 10^6$
- $1 \leq M \leq 5 \times 10^3$
- $1 \leq X_i, A_i \leq 5 \times 10^2$
- $1 \leq L_i \leq R_i \leq N$
- $1 \leq P_i \leq N$

Output.

For each type 1 query, output a single integer representing the position of the X_i^{th} piplup only considering piplups from day L_i to R_i inclusive.

Figure 29: Full problem, gold editorial, and model-generated editorial for HUNGRY PIPLUPS, with highlighted diagnostic evidence. (part 1)

Sample Explanation.

For the first type 1 query, the piplups are at icebergs 2, 3, 4, thus the 2nd piplup will be at iceberg 5.
 For the second type 1 query, the piplups are at icebergs 1, 2, 3, thus the 2nd piplup will be at iceberg 5.
 For the second type 1 query, the piplups are at icebergs 1, 2, 3, 4, 5, thus the 10th piplup will be at iceberg 15.

Sample Input #1

```
5 4
1 3 2 4 5
1 2 4 2
0 2 1
1 1 3 2
1 1 5 10
```

Sample Output #1

```
5
5
15
```

Gold editorial (full).

Let $S_{L,R}$ be the set of locations of piplups, if we only consider the piplups from days $[L, R]$. Let's first try to generate this set for all queries quickly for each query.

Observation 1: Firstly, note that the operation (piplups going to icebergs) is associative. I.e. instead of processing the piplups one by one from left to right, we can first process the left half, $S_{L,M}$ and the right half, $S_{M+1,R}$ and merge them together. The merging can be done simply in $\mathcal{O}(R - L)$ via a merge sort like operation. This motivates a segment tree like solution.

This can be done by constructing a normal segment tree on the range of $[1, N]$ and in each node storing $S_{s,e}$. Note that this takes $\mathcal{O}(N \log N)$ to construct. Unfortunately, for a $[L, R]$ query, it still takes $\mathcal{O}(N)$ to generate $S_{L,R}$ as the size of the sets may be up to $\mathcal{O}(N)$ and merging will take $\mathcal{O}(N)$.

Observation 2: Note that for a set $S_{L,R}$, it consists of consecutive segments of icebergs where the starting iceberg of the segment is $\leq \max A_i$. Assume otherwise, it means that there exists a segment that start at $X > \max A_i$, which means that iceberg $X - 1 \geq \max A_i$ is empty. In which case, consider the piplup at X it should have started at a position $\leq \max A_i$, and thus stop at $X - 1$ instead of X which is a contradiction.

Thus, instead of storing all the positions, we can reduce the size of the set to be $\mathcal{O}(\max A_i)$ and simply store the segments (StartIdx, Size). The merging operation should still be similar but take $\mathcal{O}(\max A_i)$.

Finally to answer the queries we can simply generate $S_{L,R}$ in $\mathcal{O}(\max A_i \times \log N)$ then iterate through the set until we encounter X_i empty icebergs, which can be done in $\mathcal{O}(\max A_i)$. For updates, we can do a normal segment tree update and regenerate the node sets in $\mathcal{O}(\max A_i \times \log N)$ as well.

Final complexity:

$$\mathcal{O}(N \log N + M \log N \times \max A_i).$$

Note: Alternative solutions that simply involves a min+index segment tree can also pass but may require some constant time optimisations.

Model editorial (GPT-5, full; verbatim content but LaTeX-formatted).**Restatement.**

We have an infinite line of icebergs labeled 1, 2, 3, Each recorded day i contributes one Piplup trying to claim iceberg A_i , and if it is already occupied he moves right to the first free iceberg $\geq A_i$. Over a range of days $[L, R]$, this produces a set S of occupied icebergs (one distinct iceberg per day). A group of X new Piplups then leaves for iceberg 1 (one at a time with the same “go to first free \geq start” rule), and we want the furthest iceberg any of them reaches. Equivalently, we want the X -th smallest positive integer not in S .

Additionally, records can be updated: a single $A[P]$ changes to a new value.

Constraints to exploit.

- N up to 10^6 (large array).
- M up to 5×10^3 (few total operations).
- All values A_i and X in queries are small: ≤ 500 . This is the key.

Key ideas.

(1) **The final set depends only on the multiset of starting points (order does not matter).** The insertion rule “place each item at the smallest free \geq its starting point” admits an equivalent left-to-right scan:

Figure 30: Full problem, gold editorial, and model-generated editorial for HUNGRY PIPLUPS, with highlighted diagnostic evidence. (part 2)

Maintain a backlog b of Piplups that have started but not yet been placed. Let $c[p]$ be the number of Piplups starting at position p . For $p = 1, 2, 3, \dots$:

- Add arrivals: $b \leftarrow b + c[p]$.
- If $b > 0$, occupy p and set $b \leftarrow b - 1$; otherwise, p is empty.

This reproduces the greedy placement: position p is the earliest free location for any still-unplaced Piplup whose start is $\leq p$.

(2) **Bounded region (only first 500 positions can have arrivals).** Let $\text{MAXA} = 500$. Then $c[p] = 0$ for all $p > \text{MAXA}$. After processing $p = \text{MAXA}$, the backlog drains deterministically: positions $\text{MAXA} + 1, \dots, \text{MAXA} + b$ are all occupied, and everything after that is empty.

(3) **Query answer from holes + tail.** Let:

- $\text{occPrefix} = \text{number of occupied positions among } 1.. \text{MAXA}$,
- $\text{holes} = \text{MAXA} - \text{occPrefix}$,
- $\text{tail} = \text{backlog } b \text{ after processing } p = \text{MAXA}$.

Then:

- If $X \leq \text{holes}$, the answer is the position of the X -th empty among $1.. \text{MAXA}$.
- Otherwise, the answer is $\text{MAXA} + \text{tail} + (X - \text{holes})$.

(4) **Data structure for dynamic range counts.** Maintain an ordered set of positions for each value $v \in [1, \text{MAXA}]$ to count occurrences in any range $[L, R]$ efficiently:

$$c[v] = \#\{i \in [L, R] : A[i] = v\}.$$

Support point updates by moving index P between sets.

Algorithm in detail.

Preprocessing. For each $v \in [1, \text{MAXA}]$, build an ordered set $S[v]$ containing all indices i such that $A[i] = v$.

Update operation Θ $P \leftarrow X$. Let $\text{old} = A[P]$. Remove P from $S[\text{old}]$, insert P into $S[X]$, and set $A[P] \leftarrow X$.

Query operation 1 $L \text{ R } X$. First compute $c[v]$ for all $v = 1.. \text{MAXA}$ using two order-statistic queries per v (count of indices in $[L, R]$). Then run the backlog scan for $p = 1.. \text{MAXA}$:

- $b \leftarrow b + c[p]$.
- If $b > 0$, occupy p and decrement b ; else, p is a hole (empty).

If the X -th hole occurs within $1.. \text{MAXA}$, output its position immediately. Otherwise, after the scan, output $\text{MAXA} + b + (X - \text{holes})$.

Why it works. The backlog scan is equivalent to the greedy “first free \geq start” placement: at each position p , after adding arrivals, $b > 0$ exactly when some unplaced Piplup has start $\leq p$, so the earliest free position must be p and should be occupied. After $p > \text{MAXA}$, there are no new arrivals, so the remaining backlog occupies a contiguous tail.

Complexity and limits. Per query:

$$\mathcal{O}(\text{MAXA} \log N) \text{ to compute all } c[v] \quad + \quad \mathcal{O}(\text{MAXA}) \text{ for the scan.}$$

Per update: $\mathcal{O}(\log N)$. With $\text{MAXA} = 500$ and $M \leq 5000$, this is well within limits.

Notes and edge cases. This approach relies on the constraint $\max A_i \leq 500$. If A_i can be 0, either start the scan at $p = 0$ or shift indices by +1.

Worked example (sample).

- Initial $A = [1, 3, 2, 4, 5]$, query $(L, R, X) = (2, 4, 2)$: occupied at 2, 3, 4, so the 2-nd missing is 5.
- Update $P = 2$ to 1: $A = [1, 1, 2, 4, 5]$. Query $(1, 3, 2)$: occupied at 1, 2, 3, so the 2-nd missing is 5.
- Query $(1, 5, 10)$: occupied at 1, 2, 3, 4, 5, so the 10-th missing is 15.

Conclusion. Aggregate counts by starting positions, simulate occupancy with the backlog scan on the bounded region $1.. \text{MAXA}$, and answer the X -th missing via holes + tail. With ordered sets per value, both updates and range queries are efficient.

Figure 31: Full problem, gold editorial, and model-generated editorial for HUNGRY PIPLUPS, with highlighted diagnostic evidence. (part 3)